On Ensemble Learning

Mark Stamp and Aniket Chandak and Gavin Wong and Allen Ye

Abstract In this chapter, we consider ensemble classifiers, that is, machine learning based classifiers that utilize a combination of scoring functions. We provide a framework for categorizing such classifiers, and we outline several ensemble techniques, discussing how each fits into our framework. From this general introduction, we then pivot to the topic of ensemble learning within the context of malware analysis. We present a brief survey of some of the ensemble techniques that have been used in malware (and related) research. We conclude with an extensive set of experiments, where we apply ensemble techniques to a large and challenging malware dataset. While many of these ensemble techniques have appeared in the malware literature, previously there has been no way to directly compare results such as these, as different datasets and different measures of success are typically used. Our common framework and empirical results are an effort to bring some sense of order to the chaos that is evident in the evolving field of ensemble learning—both within the narrow confines of the malware analysis problem, and in the larger realm of machine learning in general.

1 Introduction

In ensemble learning, multiple learning algorithms are combined, with the goal of improved accuracy as compared to the individual algorithms. Ensemble techniques are widely used, and as a testament to their strength, ensembles have won numerous machine learning contests in recent years, including the KDD Cup [15], the Kaggle competition [14], and the Netflix prize [26].

Many such ensembles resemble Frankenstein's monster [33], in the sense that they are an agglomeration of disparate components, with some of the components being of questionable value—an "everything and the kitchen sink" approach clearly prevails. This effect can be

Mark Stamp
San Jose State University, San Jose, California, e-mail: mark.stamp@sjsu.edu
Aniket Chandak
San Jose State University, San Jose, California, e-mail: aniket.chandak@sjsu.edu
Gavin Wong
TBD, San Jose, California, e-mail: gavinmwong@gmail.com
Allen Ye
TBD, San Jose, California, e-mail: allenye66@gmail.com

clearly observed in the aforementioned machine learning contests, where there is little (if any) incentive to make systems that are efficient or practical, as accuracy is typically the only criteria for success. In the case of the Netflix prize, the winning team was awarded $1,000,000, yet Netflix never implement the winning scheme, since the improvements in accuracy "did not seem to justify the engineering effort needed to bring them into a production environment" [3]. In real-world systems, practicality and efficiency are necessarily crucial factors.

In this chapter, we provide a straightforward framework for categorizing ensemble techniques. We then consider specific (and relatively simple) examples of various categories of such ensembles, and we show how these fit into our framework. For various examples of ensembles, we also provide experimental results, based on a large and diverse malware dataset.

While many of the techniques that we consider have previously appeared in the malware literature, we are not aware of any comparable study focused on the effectiveness of various ensembles using a common dataset and common measures of success. While we believe that these examples are interesting in their own right, they also provide a basis for discussing various tradeoffs between measures of accuracy and practical considerations.

The remainder of this chapter is organized as follows. In Section 2 we discuss ensemble classifiers, including our framework for categorizing such classifiers. Section 3 contains our experimental results and some discussion of these results. This section also includes a discussion of our dataset, scoring metrics, software used, and so on. Finally, Section 4 concludes the paper and includes suggestions for future work.

2 Ensemble Classifiers

In this section, we first give a selective survey of some examples of malware (and closely related) research involving ensemble learning. Then we provide a framework for discussing ensemble classifiers in general.

2.1 Examples of Related Work

The paper [18] discusses various ways to combine classifiers and provides a theoretical framework for such combinations. The focus is on straightforward combinations, such as a maximum, sum, product, majority vote, and so on. The work in [18] has clearly been influential, but it seems somewhat dated, given the wide variety of ensemble methods that are used today.

The book [20] presents the topic of ensemble learning from a similar perspective as [18] but in much more detail. Perhaps not surprisingly, the more recent book [62] seems to have a somewhat more modern perspective with respect to ensemble methods, but retains the theoretical flavor of [20] and [18]. The brief blog at [35] provides a highly readable (if highly selective) summary of some of the topics covered in the books [20] and [62].

Here, we take an approach that is, in some sense, more concrete than that in [18, 20, 62]. Our objective is to provide a relatively straightforward framework for categorizing and discussing ensemble techniques. We then use this framework as a frame of reference for experimental results based on a variety of ensemble methods.

Table 1 provides a summary of several research papers where ensemble techniques have been applied to security-related problems. The emphasis here is on malware, but we have also included a few closely related topics. In any case, this represents a small sample of the many papers that have been published, and is only intended to provide an indication as to the types and variety of ensemble strategies that have been considered to date. On this list, we see examples of ensemble methods based on bagging, boosting, and stacking, as discussed below in Section 2.3.

Table 1: Security research papers using ensemble classifiers

Authors	**Application**	**Features**	**Ensemble**
Alazab et al. [2]	Detection	API calls	Neural networks
Comar et al. [8]	Detection	Network traffic	Random forest
Dimjaševic et al. [9]	Android	System calls	RF and SVM
Guo et al. [10]	Detection	API calls	BKS
Idrees et al. [12]	Android	Permissions, intents	RF and others
Jain & Meena [13]	Detection	Byte *n*-grams	AdaBoost
Khan et al. [17]	Detection	Network based	Boosting
Kong & Yan [19]	Classification	Function call graph	Boosting
Morales et al. [24]	Android	Permissions	Several
Narouei et al. [25]	Detection	DLL dependency	Random forest
Shahzad et al. [31]	Detection	Opcodes	Voting
Sheen et al. [32]	Various	Detection efficiency	Pruning
Singh et al. [34]	Detection	Opcodes	SVM
Smutz & Stavrou [36]	Malicious PDF	Metadata	Random forest
Toolan & Carthy [40]	Phishing	Various	C5.0, boosting
Ye et al. [58]	Detection	API calls, strings	SVM, bagging
Ye et al. [59]	Categorization	Opcodes	Clustering
Yerima et al. [60]	Zero day	179 features	RF, regression
Zhang et al. [61]	Detection	*n*-grams	Dempster-Shafer

2.2 A Framework for Ensemble Classifiers

In this section, we consider various means of constructing ensemble classifiers, as viewed from a high-level perspective. We then provide an equally high level framework that we find useful in our subsequent discussion of ensemble classifiers in Sections 2.3 and, especially, in Section 2.4.

We consider ensemble learners that are based on combinations of scoring functions. In the general case, we assume the scoring functions are real valued, but the more restricted case of zero-one valued "scoring" functions (i.e., classifiers) easily fits into our framework. We place no additional restrictions on the scoring functions and, in particular, they do not necessarily represent "learning" algorithms, per se. Hence, we are dealing with ensemble methods broadly speaking, rather than ensemble learners in a strict sense. We assume that the ensemble method itself—as opposed to the scoring functions that comprise the ensemble—is for classification, and hence ensemble functions are zero-one valued.

