On the use of AutoML for Combating Alert Fatigue in Security Operations Centers

Davy Preuveneers[1], Javier Martínez Llamas[1], Irfan Bulut[2], Enrique Argones Rúa[2], Pieter Verfaillie[3], Vince Demortier[3], Dimitri Surinx[3], and Wouter Joosen[1]

[1] imec-DistriNet, KU Leuven, Belgium, davy.preuveneers@kuleuven.be
[2] imec-COSIC, KU Leuven, Belgium
[3] Spotit, Merelbeke, Belgium

Abstract. An overwhelming number of alerts – especially false ones – can desensitize analysts in security operations centers (SOC), possibly resulting in missed critical incidents and attacks going unnoticed. With inadequate alert monitoring, improper thresholds, and missing feedback loops as lead causes of alert fatigue, we investigate the use of automated machine learning to increase the efficiency of a SOC through automation of false alerts filtering. More specifically, we design a methodology to allow a safer use of AutoML to reduce false alerts, and validate this on a real-world case study. To be more precise, our approach is tailored to address datasets that exhibit limited instances of true positives, possess high dimensionality relative to their size, and demonstrate temporal fluctuations. We have identified diverse setups that provide comparable and reliably effective results in minimizing false positive alerts, all the while avoiding instances of false negatives. Furthermore, we provide valuable insights into the application of these automated frameworks within the realm of security.

Keywords: Security Operations Centers · AutoML · Alert fatigue.

1 Introduction

With the proliferation of advanced cyber threats becoming a growing concern, organizations rely on security operation centers (SOC) to continuously improve the organization's security posture. A SOC is a centralized hub that monitors the organization's IT infrastructure around the clock, while preventing, detecting, analyzing, and responding to potential security incidents.

When growing volumes of security alerts overwhelm security analysts, alerts will be ignored, incidents will not be investigated, and attacks will go unnoticed. Hence, the key to the success of a SOC is the ability to configure these monitoring systems and tune their alerting for the least number of false positives without missing any true security incidents. Security orchestration and automation (SOAR) playbooks can help fight alert fatigue by scripting repetitive work for known scenarios with a prescribed course of action, such as enriching an alert

with additional information and context. However, even if many monitoring systems provide severity or priority fields, improper thresholds may still cause false alerts. Tuning a single threshold to reduce false alerts while at the same time not introducing any false negative ones is not always straightforward because that threshold lacks context.

In this work, we leverage Artificial Intelligence (AI)-assisted techniques to combat false alerts and alert fatigue in general. Machine learning (ML) and deep learning (DL) are now mainstream − also in security [5] − to make sense of large volumes of data. In this application domain, an ML model can help to increase the efficiency of a SOC by filtering false alerts, leaving more time and resources for the security analysts for those signals and alerts that truly deserve their attention. However, constructing an effective ML classifier is time-consuming. Furthermore, new threats emerge continuously and these can trigger concept drift [10] causing existing ML models to become ineffective over time. Also, lack of ground truth, scarcity of true positives, and high dimensionality of the input space make the construction of these models cumbersome, and their generalization ability limited. These challenging conditions make it necessary to build a specific methodology to appropriately assess the *reliability*, *consistency*, *stability*, and *utility* of the different possible configurations of AutoML [9]. After proposing such a methodology, we have performed an extensive evaluation with real-world data with the intention of providing guidance toward a safer deployment of these types of systems in the real world. The main contributions are as follows:

- We propose an AutoML-based methodology to minimize false positive alerts, while avoiding instances of false negative alerts.
- For a real-world case study, we have identified diverse setups that provide comparable and reliably effective results in minimizing false positive alerts.
- We provide valuable insights into the application of these automated frameworks within the realm of security.

The remainder of this paper is structured as follows. We review relevant related work in Section 2. Section 3 discusses our approach on how we used AutoML to combat alert fatigue and systematically compared different ML pipeline configurations. We evaluate their benefits and drawbacks in Section 4. In Section 5, we conclude with summarizing the main insights and opportunities for further research.

