SAND2013-6555C

Streaming Malware Classification in the Presence of Concept Drift and Class Imbalance

W. Philip Kegelmeyer and Ken Chiang
Sandia National Laboratories
Livermore, CA 94551-0969, USA
Email: {wpk, kchiang}@sandia.gov

Joe Ingram
Sandia National Laboratories
Albuquerque, NM 87185-0932, USA
Email: jbingra@sandia.gov

***Abstract*—Malware, or malicious software, is capable of performing any action or command that can be expressed in code and is typically used for illicit activities, such as e-mail spamming, corporate espionage, and identity theft.**

Most organizations rely on anti-virus software to identify malware, which typically utilize signatures that can only identify previously-seen malware instances. We consider the detection of malware executables that are downloaded in streaming network data as a supervised machine learning problem. Using malware data collected over multiple years, we characterize the effect of concept drift and class imbalance on batch and streaming decision tree ensembles. In particular, we illustrate a surprising vulnerability generated by precisely the aspect of streaming methods that seemed most likely to help them, when compared to batch methods.

I. Introduction

Malware, or malicious software, has been around since the early 1970's and is capable of performing any action or command that can be expressed in code. It is typically used for illicit activities, such as e-mail spamming, corporate espionage, and identity theft. It has been regarded as one of the most prevalent cybersecurity threats and is increasingly becoming more difficult to detect and avoid [8].

Symantec, a leading software security company, estimates that over 60 percent of websites used to distribute malware in 2012 were legitimate websites that had been compromised. Websites were not the only delivery mechanism for infection; it was also estimated that one in every 291 e-mails contained exploits. Furthermore, targeted attacks, which combine social engineering and malware to target individual employees at specific companies, increased by over 40% from the previous year. These targeted attacks are commonly used for industrial espionage and lead to exfiltration of confidential information, such as customer data [1].

Most organizations rely on anti-virus software to identify malware, which typically utilize signatures. Signature-based detection only allows anti-virus software to detect known malware; they typically cannot generalize to unseen instances, such as zero-day exploits [8]. In addition, as this software is publicly available, malware authors also have access, which allows them to ensure that their exploits circumvent detection.

Signature detection can be seen as an example of crisp rule-based detection, one where the rules are hand-crafted to match specific experience. Such rules are often brittle as data changes over time. Supervised machine learning provides a mechanism for *learning* rules from the data at hand; rules that come to understand what counts as normal as well as the "signatures" of malware. Such rules are typically more robust, partly because they are easy to automatically re-learn as new data arrives.

Supervised machine learning can be regarded as function approximation and estimation. That is, given a set of labeled data instances $\mathcal{L} = \{\langle x, y\rangle^{(l)}\}_{l=1}^{N}$, where $\langle x, y\rangle$ denotes an instance x and its associated label y, the goal is to generate a hypothesis (or model) $h : \mathcal{X} \rightarrow \mathcal{Y}$, which maps the input to its associated output. The resulting model can then be used to label future instances.

II. Data Acquisition and Labeling

We consider the detection of malware executables that are downloaded in streaming network data as a supervised machine learning problem. That is, we will start with a set of executables for which we have

- A label, "goodware" or "malware", and
- A vector of attributes, or features, whose length and organization is constant across all of the executables.

A common issue when employing machine learning in practice is obtaining labeled training data, as well as defining relevant features. We briefly describe our process for collecting and labeling malware and goodware samples and also what features were used in the learning problem.

A. Data Collection

We utilized three batches of data in our experiments, which were collected over the course of three years. Each set consists of about three months of data. Each dataset is characterized in TABLE I.

The goodware samples were collected from live feeds of all files that crossed a corporate network border. However, there could potentially be malware in these feeds. We mitigate this risk by filtering the feed through anti-virus scanners before labeling the data stream as "good". An alternative source of goodware data could have been to use known software included in the Windows operating system, or to download the various versions of browser and PDF reader applications. Using actual goodware executables as they arrive at a corporate network, however, allows us a more accurate, temporally-nuanced perspective of the data we might expect in the future.