Let $\omega_1, \omega_2, \ldots, \omega_n$ be training samples, and let v_i be a feature vector of length m, where the features that comprise v_i are extracted from sample ω_i. We collect the feature vectors for all n training samples into an $m \times n$ matrix that we denote as

$$V = \begin{pmatrix} v_1 & v_2 & \cdots & v_n \end{pmatrix} \tag{1}$$

where each v_i is a column of the matrix V. Note that each row of V corresponds to a specific feature type, while column i of V corresponds to the features extracted from the training sample ω_i.

Let $S : \mathbb{R}^m \to \mathbb{R}$ be a scoring function. Such a scoring function will be determined based on training data, where this training data is given by a feature matrix V, as in equation (1). A scoring function S will generally also depend on a set of k parameters that we denote as

$$\Lambda = \begin{pmatrix} \lambda_1 & \lambda_2 & \ldots & \lambda_k \end{pmatrix} \tag{2}$$

The score generated by the scoring function S when applied to sample x is given by

$$S(x;V,\Lambda)$$

where we have explicitly included the dependence on the training data V and the function parameters Λ.

For any scoring function S, there is a corresponding classification function that we denote as $\widehat{S} : \mathbb{R}^m \to \{0,1\}$. That is, once we determine a threshold to apply to the scoring function S, it provides a binary classification function that we denote as $\widehat{S}$. As with S, we explicitly indicate the dependence on training data V and the function parameters Λ by writing

$$\widehat{S}(x;V,\Lambda).$$

For example, each training sample ω_i could be a malware executable file, where all of the ω_i belong to the same malware family. Then an example of an extracted feature v_i would be the opcode histogram, that is, the relative frequencies of the mnemonic opcodes that are obtained when ω_i is disassembled. The scoring function S could, for example, be based on a hidden Markov model that is trained on the feature matrix V as given in equation (1), with the parameters Λ in equation (2) being the initial values that are selected when training the HMM.

In its most general form, an ensemble method for a binary classification problem can be viewed as a function $F : \mathbb{R}^\ell \to \{0,1\}$ of the form

$$F\big(S_1(x;V_1,\Lambda_1), S_2(x;V_2,\Lambda_2), \ldots, S_\ell(x;V_\ell,\Lambda_\ell)\big) \tag{3}$$

That is, the ensemble method defined by the function F produces a classification based on the scores $S_1, S_2, \ldots, S_\ell$, where scoring function S_i is trained using the data V_i and parameters Λ_i.

2.3 Classifying Ensemble Classifiers

From a high level perspective, ensemble classifiers can be categorized as bagging, boosting, stacking, or some combination thereof [20, 35, 62]. In this section, we briefly introduce each of these general classes of ensemble methods and give their generic formulation in terms of equation (3).

2.3.1 Bagging

In bootstrap aggregation (i.e., bagging), different subsets of the data or features (or both) are used to generate different scores. The results are then combined in some way, such as a sum of the scores, or a majority vote of the corresponding classifications. For bagging we assume that the same scoring method is used for all scores in the ensemble. For example, bagging is used when generating a random forest, where each individual scoring function is based on a decision tree structure. One benefit of bagging is that it reduces overfitting, which is a particular problem for decision trees.

For bagging, the general equation (3) is restricted to

$$F\big(S(x;V_1,\Lambda),S(x;V_2,\Lambda),\ldots,S(x;V_\ell,\Lambda)\big) \tag{4}$$

That is, in bagging, each scoring function is essentially the same, but each is trained on a different feature set. For example, suppose that we collect all available feature vectors into a matrix V as in equation (1). Then bagging based on subsets of samples would correspond to generating V_i by deleting a subset of the columns of V. On the other hand, bagging based on features would correspond to generating V_i by deleting a subset of the rows of V. Of course, we can easily extend this to bagging based on both the data and features simultaneously, as in a random forest. In Section 2.4, we discuss specific examples of bagging.

2.3.2 Boosting

Boosting is a process whereby distinct classifiers are combined to produce a stronger classifier. Generally, boosting deals with weak classifiers that are combined in an adaptive or iterative manner so as to improve the overall classifier. We restrict our definition of boosting to cases where the classifiers are closely related, in the sense that they differ only in terms of parameters. From this perspective, boosting can be viewed as "bagging" based on classifiers, rather than data or features. That is, all of the scoring functions are reparameterized versions of the same scoring technique. Under this definition of boosting, the general equation (3) becomes

$$F\big(S(x;V,\Lambda_1),S(x;V,\Lambda_2),\ldots,S(x;V,\Lambda_\ell)\big) \tag{5}$$

That is, the scoring functions differ only by re-parameterization, while the scoring data and features do not change.

Below, in Section 2.4, we discuss specific examples of boosting; in particular, we discuss the most popular method of boosting, AdaBoost. In addition, we show that some other popular techniques fit our definition of boosting.

2.3.3 Stacking

Stacking is an ensemble method that combines disparate scores using a meta-classifier [35]. In this generic form, stacking is defined by the general case in equation (3), where the scoring functions can be (and typically are) significantly different. Note that from this perspective, stacking is easily seen to be a generalization of both bagging and boosting.

Because stacking generalizes both bagging and boosting, it is not surprising that stacking based ensemble methods can outperform bagging and boosting methods, as evidenced by recent machine learning competitions, including the KDD Cup [15], the Kaggle competition [14], as well as the infamous Netflix prize [26]. However, this is not the end of the story, as efficiency and practicality are often ignored in such competitions, whereas in practice, it is virtually always necessary to consider such issues. Of course, the appropriate tradeoffs will depend on the specifics of the problem at hand. Our empirical results in Section 3 provide some insights into these tradeoff issues within the malware analysis domain.

In the next section, we discuss concrete examples of bagging, boosting, and stacking techniques. Then in Section 3 we present our experimental results, which include selected bagging, boosting, and stacking architectures.

2.4 Ensemble Classifier Examples

Here, we consider a variety of ensemble methods and discuss how each fits into the general framework presented above. We begin with a few fairly generic examples, and then discuss several more specific examples.

2.4.1 Maximum

In this case, we have

$$F\big(S_1(x;V_1,\Lambda_1),S_2(x;V_2,\Lambda_2),\ldots,S_\ell(x;V_\ell,\Lambda_\ell)\big) = \max\{S_i(x;V_i,\Lambda_i)\} \tag{6}$$

2.4.2 Averaging

Averaging is defined by

$$F\big(S_1(x;V_1,\Lambda_1),S_2(x;V_2,\Lambda_2),\ldots,S_\ell(x;V_\ell,\Lambda_\ell)\big) = \frac{1}{\ell}\sum_{i=1}^{\ell} S_i(x;V_i,\Lambda_i) \tag{7}$$

2.4.3 Voting

Voting could be used as a form of boosting, provided that no bagging is involved (i.e., the same data and features are used in each case). Voting is also applicable to stacking, and is generally applied in such a mode, or at least with significant diversity in the scoring functions, since we want limited correlation when voting.