2 Related work

In this section, we review related work on the application of AI and ML in security operations centers to combat *false alerts* and alert fatigue in general.

NoDoze is a system proposed by Hassan et al. [8] to combat alert fatigue by leveraging the contextual and historical information of threat alerts. NoDoze creates a causal dependency graph of an alert, and then an anomaly score for each edge to reflect the frequency of its past occurrence in the enterprise network.

The authors propose a network diffusion algorithm to aggregate these scores into a single value which is then used for triage. After setting a cutoff threshold for anomaly scores, they were able to reduce the volume of *false alerts* by 84%.

Addressing alert fatigue was also the goal of the work by Aminanto et al. [3]. Their approach exploits the temporal information in the alerts, and relies on online anomaly detection using Isolation Forests. They use the data of the previous day for training purposes and screen the incoming alerts simultaneously. On a 10-month dataset with 564,561 alerts, they were able to filter 87.41% of the false alerts without producing any false negative, i.e., without missing any true alert, by using an unsupervised method. The advantage of this method is that it does not require prior labeling, and it is lightweight to implement. A similar approach was proposed by Ahmed et al. [1]. They instead used Extended Isolation Forests to mitigate the bias that is induced by the tree branching in traditional Isolation Forest models. For an IDS dataset containing 50,000 alerts, the authors were able to reduce the number of false alerts by 82.15%.

Follow-up work by Aminanto et al. [2] augments the Isolation Forest unsupervised learning model with deep learning-based stacked auto-encoders (SAE) to further reduce false alerts. The SAE is a symmetric neural network model that aims to reconstruct its input after pushing through a bottleneck network layer with fewer neurons than its input layer. When trained on benign data, the SAE exhibits a high reconstruction error for critical alerts. Using the same experimental dataset as before [3], the authors now achieve a recall score of 95.89% and a false positive rate of 5.86%.

Ban et al. [4] proposed an alert screening scheme that uses AI-based tools to distinguish true threats from false alarms produced by a SIEM system. Their solution consists of carefully selected ML algorithms and visualization components to accelerate alert analysis and incident response. Their solution relies on 6 classifiers that can efficiently handle high-dimensional data (i.e., K-nearest neighbors, naive Bayes, linear discriminant analysis, decision tree, AdaBoost, SVM, and WSVM) as part of an ML pipeline that also takes care of feature selection and one-hot encoding. They report a recall rate of 99.598% for highly critical alerts and a false positive rate of 0.001% for a dataset of alert logs recorded over a 10-month period having 133.77 million alerts with only 593 confirmed by security experts as true positives. Building on top of the previous work, Ndichu et al. [12] aim to detect critical alerts but they specifically look at dealing with the imbalanced nature of the datasets, as well as the impact of having alert data from eight security appliances. Using the same dataset as Ban et al. [4], the authors evaluate the impact of the neighborhood cleaning rule (NCR) to eliminate ambiguous, noisy, and redundant false alerts. They then use the support vector machine synthetic minority oversampling technique (SVMSMOTE) to generate synthetic training true alerts. Finally, the authors evaluated the decision tree and random forest classifiers, demonstrating a significant reduction in false alerts while achieving a recall of 99.524%.

Pang et al. [13] argue that the security risks associated with models generated through Neural Architecture Search (NAS) − a subset of AutoML that

specifically targets the optimization of neural network architectures – are not well understood. By examining 10 popular NAS methods, the researchers find that models created through NAS are more vulnerable to various malicious actions like adversarial attacks, model tampering, and function copying compared to manually designed models. They argue this might be due to NAS methods favoring fast-converging models in early training stages due to the high search space and training costs. Their study not only uncovers the link between model features and vulnerability but also reveals connections between different attack types.