TABLE I. DATA SOURCES AND MAGNITUDE

ID	Dates	Goodware count	Malware count
2010	10-2010 to 01-2011	10260	8501
2011	11-2011 to 02-2012	3409	13011
2012	01-2012 to 03-2012	54153	16911

For malware, we used a daily feed from Arbor Networks[1], a security company that collects malicious software from the many network sensors that they own.

B. Feature Extraction

The features used for our supervised machine learning problem are based on Portable Executable (PE) headers, which specify the layout of executable files for the Windows operating system. The PE header resides at the beginning of a Windows executable file and is marked by the signature bytes 'MZ'. As examples, PE headers indicate where the code sections are on file and where they should be mapped in memory during execution, what imported dynamically linked libraries (DLLs) are loaded and which of their library functions to use in the software, checksums, etc. For a good introduction to the PE format, see [11]. We also compute a few additional features based on group properties of the PE headers, such as the entropy of the different sections in the file.

A Python script based on the `pefile`[2] library was used to extract header features from each file. Since the executables that we collected were all Windows-formatted files, the features we extracted are all based on parsing and characterizing the PE headers. Therefore, collecting these features does not require a complete and non-corrupted file, but only a complete and non-corrupted header.

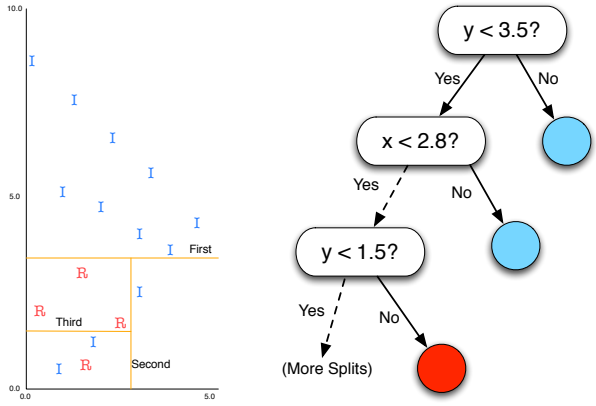
III. SUPERVISED LEARNING METHODOLOGY

The current state of the art method for practical, robust, accurate supervised machine learning, in a *fixed* data set, is ensembles of decision trees [2]. The malware detection problem, however, presents a *streaming* data problem, in that data is constantly arriving.

If we take "streaming data" to mean that we are constrained to examine each executable and its vector of attributes only once, we note that traditional ensembles are ill-suited for streaming data, for three reasons:

- The standard method of building an individual decision tree requires repeatedly revisiting each data point's attribute vector.
- The standard method of building an ensemble requires repeatedly resampling from the training data to make a specialized training set for each tree.
- The "streaming" nature of the arrival of executables is not solely a processor or storage resource concern. The executables arrive over time, and so their nature can change over time, as malware methods change, and fall into or out of favor. A static ensemble of trees that does not adjust to the temporal nature of the data, to the "concept drift" inherent in the data, will be useful, at best, only shortly after the ensemble is trained.

[1]http://www.arbornetworks.com/
[2]http://code.google.com/p/pefile/



(a) Feature space partitioned (b) Decision tree representation

Fig. 1. A 2D Feature Space and its Decision Tree Partitioning

A. Batch Trees

To understand how a batch decision tree is built (and ultimately how it is inappropriate for streaming data), take the geometric view of imagining each data point as existing in an N-dimensional space, where N is the number of attributes. To determine the optimal partitioning of the data, the tree induction algorithm defines an objective function (generally called a "purity" function) that measures how much better the classes are separated in the children of any proposed split. Then it considers *all* possible splits, and picks the one which maximizes the objective function.

Fig. 1(a) illustrates a very simple case, with only two features (x and y), 16 data points, and two classes, Blue I and Red R. What the decision tree algorithm does is figure the best way to incrementally partition the feature space, giving each partition a unique label. Then to figure out the label of a new point, you figure out which partition it fell into, and look up its label. Or, equivalently, you march down the decision tree representation of the partition (as in Fig. 1(b)) and look up the label associated with the leaf node of the tree.

