A New Adaptive Learning Algorithm and Its Application to Online Malware Detection

Ngoc Anh Huynh[1,2(✉)], Wee Keong Ng[1,2], and Kanishka Ariyapala[1,2]

[1] Nanyang Technological University, Singapore, Singapore
hu0001nh@e.ntu.edu.sg, wkn@pmail.ntu.edu.sg,
kanishka.ariyapala@math.unifi.it
[2] University of Padua, Padua, Italy

Abstract. Nowadays, the number of new malware samples discovered every day is in millions, which undermines the effectiveness of the traditional signature-based approach towards malware detection. To address this problem, machine learning methods have become an attractive and almost imperative solution. In most of the previous work, the application of machine learning to this problem is batch learning. Due to its fixed setting during the learning phase, batch learning often results in low detection accuracy when encountered zero-day samples with obfuscated appearance or unseen behavior. Therefore, in this paper, we propose the FTRL-DP online algorithm to address the problem of malware detection under concept drift when the behavior of malware changes over time. The experimental results show that online learning outperforms batch learning in all settings, either with or without retrainings.

Keywords: Malware detection · Batch learning · Online learning

1 Introduction

VirusTotal.com is an online service which analyzes files and urls for malicious content such as virus, worm and trojan by leveraging on an array of 52 commercial antivirus solutions for the detection of malicious signatures. On record, VirusTotal receives and analyzes nearly 2 million files every day. However, only a fraction of this amount (15%) can be identified as malicious by at least one antivirus solution. Given the fact that it is fairly easy nowadays to obfuscate a malware executable [23], it is rather reasonable to believe that a sheer number of the unknown files are actually obfuscated malware samples. In principle, the rest of the unknown cases should be manually reverse engineered to invent new signatures, but this is infeasible due to the large number of files to be analyzed. Therefore, looking for an automated way to address this problem is imperative and has attracted a lot of research effort, especially in the direction of using machine learning which has gained a lot of successes in various domains of pattern recognition such as face analysis [22] and sentiment analysis [4].

In a recent paper, Saxe et al. [20] train a 3-layer neural network to distinguish between malicious and benign executables. In the first experiment, the

A. Yamamoto et al. (Eds.): DS 2017, LNAI 10558, pp. 18–32, 2017.
DOI: 10.1007/978-3-319-67786-6_2

author randomly splits the whole malware collection into train set and test set. The trained network can achieve a relatively high detection accuracy of 95.2% at 0.1% false positive rate. It is noted that the first experiment disregards the release time of the executables, which is an important dimension due to the adversarial nature of malware detection practice since malware authors are known to regularly change their tactics in order to stay ahead of the game [7]. In the second experiment, the author uses a timestamp to divide the whole malware collection into train set and test set. The results obtained show that the detection accuracy drops to 67.7% at the same false positive rate. We hypothesize that the reasons for this result are two-fold: the change in behavior of malware over time and the poor adaptation of neural network trained under batch mode to the behavioral changes of malware. In addition, another recent study also reports similar findings [3].

The working mechanism of batch learning is the assumption that, the samples are independently and identically drawn from the same distribution (iid assumption). This assumption may be true in domains such as face recognition and sentiment analysis where the underlying concept of interest hardly changes over time. However, in various domains of computer security such as spam detection and malware detection, this assumption may not hold [2] due to the inherently adversarial nature of cyber attackers, who may constantly change their strategy so as to maximize the gains. To address this problem of concept drift, we believe online learning is a more appropriate solution than batch learning. The reason is that, online algorithms are derived from a theoretical framework [11] which does not impose the iid assumption on the data, and hence can work well under concept drift or adversarial settings [5]. Motivated by this knowledge, we propose the Follow-the-Regularized-Leader with Decaying Proximal (FTRL-DP) algorithm – a variant of the proximal Follow-the-Regularized-Leader (FTRL-Proximal) algorithm [12] – to address the problem of malware detection.

To be specific, the contributions of this paper are as follow:

- A new online algorithm (FTRL-DP) to address the problem of concept drift in Windows malware detection. Our main claim is that online learning is superior to batch learning in the task of malware detection. This claim is substantiated in Sect. 7 by analyzing the accuracy as well as the running time of FTRL-DP, FTRL-Proximal and Logistic Regression (LR). The choices of the algorithms are clarified in Sect. 4.
- An extensive data collection of more than 100k malware samples using the state-of-the-art open source malware analyzer, Cuckoo sandbox [6], for the evaluation. The collected data comprises many types such as system calls, file operations, and others, from which we were able to extract 482 features. The experiment setup for data collection and the feature extraction process are described in Sect. 5.