The gap that we aim to address in this work is the fact that in the real world security analysts rely on many open-source and/or commercial threat monitoring solutions. These are already able to automatically filter many low-risk alerts via automated playbooks, but they nonetheless still trigger false alerts. Our goal is not to operate on the raw data (e.g., replaying PCAP files), but rather to further filter the alert output of security appliances with ML-based techniques. This problem is much harder to solve than the one addressed in the literature.

3 Methodology

This section first elaborates on the real-world case study, and then describes, in more detail, the approach we followed to reduce false positive alerts.

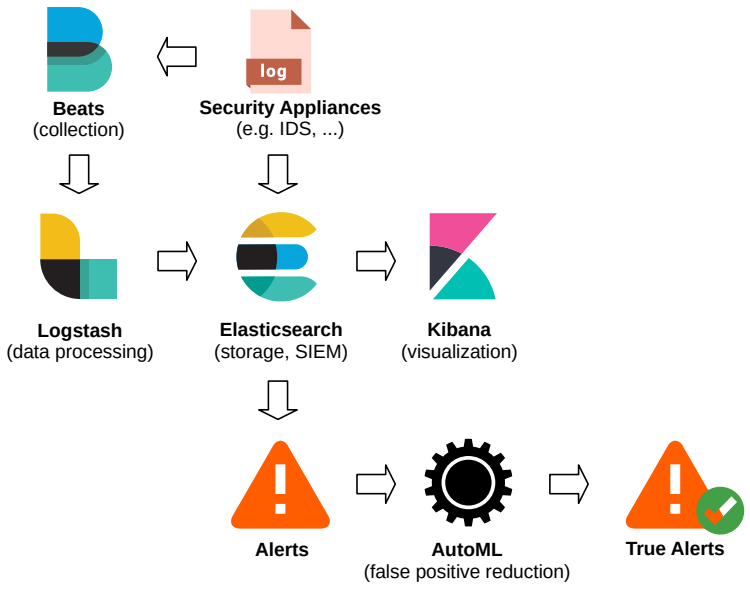


Fig. 1: Overview of alert reduction architecture on top of the ELK software stack.

3.1 Case study

Our case study builds upon the output of various security appliances that store their log outputs and alerts in an ELK stack as depicted in Fig. 1. The alert data that is used to feed our AutoML component is stored in the Elasticsearch database. For the sake of simplicity, we will only report on the results of one particular technology, in this case, a commercial extended detection and response (XDR) system that leverages network, endpoint, and cloud data to defend against sophisticated attacks. Under the hood, this security appliance uses out-of-the-box rules, as well as AI techniques, to detect attacks. We consider the behavior of this appliance as a black box, assuming no knowledge of the internal modus operandi nor any insights into the AI techniques used. The severity of each alert is computed as an aggregated score and a simple threshold mechanism already filters many non-critical alerts. The remaining ones are evaluated and possibly responded to by SOC analysts.

The objective is to design and evaluate a classifier that reduces the number of *false alerts* delivered to the SOC analysts, while preserving all *true alerts* for further examination. Our classifier will process the non-filtered alerts, which would be otherwise directly delivered to the SOC analysts. The output of the classifier would determine whether these alerts can be discarded as false alerts, or they require further attention. Therefore, our binary classifier can make two distinct types of errors, with different relevance: (1) false positives, which are false alerts classified as true alerts decreasing the utility of the classifier, and (2) false negatives, which are true alerts classified as false alerts. While the former are undesirable, the latter need to be avoided.

3.2 Security alert dataset

In collaboration with Spotit − a company offering security and SOC services to external customers − we collected and processed a dataset of alerts over a period of 10 months. The company uses the same assets to monitor its own activities. All the experiments were done with internal Spotit data and no client data was ever used or shared to mitigate any confidentiality and privacy or other ethical concerns when training with client data.