In the case of stacking, a simple majority vote is of the form

$$\begin{aligned}F\big(\widehat{S}_1(x;V_1,\Lambda_1),\widehat{S}_2(x;V_2,\Lambda_2),\ldots,\widehat{S}_\ell(x;V_\ell,\Lambda_\ell)\big)\\ = \operatorname{maj}\big(\widehat{S}_1(x;V_1,\Lambda_1),\widehat{S}_2(x;V_2,\Lambda_2),\ldots,\widehat{S}_\ell(x;V_\ell,\Lambda_\ell)\big)\end{aligned}$$

where "maj" is the majority vote function. Note that the majority vote is well defined in this case, provided that ℓ is odd—if ℓ is even, we can simply flip a coin in case of a tie.

As an aside, we note that it is easy to see why we want to avoid correlation when voting is used as a combining function. Consider the following example from [47]. Suppose that we have the three highly correlated scores

$$\begin{pmatrix} \widehat{S}_1 \\ \widehat{S}_2 \\ \widehat{S}_3 \end{pmatrix} = \begin{pmatrix} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \end{pmatrix}$$

where each 1 indicates correct classification, and each 0 is an incorrect classification. Then, both $\widehat{S}_1$ and $\widehat{S}_2$ are 80% accurate, and $\widehat{S}_3$ is 70% accurate. If we use a simple majority vote, then we obtain the classifier

$$C = (\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ )$$

which is 80% accurate. On the other hand, the less correlated classifiers

$$\begin{pmatrix} \widehat{S}_1' \\ \widehat{S}_2' \\ \widehat{S}_3' \end{pmatrix} = \begin{pmatrix} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0 \\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1 \\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1 \end{pmatrix}$$

are only 80%, 70% and 60% accurate, respectively, but the majority vote in this case gives us

$$C' = (\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ )$$

which is 90% accurate.

2.4.4 ML-Based Combination

Recall that the most general formulation of an ensemble classifier is given in equation (3). In this formulation, we can select the function F based on a machine learning technique, which is applied to the individual scores $S(x;V_i,\Lambda_i)$. In the remainder of this section, we consider specific ensemble examples involving machine learning techniques.

2.4.5 AdaBoost

Given a collection of (weak) classifiers $c_1,c_2,\ldots,c_\ell$, AdaBoost is an iterative algorithm that generates a series of (generally, stronger) classifiers, $C_1,C_2,\ldots,C_M$ based on the classifiers c_i. Each classifier is determined from the previous classifier by the simple linear extension

$$C_m(x) = C_{m-1}(x) + \alpha_m c_i(x)$$

and the final classifier is given by $C = C_M$. Note that at each iteration, we include a previously unused c_i from the set of (weak) classifiers and determine a new weight α_i. A greedy approach is used when selecting c_i, but it is not a hill climb, so that results might get worse at any step in the AdaBoost process.

From this description, we see that the AdaBoost algorithm fits the form in equation (5), with $\widehat{S}(x;V,\Lambda_i) = C_i(x)$, and

$$F\big(\widehat{S}(x;V,\Lambda_1),\widehat{S}(x;V,\Lambda_2),\ldots,\widehat{S}(x;V,\Lambda_M)\big) = \widehat{S}(x;V,\Lambda_M) = C_M(x)$$

2.4.6 SVM as Meta-Classifier

It is natural to use an SVM as a meta-classifier to combine scores [38]. For example, in [34], an SVM is used to generate a malware classifier based on several machine learning and statistical based malware scores. In [34], it is shown that the resulting SVM classifier consistently outperforms any of the component scores, and the differences are most pronounced in the most challenging cases.

The use of SVM in this meta-classifier mode can be viewed as a general stacking method. Thus, this SVM technique is equivalent to equation (3), where the function F is simply an SVM classifier based on the component scores $S_i(x;V_i,\Lambda_i)$, for $i = 1,2,\ldots,\ell$.

2.4.7 HMM with Random Restarts

A hidden Markov model can be viewed as a discrete hill climb technique [37, 38]. As with any hill climb, when training an HMM we are only assured of a local maximum, and we can often significantly improve our results by executing the hill climb multiple times with different initial values, selecting the best of the resulting models. For example, in [51] it is shown that an HMM can be highly effective for breaking classic substitution ciphers and, furthermore, by using a large number of random restarts, we can significantly increase the success rate in the most difficult cases. The work in [51] is closely related to that in [7], where such an approach is used to analyze the unsolved Zodiac 340 cipher.

From the perspective considered in this paper, an HMM with random restarts can be seen as special case of boosting. If we simply select the best model, then the "combining" function is particularly simple, and is given by

$$F\big(S(x;V,\Lambda_1),S(x;V,\Lambda_2),\ldots,S(x;V,\Lambda_\ell)\big) = \max\{S(x;V,\Lambda_i)\} \tag{8}$$

Here, each scoring function is an HMM, where the trained models differ based only on different initial values. We see that equation (8) is a special case of equation (6). However, the "max" in equation (8) is the maximum over the HMM model scores, not the maximum over any particular set of input values. That is, we select the highest scoring model and use it for scoring. Of course, we could use other combining functions, such as an average or majority vote of the corresponding classifiers. In any case, since there is a score associated with each model generated by an HMM, any such combining function is well-defined.

2.4.8 Bagged Perceptron

Like a linear SVM, a perceptron will separate linearly separable data. However, unlike an SVM, a perceptron will not necessarily produce the optimal separation, in the sense of maximizing the margin. If we generate multiple perceptrons, each with different random initial weights, and then average these models, the resulting classifier will tend to be nearer to optimal, in the sense of maximizing the margin [21, 47]. That is, we construct a classifier

$$F\big(S(x;V,\Lambda_1),S(x;V,\Lambda_2),\ldots,S(x;V,\Lambda_\ell)\big) = \frac{1}{\ell}\sum_{i=1}^{\ell} S(x;V,\Lambda_i) \tag{9}$$

where S is a perceptron and each P_i represents a set of initial values. We see that equation (9) is a special case of the averaging example given in equation (7). Also, we note that in this sum, we are averaging the perceptron models, not the classifications generated by the models.

Although this technique is sometimes referred to as "bagged" perceptrons [47], by our criteria, it is a boosting scheme. That is, the "bagging" here is done with respect to parameters of the scoring functions, which is our working definition of boosting.

2.4.9 Bagged Hidden Markov Model

Like the HMM with random restarts example given above, in this case, we generate multiple HMMs. However, here we leave the model parameters unchanged, and simply train each on a subset of the data. We could then average the model scores (for example) as a way of combining the HMMs into a single score, from which we can easily construct a classifier.