B. Batch Ensembles

The previous section describes the process for building a single batch decision tree. There are many methods for building *ensembles* of classifiers. They can all be illustrated well enough by considering the "bagging" method.

The core idea behind ensembles via bagging is to generate many variants of a data set from an original baseline data set. Each variant is used to train a single machine learning model, and since each model is based on slightly varying data, the models will vary as well. Fig. 2 illustrates the core idea for a toy example which has only 10 training data points, labeled 0–9. Each of the four bagging examples depict picking randomly from those ten data points, *with replacement*, until ten points have been picked. Since the selection is done with replacement, any one data point might be picked more than once. Also, since a data point can be picked more than once, but the resultant

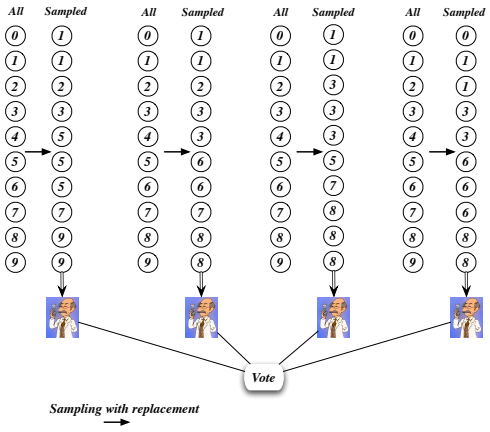


Fig. 2. Bagging Seems to Require Revisiting Data

data set is the same size, some points will be omitted. The result is a skewed, variant version of the original data. As an example, in Fig. 2, the first bag has selected multiple copies of data points 5 and 9, but has omitted data points 4 and 6.

The point of ensemble methods is that simple majority voting of the resulting variant machine learning models is, nearly inevitably, more accurate than consulting a single model learned from the unmodified data [9].

C. Streaming Trees

As was mentioned in Section III-A, the standard decision tree algorithm must constantly revisit every data point, as each data point must be sorted along every attribute dimension and every possible threshold investigated in order to determine the *best* split. This process is obviously incompatible with the streaming constraint of examining each data point only once.

However, if we relax the requirement that we are required to find the *best* split, a streaming approach becomes possible. The basic idea is to recognize that, quite frequently, only a small subset of the data may be needed to find the best attribute at a given node. So the initial core of a streaming tree algorithm is to:

- Maintain a list of leaves in the current tree.
- Filter an example from the stream into the appropriate leaf.
- Extract statistics (its class and its contribution to histograms on each attribute) from the example, and discard.
- Spilt a leaf into two new children *only* when it contains enough examples to "reliably" pick the "best" attribute for splitting.

To reliably pick the best attribute, we make use of the Hoeffding bound [6]. Let r be a real-valued random variable with range R. After n observations the computed mean is $\overline{r}$. The Hoeffding bound says that, with probability $1-\delta$, the true mean of r is at least $\overline{r} - \epsilon$, where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

This holds *regardless* of the probability distribution of r.

To make use of this with decision trees, let $\overline{G}(A)$ be the quality (the overall increase in purity) that comes from splitting a leaf using the best threshold on attribute A. Suppose that A_1 and A_2 are the first and second best attributes with respect to $\overline{G}(\cdot)$, so far. Then, define

$$\Delta\overline{G} = \overline{G}(A_1) - \overline{G}(A_2) \geq 0.$$

$\Delta\overline{G}$ is thus how much better attribute A_1 is than its closest competitor, A_2, at splitting the data in the current leaf. It makes some intuitive sense that if this delta is large enough, we could go ahead and split on attribute A_1 and be fairly confident that this would be the split chosen if we waited to see all the data and did the full exhaustive search.

And in fact, letting $\Delta\overline{G}$ be the random variable r mentioned above, the Hoeffding bound lets us assert that if $\Delta\overline{G} > \epsilon$, then A_1 is the best attribute to split on, with probability $1-\delta$.