For LR, the samples in one month constitutes the test set and the samples in a number of preceding months are used to form the train set; for FTRL-DP, a sample is used for training right after it is tested. The detailed procedure

is presented in Sect. 6. In Sect. 2, we review the previous works related to the problem of malware detection using machine learning. We formulate the problem of malware detection as a regression problem in Sect. 3. Lastly, in Sect. 8, we conclude the paper and discuss the future work.

2 Related Work

2.1 Batch Learning in Malware Detection

The analysis of suspicious executables for extracting the features used for automated classification can be broadly divided into two types: static analysis and dynamic analysis. In static analysis, the executables are executed and only features extracted directly from the executables are used for classification such as file size, readable strings, binary images, n-gram words, etc. [10,19].

On the other hand, dynamic analysis requires the execution of the executables to collect generated artifacts for feature extraction. Dynamic features can be extracted from host-based artifacts such as system call traces, dropped files and modified registries [17]. Dynamic features can also be extracted from network traffic such as the frequency of TCP packets or UDP packets [16,18]. While static analysis is highly vulnerable to obfuscation attacks, dynamic analysis is more robust to binary obscuration techniques.

To improve detection accuracy, the consideration of a variety of dynamic features of different types is recently gaining attention due to the emergence of highly effective automated malware analysis sandboxes. In their work, Mohaisen et al. [14] studied the classification of malware samples into malware families by leveraging on a wide set of features (network, file system, registry, etc.) provided by the proprietary AutoMal sandbox. In a similar spirit, Korkmaz et al. [9] used the open-source Cuckoo sandbox to obtain a bigger set of features aiming to classify between traditional malware and non-traditional (Advanced Persistent Threat) malware. The general conclusion in these papers is that combining dynamic features of different types tends to improve the detection accuracy.

In light of these considerations, we decided to use the Cuchoo sandbox to execute and extract the behavioral data generated by a collection of more than 100k suspicious executables. Our feature set (Sect. 5.3) is similar to that of Korkmaz et al. although we address a different problem: regressing the risk levels of the executables. Furthermore, our work differs from previous work in the aspect that we additionally approach this problem as an online learning problem rather than just a batch learning one.

2.2 Online Learning in Malware Detection

The importance of malware's release time has been extensively studied in the domain of Android malware detection. In their paper [2], Allix et al. studied the effect of history on the biased results of existing works on the application of machine learning to the problem of malware detection. The author notes that

most existing works evaluate their methodology by randomly picking the samples for the train set and the test set. The conclusion is that, this procedure usually leads to much higher accuracy than the cases when the train set and the test set are historically coherent. The author argues that this result is misleading as it is not useful for a detection approach to be able to identify randomly picked samples but fail to identify zero-day or new ones.

To address this problem of history relevance, Narayanan et al. [15] have developed an online detection system, called DroidOL, which is based on the online Passive Aggressive (PA) algorithm. The novelty of this work is the use of an online algorithm with the ability to adapt to the change in behavior of malware in order to improve detection accuracy. The obtained result shows that, the online PA algorithm results in a much higher accuracy (20%) compared to the typical setting of batch learning and 3% improvement in the settings when the batch model is frequently retrained.

3 Problem Statement

Given 52 antivirus solutions, we address the problem of predicting the percentage of solutions that would flag an executable file as malicious. Formally, this is a regression problem of predicting the output $y \in [0, 1]$ based on the input $x \in R^n$ which is the set of 482 hand-crafted features extracted from the reports provided by the Cuckoo sandbox (Sect. 5.3). The semantic of the output defined in this way can be thought of as the risk level of an executable. We augment the input with a constant feature which always has the value 1 to simulate the effect of a bias. In total, we have 483 features for each malware sample.

In this case, we have framed the problem of malware detection as the regression problem of predicting the risk level of an executable. We rely on the labels provided by all 52 antivirus solutions and do not follow the labels provided by any single one as different antivirus solutions are known to report inconsistent labels [8]. In addition, we also do not use two different thresholds to separate the executables into two classes, malicious and benign, as in [20] since it would discard the hard cases where it is difficult to determine the nature of the executables, which may be of high value in practice.

4 Methodology

To allow a fair comparison between batch learning and online learning, we use the models of the same linear form, represented by a weight vector w, in both cases. The sigmoid function ($\frac{1}{1+e^{-z}}$) is then used to map the dot product (biased by the introduction of a constant feature) between the weight vector and the input, $w^\top x$, to the [0, 1] interval of possible risk levels. Additionally, in both cases, we optimize the same objective function, which is the sum of logistic loss (log loss – the summation term in Eq. 1). In the batch learning setting, the sum of log loss is optimized in a batch manner in which each training example is visited multiple times in minimizing the objective function. The resultant algorithm is

usually referred to as Logistic Regression with log loss (Sect. 4.1). On the other hand, in the online setting, we optimize the sum of log loss in an online manner, in which each training sample is only seen once. The resultant algorithm is the proposed FTRL-DP algorithm (Sect. 4.2).