The XDR dataset is exported from the Elasticsearch database in JSON format, and after flattening the data structure, we end up with a tabular dataframe having more than 60 columns. These columns are of different types (i.e., categorical, numeric, lists, etc.) and may even have missing values. Alerts that pass a certain severity threshold are passed along and analyzed by a SOC analyst, who labels them as *true positive* (a real threat), *false positive* (a bogus or innocuous threat), a *duplicate* (a recurrence of a previous true or false alert), or *other* if the label is not clear. Alerts below the threshold are not investigated in a similar fashion, and hence do not follow this analysis pipeline. Therefore, they are out of scope and are labeled *true negative* alerts in our ML pipeline. The labels in this dataset are subsequently binarized by (1) removing the small number of *other* entries, (2) reusing the labels of the original alerts for the *duplicate* entries, and

(3) combining the *false positive* and *true negative* alerts in *class 0* (false alerts raised by bogus or innocuous events), and the *true positive* alerts in *class 1*. From this dataset, we create 3 variants with different degrees of manual feature engineering, which are then used as training and test set in our ML pipeline:

- **None**: No manual feature engineering based on knowledge or insights by domain experts. In this variant, min-max normalization is performed to numerical features, and ordinal transformation to categorical ones.
- **Pre-processing**: Create additional columns based on existing ones (e.g., split a URL feature in a separate domain and a path feature).
- **Encoding**: After pre-processing, additionally apply one-hot encoding and other data transformation methods on selected columns.

The second variant (and hence also the third) applies domain-specific knowledge to investigate whether this manual effort pays off to combat alert fatigue, or whether the AutoML pipeline can automate the feature engineering and selection in an equally effective manner even without any domain expertise or knowledge about the protected infrastructure.

3.3 AutoML-driven automation methodology

Our automation methodology is based on the Python-based AutoSklearn [7] framework. A classifier is implemented in a similar manner as in the well-known scikit-learn ML library[4].

The explored aspects of the AutoML pipeline include: types of scikit-learn classifiers to be included in the ensemble, total training time, training time per classifier in the ensemble, size of the ensemble, and the number of training tasks in parallel. Last but not least, AutoSklearn internally splits the training data into two parts, i.e., (1) one internal subset to train the individual models, (2) a second internal subset to evaluate the F1 score and other metrics. We implemented a custom strategy to subsample the original training set into two subsets taking into account the date of acquisition. Beyond selecting the best base ML models to construct the ensemble, AutoSklearn can also automatically optimize the feature selection and encoding, and apply various methods to reduce the dimensionality of data before training. These optional parameters were also investigated.

During initial tests, we noticed a significant variation across multiple runs with the same settings, which we attribute to a mismatch between the standard error metric(s) that AutoSklearn uses to optimize the final ensemble and the actual goal of reducing the number of false alerts (class 0). That is why we implemented our own custom error metric for AutoSklearn to optimize:

$$\text{custom error metric} = min(max(\frac{\text{FP}}{\text{FP} + \text{TN}}, \text{FN}), 1.0) \tag{1}$$

This metric aims first to eliminate false negatives, which is the primary requirement of our use case, and then to maximize the specificity, which is the utility function in our use case.

[4] `https://scikit-learn.org`

3.4 Evaluation methodology

Our evaluation pursues to evaluate *reliability*, *consistency*, *stability*, and *utility* of the trained classifiers. We train each configuration 10 times in order to allow the assessment of stability. Consistency, reliability, and utility are also verified in all the trained classifiers in each configuration. For these evaluations, the security alert dataset has been temporally split into 3 subsets:

- **Training set**: train the different AutoSklearn ensemble classifiers.
- **Validation set**: set appropriate thresholds of the classifiers.
- **Test set**: estimate a priori the performance of the classifiers.

