POSTER: Enabling Fair ML Evaluations for Security

Feargus Pendlebury[*]
Royal Holloway, University of London
King's College London

Fabio Pierazzi[*]
Royal Holloway, University of London
King's College London

Roberto Jordaney
Royal Holloway, University of London
King's College London

Johannes Kinder
Royal Holloway, University of London

Lorenzo Cavallaro
King's College London

ABSTRACT

Machine learning is widely used in security research to classify malicious activity, ranging from malware to malicious URLs and network traffic. However, published performance numbers often seem to leave little room for improvement and, due to a wide range of datasets and configurations, cannot be used to directly compare alternative approaches; moreover, most evaluations have been found to suffer from experimental bias which positively inflates results. In this manuscript we discuss the implementation of Tesseract, an open-source tool to evaluate the performance of machine learning classifiers in a security setting mimicking a deployment with typical data feeds over an extended period of time. In particular, Tesseract allows for a fair comparison of different classifiers in a realistic scenario, without disadvantaging any given classifier. Tesseract is available as open-source to provide the academic community with a way to report sound and comparable performance results, but also to help practitioners decide which system to deploy under specific budget constraints.

KEYWORDS

Evaluation; Malware; Machine Learning; Experimental Bias

1 INTRODUCTION

Machine learning (ML) is now a widespread approach in security literature to tackle *malware detection*: from more traditional malware such as Android, Windows, and PDF, to other malicious activities such as botnet traffic, DGA domains, malicious URLs and harmful Javascript. Given high performance with F_1 of up to 95-99%, achieved especially on Windows [6, 10] and Android [3, 9, 15] platforms, it may seem that using machine learning for malware detection is a solved problem. However, recent studies [1, 11, 12] have shown that most evaluations suffer from experimental bias that positively inflates results. Although security researchers borrowed best practices from the machine learning community, these may not be appropriate for a security setting. For example, *k-fold cross validation* highly inflates results in a malware context, because it integrates future knowledge about not-yet-seen malware into training [1, 11, 12]; when the training set is chosen with objects that are temporally precedent to the testing objects, the classifier performance degrades significantly due to concept drift [8]. In a recent study [12], we have identified and systematized sources of *experimental bias* that affect even recent top-tier papers (e.g., [3], [9]): *temporal bias* (caused by violating temporal consistency of train and test objects) and *spatial bias* (caused by using unrealistic ratios of malware-to-goodware in the test set).

In this manuscript, we discuss the implementation of Tesseract [12], an open-source tool that we have made available to the community to ensure fair, sound and comparable evaluations of machine learning classifiers. The theoretical findings on which Tesseract is based are described in [12]. Tesseract supports any Scikit-learn and Keras classifier and feature space, and outputs plots and metrics that consider performance decay over time due to concept drift. This manuscript illustrates how to properly use Tesseract, and clarifies how we engineered it to obtain a simple, flexible tool that could be easily reused by other researchers.

We remark that we use *(experimental) bias* to refer to the details of an experimental setting that depart from the conditions in a real-world deployment and can have a positive impact (*bias*) on performance. We do not intend it to relate to the classifier bias/variance trade-off [5] from traditional machine learning terminology.

We encourage the adoption of Tesseract [12] as a way to perform fair, unbiased, and comparable experiments of ML classifiers in security contexts to promote their evaluation in realistic settings.

2 BACKGROUND

Researchers have started to focus on understanding how to achieve fair ML evaluations both in the security community [1, 11–13, 17] and in the ML community [7, 16]. The first experimental bias found in security is probably associated with the *base-rate fallacy* in intrusion detection [4]: in the presence of highly imbalanced datasets (e.g., most traffic is benign) ROC curves, TPR and FPR have been misleading metrics to evaluate system performance; this is because, for an FPR of 0.1%, there may still be millions of false positives with only thousands of true positives. Moreover, in imbalanced datasets *Accuracy* is also a very misleading metric, which is discouraged to report alone [7]. Some studies [13, 17] discuss incorrect experimental setups and challenges in fair security evaluations, but do not propose practical solutions on how to remove it. Other work [1, 11] evidences that *temporal consistency* is crucial when evaluating malware classifiers: the samples in the training set must be temporally precedent to those in the testing set. In [12], we have identified stricter temporal constraints and also that the testing goodware-to-malware ratio must reflect the real world distribution, or otherwise the results are misleading and possibly inflated.