4.1 Batch Learning – LR

Given a set of n training examples $\{(x_i, y_i)\}_{i=1}^{n}$, Logistic Regression with log loss corresponds to the following optimization problem:

$$\underset{w}{\operatorname{argmin}} \left\{ -\sum_{i=1}^{n} \big(y_i \log(p_i) + (1 - y_i) \log(1 - p_i)\big) + \lambda_1 \|w\|_1 + \frac{1}{2}\lambda_2 \|w\|_2^2 \right\} \quad (1)$$
$$\text{in which } p_i = \text{sigmoid}(w^\top x_i)$$

The objective function of Logistic Regression (Eq. 1) is a convex function with respect to w as it is the sum of three convex terms. The first term is the sum of log losses associated with all training samples (within a time window). The last two terms are the L1–norm regularizer and the L2–norm regularizer. The L1 regularizer is a non–smoothed function used to introduce sparsity into the solution weight w. On the other hand, the L2 regularizer is a smooth function used to favor low variance models that have small weight.

4.2 Online Learning – FTRL-DP

Online Convex Optimization. The general framework of online convex optimization can be formulated as follows [21]. We need to design an algorithm that can make a series of optimal predictions, each at one time step. At time step t, the algorithm makes a prediction, which is a weight vector w_t. A convex loss function $l_t(w)$ is then exposed to the algorithm after the prediction. Finally, the algorithm suffers a loss of $l_t(w_t)$ at the end of time step t (Algorithm 1). The algorithm should be able to learn from the losses in the past so as to make better and better decisions over time.

Algorithm 1. Online Algorithm

```
for t = 1,2,... do
    Make a prediction w_t
    Receive the lost function l_t(w)
    Suffer the lost l_t(w_t)
```

The objective of online convex optimization is to minimize the regret with respect to the best classifier in hindsight (Eq. 2). The meaning of Eq. 2 is that

we would like to minimize the total loss incurred up to time t with respect to the supposed loss incurred by the best possible prediction in hindsight, w^*.

$$\textbf{Regret}_t = \sum_{s=1}^{t} l_s(w_s) - \sum_{s=1}^{t} l_s(w^*) \tag{2}$$

Since the future loss functions are unknown, the best guess or the greedy approach to achieve the objective of minimizing the regret is to use the prediction that incurs the least total loss on all past rounds. This approach is called Follow-the-Leader (FTL), in which the leader is the best prediction that incurs the least total loss with respect to all the past loss functions. In some cases, this simple formulation may result in algorithms with undesirable properties such as rapid change in the prediction [21], which lead to overall high regret. To fix this problem, some regularization function is usually added to regularize the prediction. The second approach is called Follow-the-Regularized-Leader (FTRL), which is formalized in Eq. 3.

$$w_{t+1} = \underset{w}{\operatorname{argmin}} \left\{ \sum_{s=1}^{t} l_s(w) + r(w) \right\} \tag{3}$$

It is notable to see that the FTRL framework is formulated in a rather general sense and performs learning without relying on the iid assumption. This property makes it more suitable to adversarial settings or settings in which the concept drift problem is present.

The Proposed FTRL-DP Algorithm. In the context of FTRL-DP, an online classification or an online regression problem can be cast as an online convex optimization problem as follows. At time t, the algorithm receives input x_t and makes prediction w_t. The true value y_t is then revealed to the algorithm after the prediction. The loss function $l_t(w)$ associated with time t is defined in terms of x_t and y_t (Eq. 4). Finally, the cost incurred at the end of time t is $l_t(w_t)$. The underlying optimization problem of FTRL-DP is shown in Eq. 5.

$$\begin{gathered} l_t(w) = -y_t \log(p) - (1 - y_t) \log(1 - p) \\ \text{in which } p = \text{sigmoid}(w^\top x_t) \end{gathered} \tag{4}$$

Compared with Eq. 3, Eq. 5 has the actual loss function $l_t(w)$ replaced by its linear approximation at w_t, which is $l_t(w_t) + \nabla l_t(w_t)^\top (w - w_t) = g_t^\top w + l_t(w_t) - g_t^\top w_t$ (in which $g_t = \nabla l_t$). The constant term $\left(l_t(w_t) - g_t^\top w_t\right)$ is omitted in the final equation without affecting the optimization problem. This approximation is to allow the derivation of a closed–form solution to the optimization problem at each time step, which is not possible with the original problem in Eq. 3.