2.4.10 Bagged and Boosted Hidden Markov Model

Of course, we could combine both the HMM with random restarts discussed in Section 2.4.7 with the bagging approach discussed in the previous section. This process would yield an HMM-based ensemble technique that combines both bagging and boosting.

3 Experiments and Results

In this section, we consider a variety of experiments that illustrate various ensemble techniques. There experiments involve malware classification, based on a challenging dataset that includes a large number of samples from a significant number of malware families.

3.1 Dataset and Features

Our dataset consists of samples from the 21 malware families listed in Table 2. These families are from various different types of malware, including Trojans, worms, backdoors, password stealers, so-called VirTools, and so on.

Each of the malware families in Table 2 is summarized below.

Adload downloads an executable file, stores it remotely, executes the file, and disables proxy settings [41].

Agent downloads Trojans or other software from a remote server [42].

Allaple is a worm that can be used as part of a denial of service (DoS) attack [52].

Table 2: Type of each malware family

Index	Family	Type	Index	Family	Type
1	Adload [41]	Trojan Downloader	12	Renos [43]	Trojan Downloader
2	Agent [42]	Trojan	13	Rimecud [54]	Worm
3	Allaple [52]	Worm	14	Small [44]	Trojan Downloader
4	BHO [45]	Trojan	15	Toga [46]	Trojan
5	Bifrose [4]	Backdoor	16	VB [6]	Backdoor
6	CeeInject [48]	VirTool	17	VBinject [50]	VirTool
7	Cycbot [5]	Backdoor	18	Vobfus [55]	Worm
8	FakeRean [53]	Rogue	19	Vundo [56]	Trojan Downloader
9	Hotbar [1]	Adware	20	Winwebsec [22]	Rogue
10	Injector [49]	VirTool	21	Zbot [23]	Password Stealer
11	OnLineGames [28]	Password Stealer	—	—	—

BHO can perform a variety of actions, guided by an attacker [45].

Bifrose is a backdoor Trojan that enables a variety of attacks [4].

CeeInject uses advanced obfuscation to avoid being detected by antivirus software [48].

Cycbot connects to a remote server, exploits vulnerabilities, and spreads through backdoor ports [5].

FakeRean pretends to scan the system, notifies the user of supposed issues, and asks the user to pay to clean the system [53].

Hotbar is adware that shows ads on webpages and installs additional adware [1].

Injector loads other processes to perform attacks on its behalf [49].

OnLineGames steals login information of online games and tracks user keystroke activity [28].

Renos downloads software that claims the system has spyware and asks for a payment to remove the nonexistent spyware [43].

Rimecud is a sophisticated family of worms that perform a variety of activities and can spread through instant messaging [54].

Small is a family of Trojans that downloads unwanted software. This downloaded software can perform a variety of actions, such as a fake security application [44].

Toga is a Trojan that can perform a variety of actions of the attacker's choice [46].

VB is a backdoor that enables an attacker to gain access to a computer [6].

VBinject is a generic description of malicious files that are obfuscated in a specific manner [50].

Vobfus is a worm that downloads malware and spreads through USB drives or other removable devices [55].

Vundo displays pop-up ads and may download files. It uses advanced techniques to defeat detection [56].

Winwebsec displays alerts that ask the user for money to fix supposed issues [22].

Zbot is installed through email and shares a user's personal information with attackers. In addition, Zbot can disable a firewall [23].

From each available malware sample, we extract the first 1000 mnemonic opcodes using the reversing tool Radare2 (also know as R2) [29]. We discard any malware executable that yields less than 1000 opcodes, as well as a number of executables that were found to be corrupted. The resulting opcode sequences, each of length 1000, serve as the feature vectors for our machine learning experiments.

Table 3 gives the number of samples (per family) from which we successfully obtained opcode feature vectors. Note that our dataset contains a total of 9725 samples from the 21 malware families and that the dataset is highly imbalanced—the number of samples per family varies from a low of 129 to a high of nearly 1000.

Table 3: Type of each malware family

Index	**Family**	**Samples**	**Index**	**Family**	**Samples**
1	Adload	162	12	Renos	532
2	Agent	184	13	Rimecud	153
3	Allaple	986	14	Small	180
4	BHO	332	15	Toga	406
5	Bifrose	156	16	VB	346
6	CeeInject	873	17	VBinject	937
7	Cycbot	597	18	Vobfus	929
8	FakeRean	553	19	Vundo	762
9	Hotbar	129	20	Winwebsec	837
10	Injector	158	21	Zbot	303
11	OnLineGames	210		Total	9725

3.2 Metrics

The metrics used to quantify the success of our experiments are accuracy, balanced accuracy, precision, recall, and the F1 score. Accuracy is simply the ratio of correct classifications to the total number of classifications. In contrast, the balanced accuracy is the average accuracy per family.

Precision, which is also known as the positive predictive value, is the number of true positives divided by the sum of the true positives and false positives. That is, the precision is the ratio of samples classified as positives that are actually positive to all samples that are classified as positive. Recall, which is also known as the true positive rate or sensitivity, is the computed by dividing the number of true positives by the number true positives plus the number of false negatives. That is, the recall is the fraction of positive samples that are classified as such. The F1 score is computed as

$$\mathrm{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}},$$

which is the harmonic mean of the precision and recall.

3.3 Software

The software packages used in our experiments include `hmmlearn` [11], `XGBoost` [57], `Keras` [16], and `TensorFlow` [39], and `scikit-learn` [30], as indicated in Table 4. In addition, we use `Numpy` [27] for linear algebra and various tools available in the package `scikit-learn` (also known as `sklearn`) for general data processing. These packages are all widely used in machine learning.

Table 4: Software used in experiments

Technique	**Software**
HMM	`hmmlearn`
XGBoost	`XGBoost`
AdaBoost	`sklearn`
CNN	`Keras`, `TensorFlow`
LSTM	`Keras`, `TensorFlow`
Random Forest	`sklearn`

3.4 Overview of Experiments

For all of our experiments, we use opcode sequences of length 1000 as features. For CNNs, the sequences are interpreted as images.

We consider three broad categories of experiments. First, we apply "standard" machine learning techniques. These experiments, serve as a baseline for comparison for our subsequent experiments. Among other things, these standard experiments show that the malware classification problem that we are dealing with is challenging.

We also conduct bagging and boosting experiments based on a subset of the techniques considered in our baseline standard experiments. These results demonstrate that both bagging and boosting can provide some improvement over our baseline techniques.

Finally, we consider a set of stacking experiments, where we restrict our attention to simple voting schemes, all of which are based on architectures previously considered in this paper. Although these are very basic stacking architectures, they clearly show the potential benefit of stacking multiple techniques.

3.5 Standard Techniques

For our "standard" techniques, we test several machine learning methods that are typically used individually. Specifically, we consider hidden Markov models (HMM), convolutional neural networks (CNN), random forest, and long short-term memory (LSTM). The parameters that we have tested in each of these cases are listed in Table 5, with those that gave the best results in boldface.