[*]Equal contribution.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

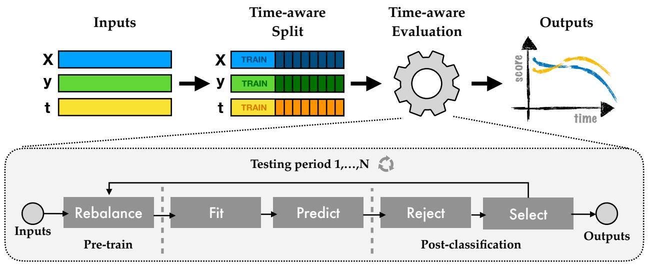


Figure 1: Tesseract workflow and the evaluation cycle.

The aim of the Tesseract tool is to aid researchers in executing bias-free evaluations of ML classifiers for security. While we present the theoretical implications of such bias and constraints and how to remove it in a full-length work [12], this manuscript focuses on the detailed description of the implementation of the open-source tool that we have released to the community.

3 SYSTEM OVERVIEW

Tesseract is implemented as a Python library, designed to integrate easily with common workflows. In particular, the API design of Tesseract is heavily inspired by and fully compatible with the popular machine learning libraries Scikit-learn and Keras. As a result, many of conventions and concepts in Tesseract should be familiar to users of those libraries.

The goal of Tesseract is to ensure a fair, time-aware evaluation of security classifiers. To achieve this, Tesseract will enforce proper temporal and spatial constraints [12] to prevent results becoming affected by experimental bias. As classifiers grow in complexity by combining multiple machine learning techniques, it becomes increasingly likely that these constraints will be violated. Tesseract aims to reduce the burden on the algorithm designer by keeping track of these properties at each stage of the experiment pipeline.

Tesseract divides the workflow into stages (Figure 1). Firstly, the dataset is ordered chronologically and divided into a single training set and multiple testing sets. Next, execution enters the *time-aware evaluation cycle* where each iteration of the cycle processes the subsequent set of test objects. The evaluation cycle is composed of multiple stages centered around the standard "training" and "prediction" procedures. The stage preceding training enables adjustments to be made to the training set while the later stages allow for policies for reacting to the results before the cycle repeats. Finally, once all test objects have been processed, the complete results are consolidated and presented to the user.

Tesseract is composed in a modular fashion, to reflect the different stages of the evaluation cycle. Different phases of the cycle are represented by subclasses of `Stage`, which can themselves be subclassed to implement specific learning strategies. Instances of these subclasses can then be injected into the function `fit_predict_update()` which will activate them appropriately throughout the evaluation or deactivate them according to a given `schedule` (a boolean array) attached to the superclass. Alternatively, any component from the framework can be appropriately selected and used in conjunction with other libraries or methodologies.

The following paragraphs highlight details of the implementation and further explore the core stages of Tesseract's workflow.

Temporal Awareness. While a single training or testing object is typically represented as a set of features X and an output variable, or ground-truth, y, Tesseract also expects a timestamp t. This allows Tesseract to enforce temporal consistency when partitioning the dataset; e.g., for training, validation or testing. Tesseract partitions testing sets further into testing *periods*. Each period contains test objects covering a particular timespan specified in days, weeks, months, quarters or years. All test periods are processed in chronological order and all are temporally posterior to the training set.