Thresholding strategy We evaluated different thresholding strategies that aim at minimizing false positives while avoiding false negatives. For a given alert a, our classifier outputs a score s resembling an estimation of the probability of that sample being a true positive, i.e., $s \sim \mathrm{P}(a \in \text{class 1})$. In other words, the score is directly correlated to the certainty of the class 1 membership test by the classifier. Therefore, in order to avoid false negatives the threshold must be set below the minimum score obtained by a true alert in the validation set, which we call $\theta_{\max}$. It is convenient to set also a security margin, to further ensure avoiding false negatives. We use both a fixed and a statistical margin for this purpose. In the fixed threshold strategy, we simply subtract a small fixed margin m from $\theta_{\max}$, thus using a threshold $\theta_m = \theta_{\max} - m$. The statistical margin strategy is a bit more involved. For a given margin $p\%$ in percentage, the threshold is set to a value $\theta_{p\%}$, such that an additional $p\%$ false positives are allowed with respect to $\theta_{\max}$. So, the margin is set to allow a fixed decrease in the utility of the classifier. Each of these thresholds, as illustrated in Fig. 2, give us a different a priori working point that we are evaluating.

Statistical characterization For each trained classifier we compute the following statistics in order to assess its individual consistency, reliability, and utility:

- A posteriori utility in the validation and test set. Since the utility is directly linked to the reduction of false positives, it is actually equivalent to the specificity, where:

$$\text{specificity} = \frac{\#\text{true negatives}}{\#\text{true negatives} + \#\text{false positives}} .$$

- A priori validation of the security constraint, i.e., #false negatives = 0 in the test set.
- The a priori eligibility of the classifier, in terms of the specificity in the validation set.

We characterize each ML configuration by analyzing these statistics along the 10 different trained systems, as follows:

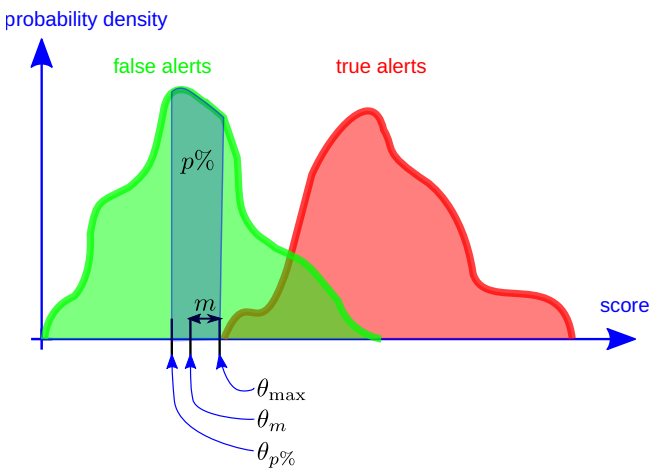


Fig. 2: Validation set score distributions and thresholds.

- **Unreliable**: One or more ML models with this configuration produces false negatives in the test set. This is the worst scenario, since eventually, an ML model trained with this configuration will provoke a security issue.
- **Inconsistent**: These ML models do not produce any false negatives in the test set, but the specificity in the test set is not always high, showing a lack of generality.
- **Not Eligible**: These ML models do not produce any false negatives in the test set, but do not obtain a high enough specificity in the validation set, which would support their deployment.
- **Consistent High Utility**: These ML models produce a high specificity in the validation set and also produce a high specificity in the test set (comparable to the specificity of the validation set), and without any false negatives.

The methodology explained above contains several design choices. For this particular case study, we decided to use the $\theta_{5\%}$ threshold for setting the classifiers' working point as a sensible safety margin, since it provides a safer threshold, while keeping the expected extra workload at only 5%, which is acceptable. Regarding the inconsistency assessment, we decided that, given the specificity of the validation set, an absolute worsening of 8% or more in the test set can safely be considered as a symptom of inconsistency. However, as we will show in Section 4, the consistency test fails very rarely, so the impact of this in the evaluation would be marginal. In the case of eligibility, the 20% minimum specificity in the validation set is just a sensible minimum expected utility for the classifier. Even though it may be seen as a low utility, it would have a positive impact on the SOC analysts' workload.