$$\begin{gathered} w_{t+1} = \underset{w}{\operatorname{argmin}} \left\{ g_{1:t}^\top w + \lambda_1 \|w\|_1 + \frac{1}{2}\lambda_2 \|w\|_2^2 + \frac{1}{2}\lambda_p \sum_{s=1}^{t} \sigma_{t,s} \|w - w_s\|_2^2 \right\} \\ \text{in which } g_{1:t}^\top = \sum_{i=1}^{t} g_t^\top \end{gathered} \tag{5}$$

FTRL-DP utilizes 3 different regularizers to serve 3 different purposes. The first two regularizers of L1–norm and L2–norm serve the same purpose as in the case of Logistic Regression introduced in Sect. 4.1. The third regularization function is the proximal term used to ensure that the current solution does not deviate too much from past solutions with more influence given to most recent ones by using an exponential decaying function $\left(\sigma_{t,s} = \gamma^{t-s} \text{ with } 1 > \gamma > 0\right)$. This is our main difference from the original FTRL-Proximal algorithm [12]. The replacement of the per coordinate learning rate schedule by the decaying function proves to improve the prediction accuracy in the face of concept drift (discussed in Sect. 7). The solution to the objective function of FTRL-DP is stated in Theorem 1, whose proof is presented in Appendix A.

Theorem 1. *The optimization problem in Eq. 5 can be solved in the following closed form:*

$$w_{t+1,i} = \begin{cases} 0 & \text{if } \|z_{t,i}\|_1 \leq \lambda_1 \\ -\frac{z_{t,i} - \lambda_1 \text{sign}(z_{t,i})}{\lambda_2 + \lambda_p \frac{1-\gamma^t}{1-\gamma}} & \text{otherwise.} \end{cases} \tag{6}$$

$$\text{in which } z_t = g_{1:t} - \lambda_p \sum_{s=1}^{t} \sigma_{t,s} w_s$$

Regret Analysis of FTRL-DP. In Theorem 2, we prove a result that bounds the regret of FTRL-DP. The bound is dependent on the decaying rate γ.

Theorem 2. *Suppose that* $\|w_t\|_2 \leq R$ *and* $\|g_t\|_2 \leq G$. *With* $\lambda_1 = \lambda_2 = 0$ *and* $\lambda_p = 1$, *we have the following regret bound for FTRL-DP:*

$$Regret(w^*) \leq 2R^2 \frac{1}{1-\gamma} + \frac{G^2}{2} \frac{1 + \ln T}{\gamma^T} \tag{7}$$

Due to space constraint, the proof of Theorem 2 will be provided in an extended version of the paper.

In summary, we aim to compare between the performance of batch learning and online learning on the problem of malware detection. To make all things equal, we use the models of the same linear form and optimize the same log loss function, which lead to the LR algorithm in the batch learning case and the FTRL-DP algorithm in the online learning case. For LR, only the samples within a certain time window contribute to the objective function (Eq. 1). On the other hand, the losses associated with all previous samples equally contribute to the objective function of FTRL-DP (Eq. 5). This difference is critical as it leads to the gains in the performance of FTRL-DP over LR, which is discussed in Sect. 7.

5 Data Collection

5.1 Malware Collection

We used more than 1 million files collected in the duration from March 2016 to Apr 2016 by VirusShare.com for the experiments. VirusShare is an online

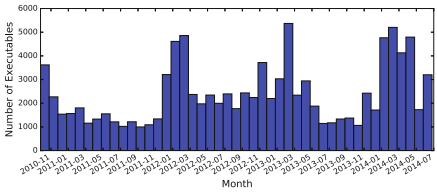

Fig. 1. Distribution of executables

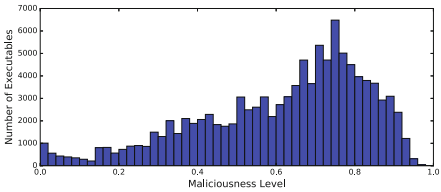

Fig. 2. Distribution of risk level.

malware analyzing service that allows Internet users to scan arbitrary files against an array of 52 antivirus solutions (the scan results are actually provided by VirusTotal). In this study, we are only interested in executable files and able to separate out more than 100k executables from the 1 million files downloaded.

Figure 1 shows the distribution of the executables with respect to executables' compile time. The horizontal axis of Fig. 1 shows the months during the 4 years from Nov/2010 until Jul/2014, which is the period of most concentration of executables and chosen for the study. The vertical axis of Fig. 1 indicates the number of executables compiled during the corresponding month. Figure 2, instead, shows the maliciousness distribution of the executables. The horizontal axis indicates the maliciousness measure and the vertical axis the number of corresponding executables.