Time-aware operations are implemented in `temporal.py` which handles the various corner cases and complications that occur when working with time deltas. A notable function from the module is `time_aware_train_test_split()` that performs the dataset partitioning mentioned previously given a time period length, granularity and an optional start date.

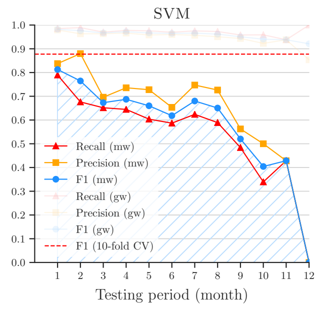
Pre-train Stages. Before training the classifier it can be beneficial to make adjustments to the training set. For example, altering the class balance of the training set can be used to tune a classifier in order to make it more or less receptive to a particular class [12]. This is especially useful in many security applications where the class of interest—often malicious—is also the minority class.

The module `spatial.py` contains `downsample_set()` to reduce the majority class until the desired class balance is achieved, as well as `search_optimal_train_ratio()` to estimate the optimal training balance for a particular target metric (e.g., F_1). Custom methods for these adjustments can be injected by subclassing the `Rebalancer` class and overriding its `alter()` method which will then be invoked before training on each cycle iteration. The function `downsample_set()` can also be used to ensure the class balance of each testing period is *realistic*, as over-representing the class of interest at test time, with respect to what would be expected during deployment, can erroneously inflate reported performance. For this, `spatial.py` also includes `assert_class_distribution()` to check that class balance reflects a given ratio within some variance.

Train and Predict Stages. During training, the classifier estimates the relationship between the features and the output variables. By default, Tesseract invokes the `fit` function on the given model, however, custom algorithms can be injected into the fitting stage to aid interoperability or for experimentation.

In the next stage, the classifier attempts to predict correct classes for the test objects in the current period. Tesseract will search for typical classification functions—prioritizing those which output raw scores (e.g., distance from hyperplane) and thus are more flexible for calculating metrics. However, custom decision functions can also be passed to the framework to override the default behavior.

Post-classification Stages. In *classification with reject option*, a classifier can choose *not* to classify a particular observation due to low confidence (e.g., class probability below a certain threshold); rejected objects are quarantined for manual inspection, and their predictions are not included in the performance results. However, Tesseract reports a *quarantine cost* associated with rejection, as manual inspection is time consuming. Rising quarantine costs signal the onset of *concept drift* and the aging of the classifier's model [8]. Policies for rejection, while optional, can be implemented by subclassing the `Rejector` stage class and overriding the `reject()`



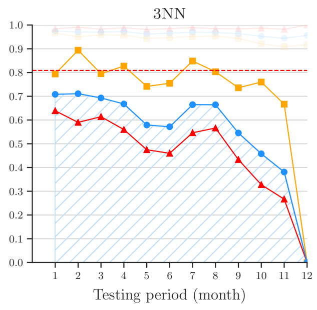


Figure 2: Plot generated by Tesseract showing a comparison of SVM and 3NN algorithms on Drebin feature space.

method. All `Rejector` subclasses will automatically keep track of which predictions were discarded, which are used to quantify the total quarantine cost.

Following the rejection stage, there is an opportunity to react to the predictions of the classifier before the rebalancing and retraining of the next evaluation cycle. For example, an *active learning* approach [14] can utilize a query strategy to select testing objects to be manually relabeled, which are then integrated into the training set before the next cycle. A popular query strategy is *uncertainty sampling*, which selects objects that the classifier was least certain about (e.g., those closest to the hyperplane in a binary SVM), because they are most likely to make the decision boundaries more precise once relabelled. Similarly to rejection, deriving ground truths for test objects is associated with a *relabeling cost*. Active learning techniques can be implemented in Tesseract by subclassing the `Selector` stage and overriding the `query()` method. Similarly to `Rejector` objects, all `Selector` stages will keep track of costs they incur. Tesseract includes some useful implementations for this stage, for example `UncertaintySamplingSelector` and `FullRetrainingSelector`. This design should encourage further experimentation with novel rejection and query strategies.