To make a tree growing procedure out of this:

1) Pick δ, such as 0.05. (Remember: ϵ is a function of δ and n.) When δ is smaller, it is more likely that the Hoeffding tree will match the batch tree, but it is also the case that much more data will be needed before the node can split.
2) Accumulate samples at each node (that is, increase n for that node) until the best split is ϵ better than the second best split.
3) Then make that best split, set the initial n of each child leaf to the number of points that fall into that leaf, and recurse on the children.
4) (Handling continuous data is a tricky special case, but one which has been addressed [7].)

If, for instance, $\delta = 0.05$, the result is a decision tree where each node is 95% likely to have the same threshold and attribute that would have been selected at that node by a batch tree growing algorithm.

This procedure has a number of nice properties in the context of streaming data:

- Each example is examined only once.
- Each leaf requires a fixed and known amount of memory.
- The resulting tree is asymptotically arbitrarily close to the tree produced by a batch learner.

"Asymptotically arbitrarily close" means, of course, that the Hoeffding trees are not identical to the batch trees that would have been generated from the same data. But as we plan to use these Hoeffding trees in ensembles of deliberately variant trees, the small deltas turn out to not matter much.

D. Streaming Ensembles

Adapting bagging to the streaming context turns out to be much simpler than adapting tree induction. The first publication on the topic was from Nikuj Oza [10], and so the method is popularly called "Oza bagging".

The key insight is that the bagging process (sampling with replacement from a data set of size N to make a another data

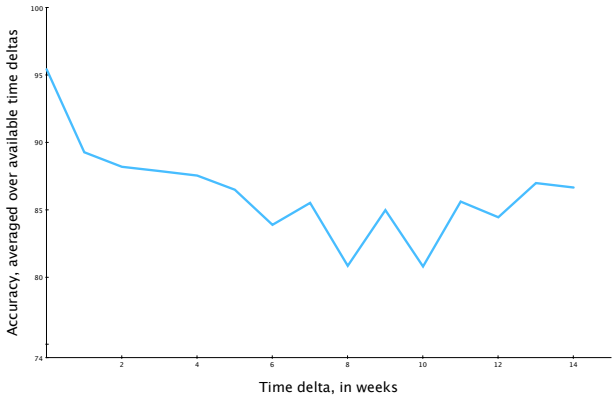


Fig. 3. Malware Detection Accuracy Degrades over Time

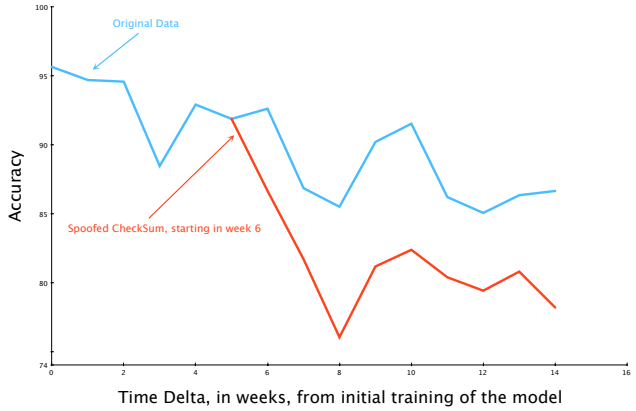


Fig. 4. Result of a Suddenly Invalidated Malware Detector

set of size N), means that each bag has K copies of a sample, where K turns out to be binomial:

$$P(K=k) \quad = \quad \binom{N}{k}\left(\frac{1}{N}\right)^{k}\left(1-\frac{1}{N}\right)^{N-k}$$

The binomial distribution has the useful properly that, as N gets large,

$$\text{As } N\to\infty, P(K=k) \quad \approx \quad \frac{e^{-1}}{k!}$$

That is, K tends towards a random variable with a $Poisson(1)$ distribution. Furthermore, this is a very robust approximation, one that is nearly perfect for N as small as thirty.