Table 5: Parameters for standard techniques

Technique	Parameters	Values tested
HMM	`n_components`	[1,2,5,**10**]
	`n_iter`	[50,100,**200**,300,500]
	`tol`	[0.01,0.5]
CNN	`learning_rate`	[**0.001**,0.0001]
	`batch_size`	[**32**,64,128]
	`epochs`	[50,75,**100**
Random Forest	`n_estimators`	[100,200,300,500,**800**]
	`min_samples_split`	[**2**,5,10,15,20]
	`min_samples_leaf`	[**1**,2,5,10,15]
	`max_features`	[**auto**,sqrt,$\log_2$]
	`max_depth`	[30,**40**,50,60,70,80]
LSTM	`layers`	[**1**,3]
	`directional`	[uni-dir,**bi-dir**]
	`learning_rate`	[**0.01**]
	`batch_size`	[**1**,16,32]
	`epochs`	[**20**]

From Table 5, we note that a significant number of parameter combinations were tested in each case. For example, in the case of our random forest model, we tested

$$5^3 \cdot 3 \cdot 6 = 2250$$

different combinations of parameters.

The confusion matrices for all of the experiments in this section can be found in the Appendix in Figure 2 (a) through Figure 2 (d). We present the results of all of these experiments—in terms of the metrics discussed previously (i.e., accuracy, balanced accuracy, precision, recall, and F1 score)—in Section 3.9, below.

3.6 Bagging Experiments

Recall from our discussion above, that we use the term bagging to mean a multi-model approach where the individual models are trained with the same technique and essentially the same parameters, but different subsets of the data or features. In contrast, we use boosting to refer to multi-model cases where the data and features are essentially the same and the models are of the same type, with the model parameters varied.

We will use AdaBoost and XGBoost results to serve as representative examples of boosting. We also consider bagging experiments (in the sense described in the previous paragraph) involving each of the HMM, CNN, and LSTM architectures. The results of these three distinct bagging experiments—in the form of confusion matrices—are given in Figure 3 in the Appendix. In terms of the metrics discussed above, the results of these experiments are summarized in Section 3.9, below.

3.7 Boosting Experiments

As representative examples of boosting techniques, we consider AdaBoost and XGBoost. In each case, we experiment with a variety of parameters as listed in Table 6. The parameter selection that yielded the best results are highlighted in boldface.

Table 6: Parameters for boosting techniques

Technique	Parameters	Values tested
AdaBoost	`n_estimators`	[100,200,300,500,800,**1000**]
	`learning_rate`	[0.5,1.0,1.5,2.0]
	`algorithm`	[**SAMME**,SAMME.R]
XGBoost	`eta`	[0.05,0.1,0.2,**0.3**,0.5]
	`max_depth`	[1,2,**3**,4]
	`objective`	[**multi:softprob**,binary:logistic]
	`steps`	[1,5,10,**20**,50]

Confusion matrices for these two boosting experiments are given in Figure 4 in the Appendix. The results of these experiments are summarized in Section 3.9, below, in terms of accuracy, balanced accuracy, and so on.

3.8 Voting Experiments

Since there exists an essentially unlimited number of possible stacking architectures, we have limited our attention to one of the simplest, namely, voting. These results serve as a lower bound on the results that can be obtained with stacking architectures.

We consider six different stacking architectures. These stacking experiments can be summarized as follows.

CNN consists of the plain and bagged CNN models discussed above. The confusion matrix for this experiment is given in Figure 5 (a).

LSTM consists of the plain and bagged LSTM models discussed above. The confusion matrix for this experiment is given in Figure 5 (b).

Bagged neural networks combines our bagged CNN and bagged LSTM models. The confusion matrix for this experiment is given in Figure 5 (c).

Classic techniques combines (via voting) all of the classic models considered above, namely, HMM, bagged HMM, random forest, AdaBoost, and XGBoost. The confusion matrix for this experiment is given in Figure 5 (d).

All neural networks consists of all of the CNN and LSTM models, bagged and plain. The confusion matrix for this experiment is given in Figure 5 (e).

All models combines all of the classic and neural network models into one voting scheme. The confusion matrix for this experiment is given in Figure 5 (f).

In the next section, we present the results for each of the voting experiments discussed in this section in terms of the our various metrics. These metrics enable us to directly compare all of our experimental results.

3.9 Discussion

Table 7 summarizes the results of all of the experiments discussed above, in term of the following metrics: accuracy, balanced accuracy, precision, recall, and F1 score. These metrics have been introduced in Section 3.1, above.

Table 7: Comparison of experimental results

Experiments	**Case**	**Accuracy**	**Balanced accuracy**	**Precision**	**Recall**	**F1 score**
Standard	HMM	0.6717	0.6336	0.7325	0.6717	0.6848
	CNN	0.8211	**0.7245**	**0.8364**	**0.8211**	0.8104
	Random Forest	0.7549	0.6610	0.7545	0.7523	0.7448
	LSTM	**0.8410**	0.7185	0.7543	0.7185	**0.8145**
Bagging	Bagged HMM	0.7168	0.6462	0.7484	0.7168	0.7165
	Bagged CNN	**0.8910**	**0.8105**	**0.9032**	**0.8910**	**0.8838**
	Bagged LSTM	0.8602	0.7754	0.8571	0.8602	0.8549
Boosting	AdaBoost	0.5378	0.4060	0.5231	0.5378	0.5113
	XGBoost	**0.7472**	**0.6636**	**0.7371**	**0.7472**	**0.7285**
Voting	Classic	0.8766	0.8079	0.8747	0.8766	0.8719
	CNN	0.9260	0.8705	0.9321	0.9260	0.9231
	LSTM	0.8560	0.7470	0.8511	0.8560	0.8408
	Bagged neural networks	$\boxed{\mathbf{0.9337}}$	$\boxed{\mathbf{0.8816}}$	$\boxed{\mathbf{0.9384}}$	$\boxed{\mathbf{0.9337}}$	$\boxed{\mathbf{0.9313}}$
	All neural networks	0.9208	0.8613	0.9284	0.9208	0.9171
	All models	0.9188	0.8573	0.9249	0.9188	0.9154

In Table 7, the best result for each type of experiment is in boldface, with the best results overall also being boxed. We see that a voting strategy based on all of the bagged neural network techniques gives us the best result for each of the five statistics that we have computed.

Since our dataset is highly imbalanced, we consider the balanced accuracy as the best measure of success. The balanced accuracy results in Table 7 are given in the form of a bar graph in Figure 1.

Note that the results in Figure 1 clearly show that stacking techniques are beneficial, as compared to the corresponding "standard" techniques. Stacking not only yields the best results, but it dominates in all categories. We note that five of the six stacking experiments perform better than any of the standard, bagging, or boosting experiments. This is particularly noteworthy since we only considered a simple stacking approach. As a results, our stacking experiments likely provide a poor lower bound on stacking in general, and more advanced stacking techniques may improve significantly over the results that we have obtained.