5.2 Malware Execution

We make use of the facility provided by DeterLab [13] as the testbed for the execution of the executables. DeterLab is a flexible online experimental lab for computer security, which provides researchers with a host of physical machines to carry out experiments. In our setup, we use 25 physical machines with each physical machine running 5 virtual machines for executing the executables. Each executable is allowed to run for 1 min. The experiment ran for more than 20 days and collected the behavioral data of roughly 100k executables.

5.3 Feature Extraction

In this paper, we mostly consider dynamic features of the following 4 categories for regression: file system category, registry category, system call category, and the category of other miscellaneous features.

API Call Category. API (Application Programming Interface) calls are the functions provided by the operating system to grant application programs the access to basic functionality such as disk read and process control. Although these calls may ease the process of manipulating the resources of the machine, it

also provides hackers with a lot of opportunities to obtain confidential information. For this category, we consider the invoking frequencies of the API calls as a set of features. In addition, we also extract as features the frequencies that the API files are linked. The total number of features in this category is 353 and the complete set of API calls as well as the set of API files are available at https://git.io/vDywd.

Registry Category. In Windows environments, the registry is a hierarchical database that holds the global configuration of operating system. Ordinary programs often use the registry to store information such as program location and program settings. Therefore, the registry system is like a gold mine of information for malicious programs, which may refer to it for information such as the location of the local browsers or the version of the host operating system. Malicious program may also add keys to the registry so as to be able to survive multiple system restarts. We extract the following 4 registry related features: the number of registries being written, opened, read and deleted.

File System Category. File system is the organization of the data that an operating system manages. It includes two basic components: file and directory. File system-related features are an important set of features to consider since malware has to deal with the file system in one way or another in order to cause harm to the system or to steal confidential information. We consider the following file-related features: the number of files being opened, written, in existence, moved, read, deleted, failed and copied. In addition, we also consider the following 3 directory related features: the number of directories being enumerated, created and removed. In total, we were able to extract 11 features in this category.

Miscellaneous Category. In addition to out-of-the-box functionalities, Cuckoo sandbox is further enhanced by a collection of signatures contributed by the public community. These signatures can identify certain characteristics of the analyzed binary such as the execution delay time or the ability to detect virtual environment. All these characteristics are good indicators for the high risk level of an executable but may just be false positives. We consider the binary features of whether the community signatures are triggered or not. In addition, we also consider 3 other features that may be relevant to the behavior characterization: the number of mutex created, the number of processes started and the depth of the process tree. The total number of features in this category is 118.

In summary, we are able to extract 482 features that spans 4 different categories: API calls, registry system, file system and miscellaneous features.

6 Evaluation

6.1 Experiment with LR

We evaluate LR in four different settings: once, multi-once, monthly, and multi-monthly. In the once setting, the samples appeared in the first month of the whole dataset are used to form the train set and the rest of the samples are used to form the test set. The multi-once setting is similar to the once setting except that the samples in the first 6 months are used to form the train set instead. It should be noted that retraining is not involved in the first two settings.

On the other hand, the other two settings do involve retraining, which is a crude mechanism to address the change in behavior of malware over time. Since it is infeasible to carry out retraining upon the arrival of every new sample, we perform retraining on a monthly basis. Due to the characteristic of our dataset, we find that the monthly basis is a good balance to ensure that we have enough samples for the train set and the training time is not too long (the monthly average number of samples is 2.4k). In the monthly setting, we use the samples released in a month to form the test set and the samples released in the immediately preceding month to form the train set. The multi-monthly setting is similar to the monthly setting except that we use the samples in the preceding 6 months to form the train set instead.

For a quick evaluation, we make use of the LR implementation, provided by the TensorFlow library [1] to train and test the LR regressors. TensorFlow is a framework for training large scale neural network, but in our case, we only utilize a single layer network with sigmoid activation, binary cross-entropy loss and two regularizations of L1-norm and L2-norm. 20% of each train set is dedicated for validation and the maximum number of epochs that we use is 100. We stop the training early if the validation does not get improved in 3 consecutive epochs.

6.2 Experiment with FTRL Algorithms

We use the standard procedure to evaluate FTRL-DP and FTRL-Proximal (jointly referred to as FTRL algorithms). Each new sample is tested on the current model giving rise to an error, which is then used to make modification to the current model right after. This evaluation is usually referred to as the mistake-bound model.