So to build a set of E Hoeffding trees while still only handling each data point only once, consider each sample x_i. For each individual Hoeffding tree, t_j, pick K_{ij} from a $Poisson(1)$ distribution and give K_{ij} copies of x_i to t_j. The tree will use those copies (if $K_{ij} > 0$ for this sample) to update the tree, and then discard them. Once every tree has been updated, the sample is discarded for the entire ensemble, and the next sample is considered.

The result is that Oza bagging achieves a data distribution which converges to that of standard bagging, yet is perfectly compatible with considering each data point only once, as is required by the streaming context.

IV. Investigation of Concept Drift

In real-world applications, the underlying concepts that machine learning algorithms try to learn are often not static but change over time. That is, with concept drift, the underlying data distribution changes with time, which typically results in a decrease in performance of a static model. There are typically two main types of drift, gradual and sudden [12].

A. Gradual Concept Drift

An early version of the data collection described in Section II resulted in fifteen weeks of malware and goodware data, all time stamped at day resolution. Each week of data was aggregated into a single dataset, yielding twelve temporally arranged datasets, where dataset ten was, for instance, collected nine weeks after dataset one.

A batch ensemble was built on *every* week and tested on every week that followed. Thus there were fourteen tests of a one week gap between training and testing, thirteen tests of a two week gap, and one test of a fourteen week gap. Further, every model could be tested on its own training data in a fair way using 10-fold cross-validation, resulting in fifteen tests of no gap.

The results are in Fig. 3, which illustrates that the malware detection accuracy started at about 95% but degraded to about 85% after 12 weeks. Further, the degradation is not smooth. In this case this could be because each subsequent data point is being averaged over fewer experiments and so should have higher variance, but in general it should not be expected that degradation will be smooth, as performance could depend on the day of the week, on the re-discovery of some specific exploit, or on the sudden patch of an exploit.

B. Sudden Concept Drift

One of the benefits of working with ensembles of decision trees is that they permit easy assessment of the importance of each of the features. When we investigated the results above, and the executable attributes were quantitatively assessed via the "Path" metric [4], we realized that the PE header "CheckSum" was the single strongest indicator of malware. For general applications CheckSum is an optional field, and so it might be zero in goodware. However, it is required of all system executables; all executables in the Windows 7 base install (the `system32` directory) have a non-zero value. The reason it seems to be useful here may be that malware, in our data, was observed to be much more likely to have a zero CheckSum. Week 1 is typical: 12% of the goodware had a CheckSum of zero, but 43% of the malware had a CheckSum of zero.

So perhaps the model has learned that malware developers were lazy or ill-informed about CheckSum's significance. To illustrate what that could mean, if true, Fig. 4 illustrates what would happen if malware developers suddenly caught on and were careful about setting CheckSum. It should be noted that the ability to set this value is easily available to malware authors as the Windows SDK provides an API for legitimate software to set it, and as far back as 1999, the W32/Kriz virus computed the checksum of files that it infected to mask its tampering.

TABLE II. BASELINE PERFORMANCE WITHOUT CONCEPT DRIFT

Data	Batch Ensemble	Streaming Ensemble
2010	98.80	96.77
2011	98.45	96.51
2012	98.60	96.73

The blue curve is the actual result of building a model on week 0 and testing on weeks 1 through 14. (This is not *averaged* delta performance, as in Fig. 3; this is the result of working with a single model.) The red line is a simulation of what would happen if, in week 6, malware writers started inserting non-zero checksum values into the PE header. We generated this simulated data by looking at the non-zero goodware checksums for a given week and inserting one, at random, to replace every malware checksum that had a value of zero.

The point of this figure, of the roughly 10% loss of accuracy, is to illustrate that we need to be vigilant toward sudden concept change as well as concept drift, particularly in the context of adversarial machine learning.

V. EXPERIMENTS AND RESULTS

As discussed in Section II, we are working with three data sets, the "2010", "2011", and the "2012" data. Each set represents three months of executables, each classified as goodware or malware.

For the implementation of batch decision tree ensembles, we used a custom C implementation called "Avatar Tools". For the implementation of Oza Bagging with incremental Hoeffding trees, we modified Massive Online Analysis (MOA), an open-source Java library developed at the University of Waikato [3].