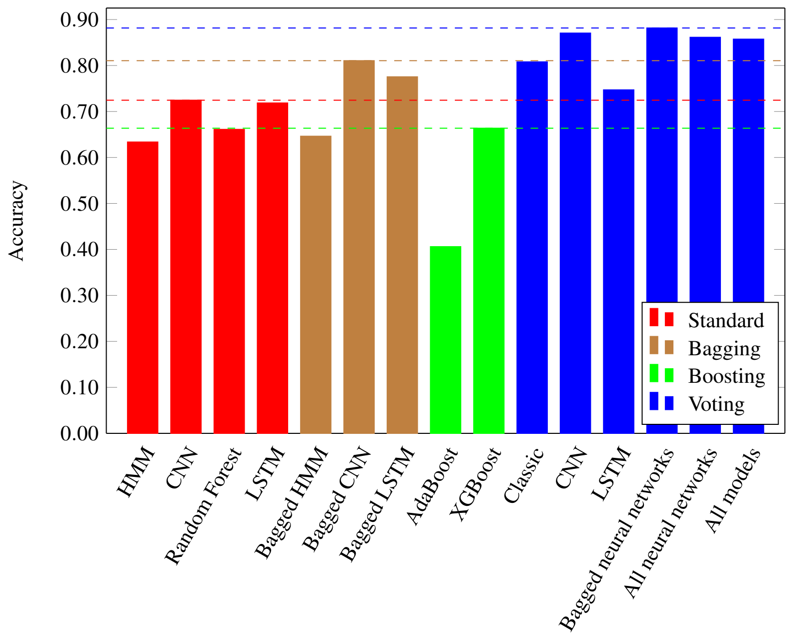


Fig. 1: Balanced accuracy results

4 Conclusion and Future Work

In this chapter, we have attempted to impose some structure on the field of ensemble learning. We showed that combination architectures can be classified as either bagging, boosting, or in the more general case, stacking. We then provided experimental results involving a challenging malware dataset to illustrate the potential benefits of ensemble architectures. Our results clearly show that ensembles improve on standard techniques, with respect to our specific dataset. Of course, in principle, we expect such combination architectures to outperform standard techniques, but it is instructive to confirm this empirically, and to show that the improvement can be substantial. These results make it clear that there is a reason why complex stacking architectures win machine learning competitions.

However, stacking models are not without potential pitfalls. As the architectures become more involved, training can become impractical. Furthermore, scoring can also become prohibitively costly, especially if large numbers of features are used in complex schemes involving extensive use of bagging or boosting.

For future work, it would be useful to quantify the tradeoff between accuracy and model complexity. While stacking will generally improve results, marginal improvements in accuracy that come at great additional cost in training and scoring are unlikely to be of any value in real world applications. More concretely, future work involving additional features would be very interesting, as it would allow for a more thorough analysis of bagging, and it would enable us to draw firmer conclusions regarding the relative merits of bagging and boosting. Of course, more more complex classes of stacking techniques could be considered.

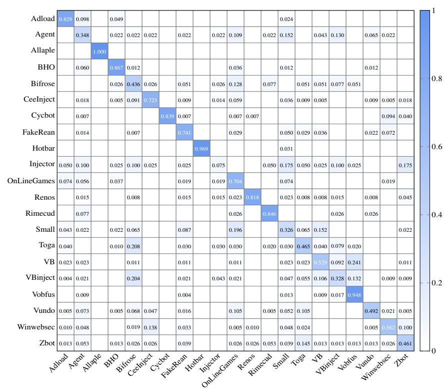
References

1. Adware:win32/hotbar. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Adware:Win32/Hotbar&threatId=6204.
2. Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Zero-day malware detection based on supervised learning algorithms of API call signatures. In *Proceedings of the Ninth Australasian Data Mining Conference*, volume 121 of *AusDM '11*, pages 171–182. Australian Computer Society, 2011.
3. Xavier Amatriain and Justin Basilico. Netflix recommendations: Beyond the 5 stars (part 1). https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429, 2012.
4. Backdoor:win32/bifrose. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Win32/Bifrose&threatId=-2147479537.
5. Backdoor:win32/cycbot.g. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Win32/Cycbot.G.
6. Backdoor:win32/vb. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Backdoor:Win32/VB&threatId=7275.
7. Taylor Berg-Kirkpatrick and Dan Klein. Decipherment with a million random restarts. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP 2013, pages 874–878, 2013.
8. Prakash Mandayam Comar, Lei Liu, Sabyasachi Saha, Pang-Ning Tan, and Antonio Nucci. Combining supervised and unsupervised learning for zero-day malware detection. In *2013 Proceedings IEEE INFOCOM*, pages 2022–2030. IEEE, 2013.
9. Marko Dimjaševic, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Android malware detection based on system calls. Technical Report UUCS-15-003, School of Computing, University of Utah, Salt Lake City, Utah, 2015.
10. Shanqing Guo, Qixia Yuan, Fengbo Lin, Fengyu Wang, and Tao Ban. A malware detection algorithm based on multi-view fusion. In *International Conference on Neural Information Processing*, ICONIP 2010, pages 259–266. Springer, 2010.
11. hmmlearn. https://hmmlearn.readthedocs.io/en/latest/.
12. Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M Chen, and Yogachandran Rahulamathavan. Pindroid: A novel android malware detection system using ensemble learning methods. *Computers & Security*, 68:36–46, 2017.
13. Sachin Jain and Yogesh Kumar Meena. Byte level *n*-gram analysis for malware detection. In *Computer Networks and Intelligent Computing*, pages 51–59. Springer, 2011.
14. Kaggle. Welcome to Kaggle competitions. https://www.kaggle.com/competitions, 2018.
15. KDD Cup of fresh air. https://biendata.com/competition/kdd_2018/, 2018.
16. Keras: The Python deep learning API. https://keras.io/.
17. Muhammad Salman Khan, Sana Siddiqui, Robert D McLeod, Ken Ferens, and Witold Kinsner. Fractal based adaptive boosting algorithm for cognitive detection of computer malware. In *15th International Conference on Cognitive Informatics & Cognitive Computing*, ICCI*CC, pages 50–59. IEEE, 2016.
18. Josef Kittler, Mohamad Hatef, Robert P. W. Duin, and Jiri Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, March 1998.
19. Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1357–1365. ACM, 2013.
20. Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley, Hoboken, New Jersey, 2004. https://pdfs.semanticscholar.org/453c/2b407c57d7512fdbe19fa1cefa08dd22614a.pdf.
21. Marios Michailidis. Investigating machine learning methods in recommender systems. Thesis, University College London, 2017.
22. Microsoft malware protection center, winwebsec. https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2fWinwebsec.
23. Symantec security response, zbot. http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99.