Due to their simplicity, FTRL-DP and FTRL-Proximal can be implemented in not more than 40 lines of python code. The implementation makes heavy use of the numpy library, which is mostly written in C++. As TensorFlow also has C++ code under the hood, we believe that the running time comparison between the two cases is sensible. Evaluated on the same computer, it actually turns out that the running time of FTRL algorithms is much lower than that of LR. We use the same amounts of three regularizations for both FTRL-DP and FTRL-Proximal. For FTRL-DP, we report the best possible setting for parameter γ.

The computer used for all the experiments has 16 GB RAM and operates with a 1.2 GHz hexa-core CPU. The running times of all experiments are shown in Table 1. The mean cumulative absolute errors are reported in Fig. 3.

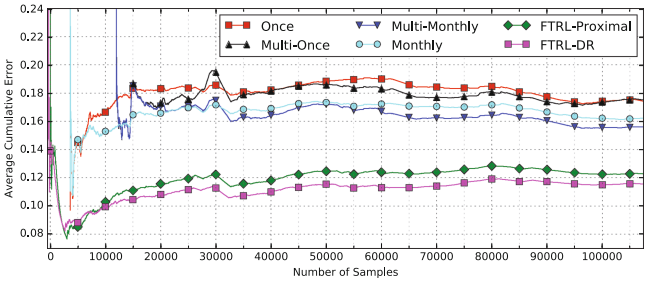

Fig. 3. Mean cumulative absolute errors of FTRL-DP, FTRL-Proximal and different settings of LR.

7 Discussion

7.1 Prediction Accuracy

We use the mean cumulative absolute error (MCAE) to compare the performance between FTRL-DP, FTRL-Proximal and different batch settings of LR, which are reported in Fig. 3. The MCAE is defined in Eq. 8, in which y_t is the actual risk level of an executable and p_t the risk level predicted by the algorithms. In Fig. 3, the horizontal line shows the cumulative number of samples and the vertical line the MCAE. There are four notable observations that we can see from Fig. 3.

$$\frac{1}{n}\sum_{t=1}^{n}|y_t - p_t| \tag{8}$$

Firstly, the more data that we train the LR model on, the better performance we can achieve. This observation is evidenced by the fact that, in most of the time, the error line of the multi-once setting stays below the error line of the once setting, and the error line of the multi-monthly setting stays below the error line of the monthly setting. A possible explanation for this observation is that the further we go back in time to obtain more data to train the model on, the less variance the model becomes, which results in the robustness to noise, and consequently, higher prediction accuracy.

Secondly, the retraining procedure does help to improve prediction accuracy. It is evidenced by the fact that the monthly setting outperforms the once setting, and similarly the multi-monthly setting outperforms the multi-once setting. This observation is a supporting evidence for the phenomenon of evolving malware behavior. As a consequence, the most recent samples would be more relevant to the current samples, and training on most recent samples would result in a more accurate prediction model.

From the first two observations, we can conclude that the further we go back in time to obtain more samples and the more recent the samples are, the better the trained model would perform. This conclusion can be exploited to improve prediction accuracy by going further and further back in time and retraining the

model more often. However, this approach would become unpractical at some point when the training time required to frequently update an accurate model via periodic retrainings would become too long to be practical. It turns out that this issue can be elegantly addressed by the FTRL algorithms, which produces much higher prediction accuracy at considerable lower running time.

Thirdly, FTRL algorithms (worse MCAE of 0.123) are shown to outperform the LR algorithm in all settings (best MCAE of 0.156). The error lines corresponding to the performance of FTRL algorithms consistently stays below other error lines. The gain in the prediction accuracy of FTRL algorithms over all settings of LR can be explained by the contribution of all previous samples to its objective function. In different batch settings of LR, only the losses associated with the samples within a certain time window contribute to the respective objective functions.

Finally, the fourth observation is that FTRL-DP (MCAE of 0.116) outperforms FTRL-Proximal (MCAE of 0.123). The gain in performance of FTRL-DP over FTRL-Proximal can be explained by the ability of FTRL-DP to cope with concept drift via the use of a specially designed adaptive mechanism. This mechanism makes use of an exponential decaying function to favor the most recent solutions over older ones. The effective result is that the most recent samples would contribute more to the current solution thereby alleviating the problem of concept drift.

7.2 Running Time

In terms of running time (training time and testing time combined), FTRL-DP and FTRL-Proximal are clearly advantageous over LR. From Table 1, we can see that the running times of FTRL algorithms are much lower than that of LR, especially compared to the settings with retraining involved (monthly and multi-monthly). The reason for this result is that FTRL algorithms only needs to see each sample once to update the current weight vector whereas in the case of LR, it requires multiple passes over each sample to ensure convergence to the optimal solution.