Metrics and Output. Tesseract maintains a set of metrics calculated during each iteration of the evaluation cycle. These range from the total positive and negative objects to metrics such as Precision, Recall, and AUROC. As Tesseract aims to encourage comparable and reproducible evaluations, we include functions for visualizing classifier assessments and for measuring the classifier robustness over a given time period with respect to each metric.

4 EXAMPLE

We present a case-study of Tesseract on Android malware analysis. We consider applications from AndroZoo [2], an open dataset which collects 6+ million apps with VirusTotal reports. We consider a test-case with 50K apps from Jan 2015 to Dec 2016, with 10% malware and 90% goodware—which is the expected ratio of malware-to-goodware in the wild [12]. We extract static features according to the Drebin [3] algorithm, which relies on a linear SVM; we perform grid-search to identify the best SVM hyperparameter, C=1. Figure 2 reports the output time-aware plots obtained by training on 2015 (25K apps) and testing on 2016 (25K apps), and compares the performance of SVM and kNN (k=3). The X-axis reports the time periods, and the Y-axis different performance metrics: F_1-Score, Precision, Recall for both malware (strong colors) and goodware (light colors). Figure 2 shows that SVM offers a better Precision-Recall trade-off than 3NN, whereas 3NN has similar F_1 over time with respect to SVM, but a higher Precision and a lower Recall. In both scenarios, the k-fold CV F_1 (dashed line) overestimates the classifiers' performance. We also observe that performance in detecting goodware does not drift much, and remains consistently high in both algorithms.

5 CONCLUSIONS

We have presented in detail the system design and implementation of the Tesseract [12] prototype, which can easily be used to remove temporal and spatial experimental bias when using machine learning in security contexts. Tesseract supports any Scikit-learn or Keras classifier and feature space, and is designed to be modular and generic.

AVAILABILITY

We make Tesseract's code and data available to the research community and practitioners. For access information, please contact Lorenzo Cavallaro <lorenzo.cavallaro@kcl.ac.uk>.

ACKNOWLEDGMENTS

This research has been sponsored by the UK EP/L022710/1 and EP/P009301/1 EPSRC research grants.

REFERENCES

[1] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Are Your Training Datasets Yet Relevant? In *ESSoS*. Springer, 2015.
[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting Millions of Android Apps for the Research Community. In *ACM MSR*, 2016.
[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.
[4] S. Axelsson. The Base-Rate Fallacy and the Difficulty of Intrusion Detection. *ACM Transactions on Information and System Security (TISSEC)*, 2000.
[5] C. M. Bishop. *Pattern Recognition and Machine Learning*. 2006.
[6] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-Scale Malware Classification Using Random Projections and Neural Networks. In *ICASSP*. IEEE, 2013.
[7] H. He and E. A. Garcia. Learning From Imbalanced Data. *IEEE TKDE*, 2009.
[8] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting Concept Drift in Malware Classification Models. In *USENIX Security*, 2017.
[9] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *NDSS*, 2017.
[10] Z. Markel and M. Bilzor. Building a Machine Learning Classifier for Malware Detection. In *Anti-malware Testing Research (WATeR) Workshop*. IEEE, 2014.
[11] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, et al. Reviewer Integration and Performance Measurement for Malware Detection. In *DIMVA*. Springer, 2016.
[12] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. *ArXiv*, 2018.
[13] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *IEEE Symp. S&P*, 2012.
[14] B. Settles. Active Learning Literature Survey. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2012.
[15] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In *ACM CODASPY*, 2017.
[16] A. Torralba and A. A. Efros. Unbiased look at dataset bias. In *CVPR*. IEEE, 2011.
[17] E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser. Benchmarking Crimes: An Emerging Threat in Systems Security. *arXiv*, 2018.