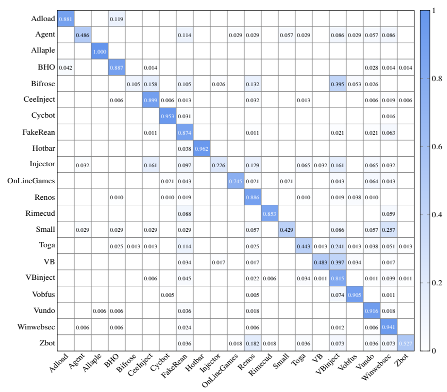
24. Salvador Morales-Ortega, Ponciano Jorge Escamilla-Ambrosio, Abraham Rodriguez-Mota, and Lilian D Coronado-De-Alba. Native malware detection in smartphones with Android OS using static analysis, feature selection and ensemble classifiers. In *11th International Conference on Malicious and Unwanted Software*, MALWARE 2016, pages 1–8. IEEE, 2016.
25. Masoud Narouei, Mansour Ahmadi, Giorgio Giacinto, Hassan Takabi, and Ashkan Sami. Dllminer: structural mining for malware detection. *Security and Communication Networks*, 8(18):3311–3322, 2015.
26. Netflix Prize. https://www.netflixprize.com, 2009.
27. Numpy. https://numpy.org/.
28. Pws:win32/onlinegames. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=PWS%3AWin32%2FOnLineGames.
29. Radare2: Libre and portable reverse engineering framework. https://rada.re/n/.
30. scikit-learn: Machine learning in Python. https://scikit-learn.org/stable/.
31. Raja Khurram Shahzad and Niklas Lavesson. Comparative analysis of voting schemes for ensemble-based malware detection. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 4(1):98–117, 2013.
32. Shina Sheen, R Anitha, and P Sirisha. Malware detection by pruning of parallel ensembles using harmony search. *Pattern Recognition Letters*, 34(14):1679–1686, 2013.
33. Mary Wollstonecraft Shelley. *Frankenstein or The Modern Prometheus*. Dent, 1869.
34. Tanuvir Singh, Fabio Di Troia, Visaggio Aaron Corrado, Thomas H. Austin, and Mark Stamp. Support vector machines and malware detection. *Journal of Computer Virology and Hacking Techniques*, 12(4):203–212, 2016.
35. Vadim Smolyakov. Ensemble learning to improve machine learning results. https://blog.statsbot.co/ensemble-learning-d1dcd548e936, 2017.
36. Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC 2012, pages 239–248. ACM, 2012.
37. Mark Stamp. A revealing introduction to hidden Markov models. https://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf, 2004.
38. Mark Stamp. *Introduction to Machine Learning with Applications in Information Security*. Chapman and Hall/CRC, Boca Raton, 2017.
39. TensorFlow: An end-to-end open source machine learning platform. https://www.tensorflow.org/.
40. Fergus Toolan and Joe Carthy. Phishing detection using classifier ensembles. In *eCrime Researchers Summit, 2009*, eCRIME '09, pages 1–9. IEEE, 2009.
41. Trojandownloader:win32/adload. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader%3AWin32%2FAdload.
42. Trojandownloader:win32/agent. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Agent&ThreatID=14992.
43. Trojandownloader:win32/renos. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Renos&threatId=16054.
44. Trojandownloader:win32/small. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Small&threatId=15508.
45. Trojan:win32/bho. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/BHO&threatId=-2147364778.
46. Trojan:win32/toga. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan:Win32/Toga&threatId=-2147259798.
47. Hendrik Jacob van Veen, Le Nguyen The Dat, and Armando Segnini. Kaggle ensembling guide. https://mlwave.com/kaggle-ensembling-guide/, 2015.
48. Virtool:win32/ceeinject. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool%3AWin32%2FCeeInject.
49. Virtool:win32/injector. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool:Win32/Injector&threatId=-2147401697.

50. Virtool:win32/vbinject. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=VirTool:Win32/VBInject&threatId=-2147367171.
51. Rohit Vobbilisetty, Fabio Di Troia, Richard M. Low, Corrado Aaron Visaggio, and Mark Stamp. Classic cryptanalysis using hidden Markov models. *Cryptologia*, 41(1):1–28, 2017.
52. Win32/allaple. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Allaple&threatId=.
53. Win32/fakerean. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/FakeRean.
54. Win32/rimecud. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Rimecud&threatId=.
55. Win32/vobfus. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Vobfus&threatId=.
56. Win32/vundo. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Vundo&threatId=.
57. XGBoost documentation. https://xgboost.readthedocs.io/en/latest/.
58. Yanfang Ye, Lifei Chen, Dingding Wang, Tao Li, Qingshan Jiang, and Min Zhao. Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. *Journal in Computer Virology*, 5(4):283, 2009.
59. Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 95–104. ACM, 2010.
60. Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. High accuracy android malware detection using ensemble learning. *IET Information Security*, 9(6):313–320, 2015.
61. Boyun Zhang, Jianping Yin, Jingbo Hao, Dingxing Zhang, and Shulin Wang. Malicious codes detection based on ensemble learning. In *International Conference on Autonomic and Trusted Computing*, ATC 2007, pages 468–477. Springer, 2007.
62. Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. CRC Press, Boca Raton, Florida, 2012. http://www2.islab.ntua.gr/attachments/article/86/Ensemble%20methods%20-%20Zhou.pdf.