Table 1. Running time of FTRL-DP, FTRL-Proximal and different LR settings.

Experiment	Running time
LR Multi-monthly	44 m 31 s
LR Monthly	14 m 14 s
LR Multi-once	55 s
LR Once	42 s
FTRL-Proximal	28 s
FTRL-DP	26 s

8 Conclusions and Future Work

The evolving nature of malware over time makes the malware detection problem more difficult. According to previous studies, batch learning based methods often perform poorly when encountered zero-days samples. Our research is motivated to fill in this gap by proposing FTRL-DP – a variant of the FTRL-Proximal algorithm – to address this problem. We evaluated two learning paradigms using an extensive dataset generated by more than 100k malware samples executed on Cuckoo sandbox. The experimental results show that FTRL algorithms (worse MCAE of 0.123) outperforms LR in the typical setting of batch learning as well as the settings with retrainings involved (best MCAE of 0.156). The gain in performance of FTRL algorithms over different batch settings of LR can be accounted for by its objective function taking into account the contribution of all previous samples. Furthermore, the improvement of FTRL-DP over FTRL-Proximal can be explained by the usage of an adaptive mechanism that regularizes the weight by favoring recent samples over older ones. In addition, FTRL algorithms are also more advantageous in terms of running time.

It can be noticed that all above methods are black-box solutions, which do not gain domain experts any insights. An interesting development of this work is to enable the direct interaction with a domain expert using a visualization. The domain expert could prioritize or discard weight alterations suggested by the learning algorithm via the interactive exploration of malware behavior. This visual analytics approach would lead to a transparent solution where the domain expert can benefit most of his knowledge in collaboration with black-box automated detection solutions.

A Proof of Theorem 1

Proof. To remind the optimization objective of FTRL-DP:

$$w_{t+1} = \underset{w}{\operatorname{argmin}}\, g_{1:t}^{\top} w + \lambda_1 \|w\|_1 + \frac{1}{2}\lambda_2 \|w\|_2^2 + \frac{1}{2}\lambda_p \sum_{s=1}^{t} \gamma^{t-s} \|w - w_s\|_2^2$$

$$w_{t+1} = \underset{w}{\operatorname{argmin}} \big(g_{1:t}^{\top} - \lambda_p \sum_{s=1}^{t} \gamma^{t-s} w_s^{\top}\big) w + \lambda_1 \|w\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p \frac{1-\gamma^t}{1-\gamma}\big) \|w\|_2^2 + \frac{1}{2}\lambda_p \sum_{s=1}^{t} \gamma^{t-s} \|w_s\|_2^2$$

Omitting the constant term $\frac{1}{2}\lambda_p \sum_{s=1}^{t} \gamma^{t-s} \|w_s\|_2^2$, we have:

$$w_{t+1} = \underset{w}{\operatorname{argmin}}\, z_t^{\top} w + \lambda_1 \|w\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big) \|w\|_2^2 \tag{9}$$

In Eq. 9, $z_t = g_{1:t}^{\top} - \lambda_p \sum_{s=1}^{t} \gamma^{t-s} w_s^{\top}$ and $r_t = \frac{1-\gamma^t}{1-\gamma}$. Each component of w contribute independently to the objective function of 9 hence can be solve separately:

$$w_{t+1,i} = \underset{w_i}{\operatorname{argmin}}\, z_{t,i} w_i + \lambda_1 \|w_i\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big)\|w_i\|_2^2 \tag{10}$$

Note that w_i in 10 refers to the i^{th} component of w. Let $f(w_i) = z_{t,i} w_i + \lambda_1 \|w_i\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big)\|w_i\|_2^2$. There are two cases:

- If $\|z_{t,i}\|_1 \leq \lambda_1$, we have:

$$f(w_i) \geq -\|z_{t,i} w_i\|_1 + \lambda_1 \|w_i\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big)\|w_i\|_2^2$$

$$f(w_i) \geq -\lambda_1 \|w_i\|_1 + \lambda_1 \|w_i\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big)\|w_i\|_2^2 = \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big)\|w_i\|_2^2 \geq 0$$

  $f(w_i)$ achieves the minimum at $w_i = 0$
- If $\|z_{t,i}\|_1 \geq \lambda_1$, $z_{t,i}$ and w_i must have opposite signs at the minimum of $f(w_i)$ as otherwise w_i can always have sign flipped to further reduce $f_i(w_i)$. Therefore, it is equivalent to solving:

$$w_{t+1,i} = \underset{w_i}{\operatorname{argmin}}\, z_{t,i} w_i - \operatorname{sign}(z_{t,i})\lambda_1 \|w_i\|_1 + \frac{1}{2}\big(\lambda_2 + \lambda_p r_t\big)\|w_i\|_2^2$$

  which achieves minimum at zero gradient or $w_i = -\frac{z_{t,i} - \operatorname{sign}(z_{t,i})\lambda_1}{\lambda_2 + \lambda_p r_t}$
  This concludes the proof.