A. Baseline Performance with No Drift

The whole point of having acquired temporally separated data sets is to explore the impact of concept drift, but in order to assess that impact, we must start from an understanding of the baseline performance in the case where concept drift is not an issue.

So, the baseline accuracies in TABLE II indicate the result of 10-fold cross-validated bagged ensemble analysis of each year individually, where the ensemble size is automatically selected for both batch and streaming ensembles, and Oza bagging is used with Hoeffding trees.

One conclusion is that both batch and streaming decision tree ensembles are pretty good at distinguishing between malware and goodware, although the streaming ensemble is somewhat less accurate than its batch counterpart. This result makes intuitive sense as the decision trees constructed in a batch setting do not have to start making commitments to their structure until they have seen *all* of their training data.

Hoeffding trees, on the other hand, begin to build the upper levels of their trees after they have seen only a fraction of the data. On one hand, this means they are always ready to classify an unlabeled sample, no matter how little data they have seen; batch trees cannot do this. On the other hand, since Hoeffding trees have to commit early, each individual tree will be less accurate than the average batch tree, and so it makes sense that the ensemble might be less accurate as well. Next we compare the baseline performance in TABLE II with a variety of concept drift experiments.

TABLE III. ACCURACY WITH CONCEPT DRIFT AND COPIUS MALWARE

Train	Test	Type	Batch Ensemble	Streaming Ensemble
2010	2011	Standard	90.97	91.87
2010	2011	Interleaved	—	95.69
2010	2012	Standard	90.03	85.80
2010	2012	Interleaved	—	92.82
2011	2012	Standard	91.25	81.36
2011	2012	Interleaved	—	93.32

B. Performance with Concept Drift and Copius Malware

As discussed in Section II, we have daily access, via Arbor Networks, to *much* more malware data than we could collect ourselves. We first investigate the impact of concept drift in the context where *all* of that additional malware data is available for both training and testing.

See TABLE III for the results. To interpret this table, note:

Train/Test:
: All of the experiments involve building a model on the training year's data, and testing on a subsequent year.

Standard/Interleaved:
: There are two possible ways to evaluate the performance of the test data. In the "standard" mode, once the ensemble model is built on the training year it remains fixed and unchanged, and the test year data is evaluated against that model all at once.
: In the "interleaved" mode, the test data is presented to the model one sample at a time. The sample is tested, the accuracy on the sample is recorded, and *then* the same sample is used to update the model.
: Note that the "interleaved" process is available only to the streaming methods, as they have the ability to be incrementally updated. The decision trees trained in a batch setting do not share this property; they cannot be incrementally updated: at best you would have to start over and rebuild the tree from scratch on the updated data.

Batch Ensemble:
: This indicates using an automatically sized bagged ensemble of batch decision trees as the model.

Streaming Ensemble:
: This indicates using one hundred Hoeffding trees with Oza bagging as the model.

The main conclusion suggested by TABLE III is that the interleaved streaming model always out-performs the batch ensemble, which, remember, was not true when there was no concept drift. This is very satisfying, as it means that no accuracy is lost even though we are using a streaming-compatible machine learning algorithm. However, it also makes sense, as interleaving means the model is able to make use of new data

TABLE IV. CLASS AVERAGED ACCURACY WITH CONCEPT DRIFT AND CLASS IMBALANCE

Train	Test	Type	Batch Ensemble	Streaming Ensemble
2010	2011	Standard	91.96	90.63
2010	2011	Interleaved	—	90.70
2010	2012	Standard	87.46	86.77
2010	2012	Interleaved	—	82.09
2011	2012	Standard	93.63	84.38
2011	2012	Interleaved	—	75.60

not available to the batch model. One might thus conclude that one should always add in new data as it arrives. The next section issues a caution around that conclusion.

C. Performance with Concept Drift and Class Imbalance

The previous section investigated the context where malware continues to populate the test data at roughly the same rate as the training data. The incidence rate of malware actually arriving in most organizations would be much lower. Quantifying that arrival rate is complex, but one measure would be to note that the intrusion detection specialists at our corporate partner spot roughly two malware samples per day that they consider worthy of investigation.