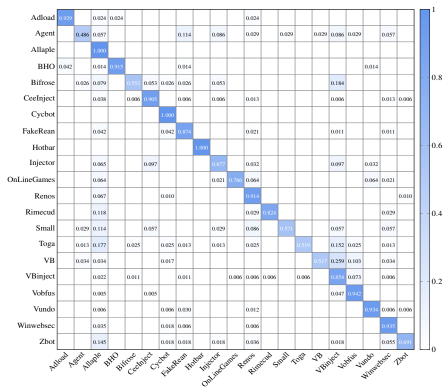
Appendix: Confusion Matrices



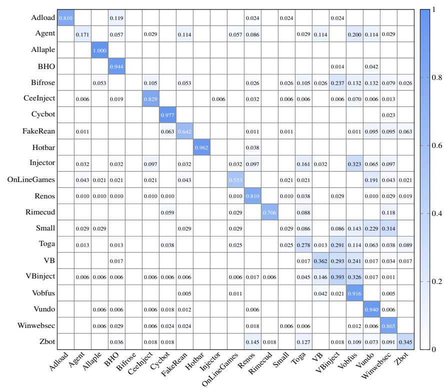
(a) HMM



(b) CNN

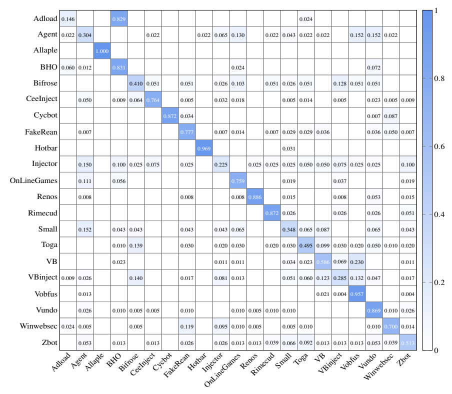


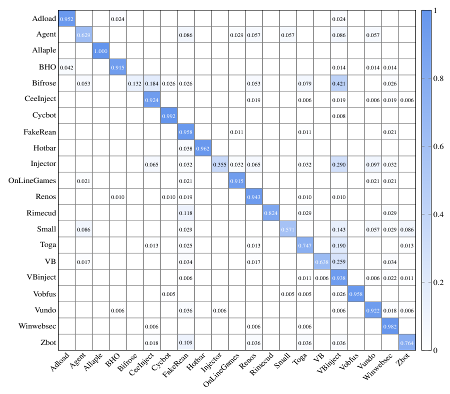
(c) Random Forest



(d) LSTM

Fig. 2: Confusion matrices for standard techniques


(a) Bagged HMM


(b) Bagged CNN

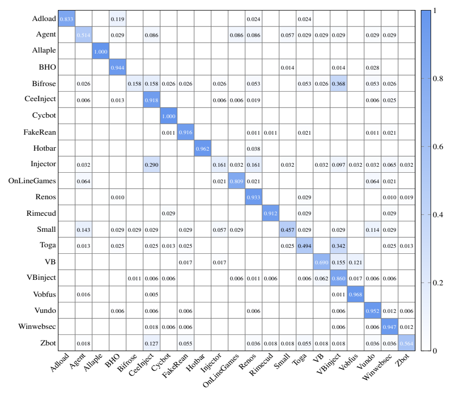
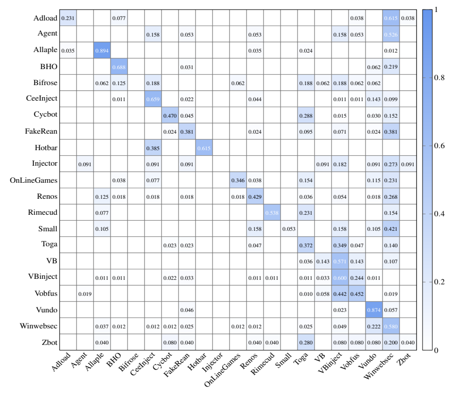

(c) Bagged LSTM

Fig. 3: Confusion matrices for bagging experiments


(a) AdaBoost

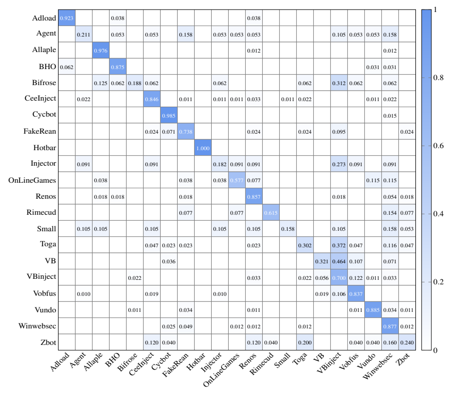
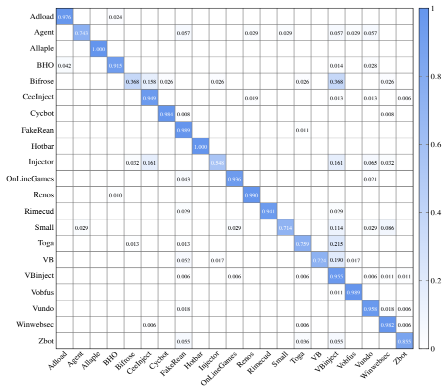
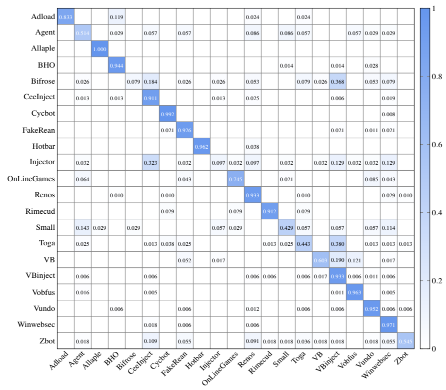

(b) XGBoost

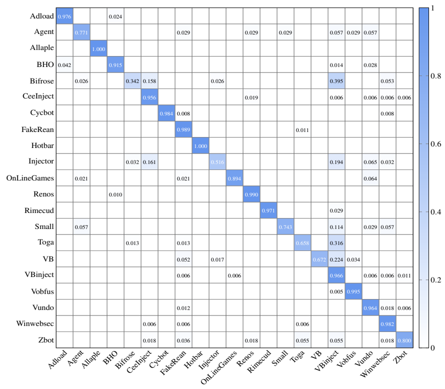
Fig. 4: Confusion matrices for boosting techniques



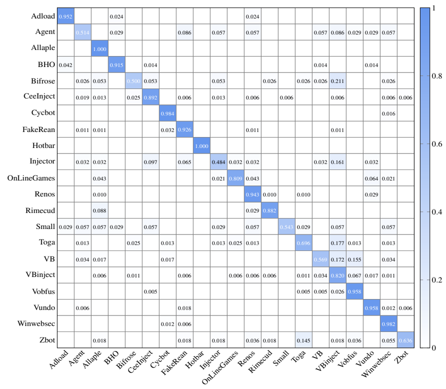
(a) CNN



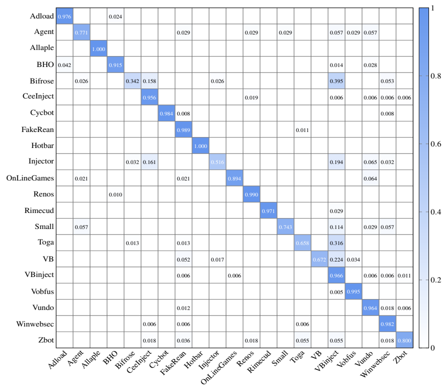
(b) LSTM



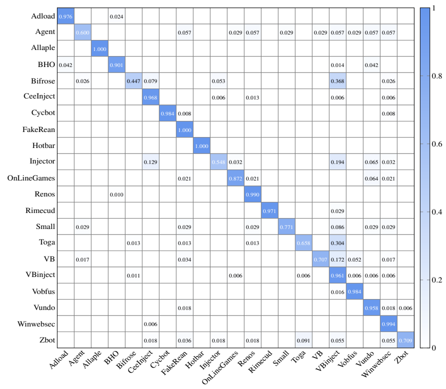
(c) Bagged neural networks



(d) Classic techniques



(e) All neural networks



(f) All models

Fig. 5: Confusion matrices for voting ensembles