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., et al.: Tensorflow: large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467 (2016)
2. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Are your training datasets yet relevant? In: Piessens, F., Caballero, J., Bielova, N. (eds.) ESSoS 2015. LNCS, vol. 8978, pp. 51–67. Springer, Cham (2015). doi:10.1007/978-3-319-15618-7_5
3. Bekerman, D., Shapira, B., Rokach, L., Bar, A.: Unknown malware detection using network traffic classification. In: IEEE Conference on Communications and Network Security (CNS), pp. 134–142 (2015)
4. Feldman, R.: Techniques and applications for sentiment analysis. Commun. ACM **4**, 82–89 (2013)
5. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A.: A survey on concept drift adaptation. ACM Comput. Surv. (CSUR) **46**(4), 44 (2014)
6. Guarnieri, C., Schloesser, M., Bremer, J., Tanasi, A.: Cuckoo sandbox-open source automated malware analysis. Black Hat USA (2013)
7. Iliopoulos, D., Adami, C., Szor, P.: Darwin inside the machines: malware evolution and the consequences for computer security. arXiv:1111.2503 [cs, q-bio] (2011)
8. Kantchelian, A., Tschantz, M.C., Afroz, S., Miller, B., Shankar, V., Bachwani, R., Joseph, A.D., Tygar, J.D.: Better malware ground truth: Techniques for weighting anti-virus vendor labels. In: Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security, pp. 45–56. ACM (2015)
9. Korkmaz, Y.: Automated detection and classification of malware used in targeted attacks via machine learning. Ph.D. thesis, Bilkent University (2015)

10. Makandar, A., Patrot, A.: Malware analysis and classification using artificial neural network. In: 2015 International Conference on Trends in Automation, Communications and Computing Technology (I-TACT 2015), vol. 01, pp. 1–6 (2015)
11. McMahan, H.B.: A survey of algorithms and analysis for adaptive online learning. arXiv preprint arXiv:1403.3465 (2014)
12. McMahan, H.B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., Nie, L., Phillips, T., Davydov, E., Golovin, D., et al.: Ad click prediction: a view from the trenches. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1222–1230. ACM (2013)
13. Mirkovic, J., Benzel, T.: Deterlab testbed for cybersecurity research and education. J. Comput. Sci. Coll. **28**(4), 163–163 (2013)
14. Mohaisen, A., Alrawi, O., Mohaisen, M.: AMAL: high-fidelity, behavior-based automated malware analysis and classification. Comput. Secur. **52**, 251–266 (2015)
15. Narayanan, A., Yang, L., Chen, L., Jinliang, L.: Adaptive and scalable android malware detection through online learning. In: 2016 International Joint Conference on Neural Networks (IJCNN), pp. 2484–2491. IEEE (2016)
16. Nari, S., Ghorbani, A.A.: Automated malware classification based on network behavior. In: 2013 International Conference on Computing, Networking and Communications (ICNC), pp. 642–647 (2013)
17. Norouzi, M., Souri, A., Samad Zamini, M.: A data mining classification approach for behavioral malware detection. J. Comput. Netw. Commun. **2016**, 1–9 (2016)
18. Rafique, M.Z., Chen, P., Huygens, C., Joosen, W.: Evolutionary algorithms for classification of malware families through different network behaviors. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, pp. 1167–1174. ACM (2014)
19. Saini, A., Gandotra, E., Bansal, D., Sofat, S.: Classification of PE files using static analysis. In: Proceedings of the 7th International Conference on Security of Information and Networks, p. 429. ACM (2014)
20. Saxe, J., Berlin, K.: Deep neural network based malware detection using two dimensional binary program features. In: 10th International Conference on Malicious and Unwanted Software (MALWARE), pp. 11–20 (2015)
21. Shalev-Shwartz, S.: Online learning and online convex optimization. Found. Trends Mach. Learn. **4**(2), 107–194 (2011)
22. Valenti, R., Sebe, N., Gevers, T., Cohen, I.: Machine learning techniques for face analysis. In: Cord, M., Cunningham, P. (eds.) Machine Learning Techniques for Multimedia, pp. 159–187. Springer, Heidelberg (2008). doi:10.1007/978-3-540-75171-7_7
23. You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA), pp. 297–300 (2010)