So we conducted another round of experiments in which we trained on the full set of malware and goodware, but thinned the test malware to only two samples per day, or 180 samples per test set. In order to ensure stable results, given that not all of the malware is tested each time, the tests were repeated ten times, and the average performance was reported.

Further, for this experiment we report *class averaged accuracy*. That is, we compute accuracy when testing on goodware only, and accuracy when testing on malware only, and then average them together. This is a more sensitive test of accuracy when one of the test classes is rare. For instance, if there are 180 malware samples in the test set, but 5000 goodware samples, then a machine learning model that calls everything goodware would have a standard accuracy of 96.4%, but a class averaged accuracy of 50% (100% accurate on goodware, 0% accurate on malware), which is a more useful indication of how well the classifier discriminates between the classes.

TABLE IV indicates the results of the experiments, using the same notation as in TABLE III. Note that in the presence of both concept drift *and* class imbalance, the interleaved streaming model did *worse* than batch analysis (and generally worse than non-interleaved streaming analysis), despite the fact that interleaved analysis provides fresh data input.

It turns out this fresh data input is exactly the source of the degradation. It is not caused by the samples themselves, but by their *proportion*. There are so few malware samples in the new data that the interleaved model quickly learns it can do best by predicting that nothing is malware.

VI. CONCLUSION AND FUTURE WORK

We have characterized the effect of concept drift and class imbalance on batch and streaming decision tree ensembles using a malware dataset collected from live feeds. We have demonstrated how bagged ensembles of decisions trees can be well-adapted to the streaming data case, and illustrated a perhaps surprising vulnerability stemming from updating a model based on new data.

Future work would be to investigate the efficacy of algorithms specifically designed to handle concept drift when coupled with class imbalance. A first step will be to integrate the SMOTE [5] algorithm for generating data-conditional synthetic training samples of the minority class.

ACKNOWLEDGMENT

Funding for this work came from the Laboratory Directed Research and Development program at Sandia National Laboratories.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] "Internet security threat report 2013," Symantec, Tech. Rep., 2013. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf

[2] R. E. Banfield, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer, "A comparison of decision tree ensemble creation techniques," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 1, pp. 173–180, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2007.2

[3] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive online analysis," *J. Mach. Learn. Res.*, vol. 99, pp. 1601–1604, August 2010, software at: http://moa.cs.waikato.ac.nz. [Online]. Available: http://dl.acm.org/citation.cfm?id=1859890.1859903

[4] R. Caruana, M. Elhawary, A. Munson, M. Riedewald, and D. Sorokina, "Mining citizen science data to predict prevalence of wild bird species," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2006.

[5] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002. [Online]. Available: http://adsabs.harvard.edu/abs/2011arXiv1106.1813B

[6] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '00. New York, NY, USA: ACM, 2000, pp. 71–80. [Online]. Available: http://doi.acm.org/10.1145/347090.347107

[7] J. Gama, R. Fernandes, and R. Rocha, "Decision trees for mining data streams," *Intell. Data Anal.*, vol. 10, pp. 23–45, January 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1239076.1239079

[8] I. Kirillov, D. Beck, P. Chase, and R. Martin, "Malware attribute enumeration and characterization," MITRE Corporation, Tech. Rep., 2010. [Online]. Available: http://maec.mitre.org/about/docs/Introduction_to_MAEC_white_paper.pdf

[9] L. Kuncheva, *Combining pattern classifiers: methods and algorithms*. Wiley-Interscience, 2004.

[10] N. Oza, "Online bagging and boosting," in *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, vol. 3, oct. 2005, pp. 2340 – 2345 Vol. 3.

[11] M. Pietrek, "An in-depth look into the Win32 portable executable file format," *MSDN Magazine*, 2002. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

[12] A. Tsymbal, "The problem of concept drift: Definitions and related work," Trinity College, Tech. Rep., 2004.