Regarding the reliability assessment, we are aware that it is very strict. Some ML configurations may provide excellent utility, eligibility, and consistency in most cases. However, if a single instance of this configuration produces a single

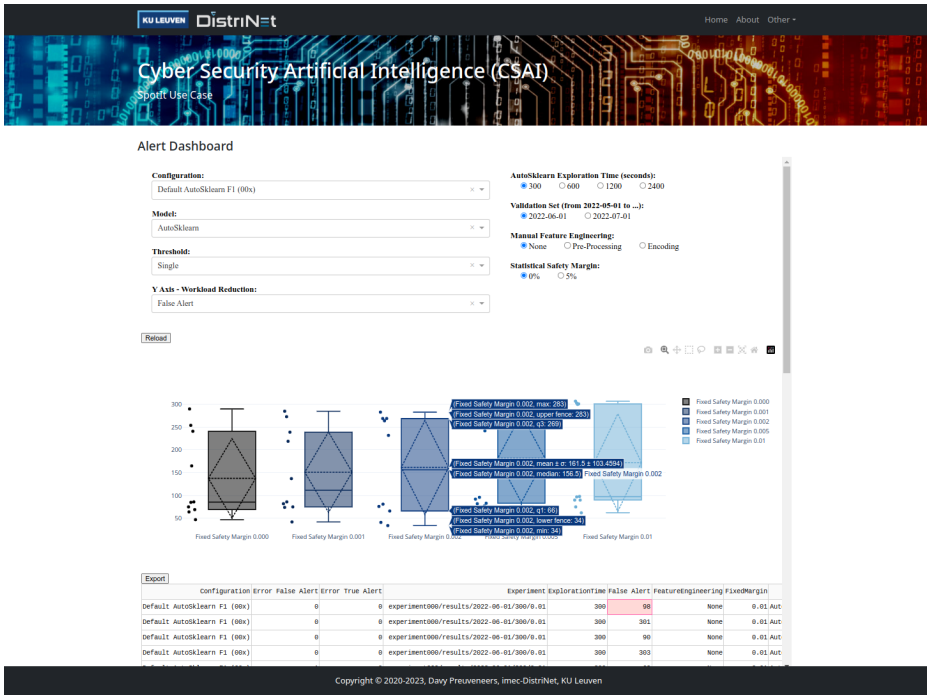


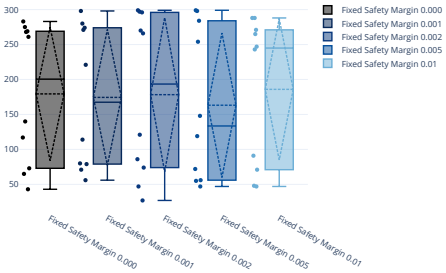
Fig. 3: Interactive visualization dashboard to compare results of different ML pipeline configurations.

false negative, then this represents statistical evidence that the configuration can provoke a security issue. As such, the safest decision is to label the configuration as unreliable. We must acknowledge that even when none of the 10 trained instances of a given configuration produces a false negative, it does not categorically guarantee reliability if the configuration would have 100 instances rather than 10. However, we at least did not find any evidence of a lack of reliability in any of the 10 instances, so we can consider the configuration as reliable (it could anyway end up being not eligible, or inconsistent).

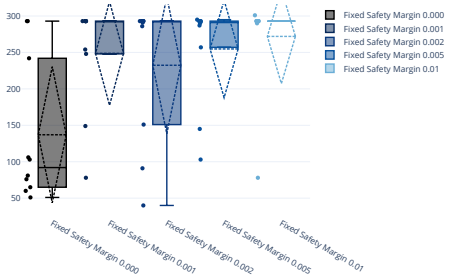
3.5 AutoSklearn ML pipeline configurations and interactive dashboard

In order to ease a visual exploration of the performed experiments, an interactive visualization dashboard was developed (see Fig. 3) allowing to explore and easily export the experimental results by selecting the following features:

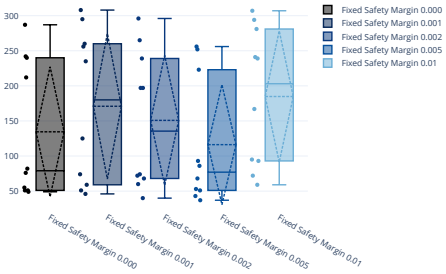
– **Configuration**: To explore different configurations of the classifier, as the optimization objective, or custom data splitting during training, etc. The optimization objectives include all standard metrics, and a custom metric designed specifically to avoid false negatives while minimizing false positives.



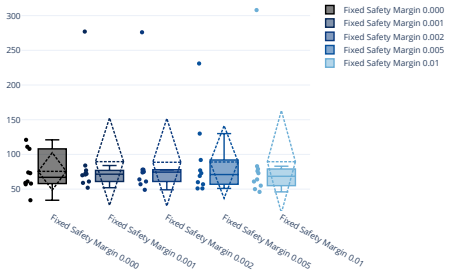
(a) Exploration: 300s, Manual feature engineering: None.



(b) Exploration: 300s, Manual feature engineering: Encoding.



(c) Exploration: 1200s, Manual feature engineering: None.



(d) Exploration: 1200s, Manual feature engineering: Encoding.

Fig. 4: Variation on the test set across 10 runs with ROC-AUC optimization objective and 0% statistical safety margin with different AutoSklearn exploration times, with and without manual feature engineering. The Y-axis counts the number of **remaining false alerts** in each run that remain to be analyzed by the SOC analyst (lower value is better).

- **AutoSklearn Exploration Time**: The amount of time (in seconds) granted to AutoSklearn to construct the best ensemble. More time may result in better ensembles due to more base ML models.
- **Model**: Comparison of AutoSklearn classifier with traditional ensemble models (e.g., RandomForest or XGBoost) with default parameters.
- **Validation Set**: Impact of a different temporal split between the end of the training dataset and the beginning of the validation dataset.
- **Manual Feature Engineering**: Added value of the (time-consuming) manual feature engineering before training the ML model.
- **Fixed Margin**: Determines the fixed margin m used during the evaluation, which can be set to $0, 0.001, 0.002, 0.005$ and 0.01.
- **Statistical Margin**: Determines the statistical margin used during the evaluation, which can be set to 0% or 5%.

This tool makes the comparison of the impact of different configurations somewhat less complicated, by showing various statistical properties of the ML

ML configuration	None			Pre-processing			Encoding		
	ET=300	600	1200	ET=300	600	1200	ET=300	600	1200
Accuracy	C	U	U	N	N	U	U	C	C
Balanced Accuracy	N	N	N	U	N	U	N	N	N
F1	I	C	N	U	N	U	C	U	U
Precision	U	U	U	U	U	N	N	U	U
Recall	N	N	N	N	N	N	N	N	N
ROC-AUC	U	U	U	N	C	U	U	C	C
Weighted F1	U	U	U	N	N	C	N	U	U
Weighted Precision	U	C	C	C	C	C	U	N	N
Weighted Recall	U	U	U	N	N	C	C	C	U
Custom Metric $m = 0.000$	N	N	N	N	N	U	C	N	N
Custom Metric $m = 0.001$	N	I	N	U	U	N	C	N	N
Custom Metric $m = 0.002$	N	C	C	N	N	N	C	C	C
Custom Metric $m = 0.005$	C	C	N	N	N	U	N	N	N
Custom Metric $m = 0.010$	N	N	N	N	N	N	N	N	N
Random Forest $m = 0.000$	C	C	C	U	U	U	N	N	N

Table 1: Performance summary of relevant ML configurations, using all feature engineering and exploration times. C, I, N, and U stand for Consistently high utility, Inconsistent utility, Not eligible, and Unreliable security respectively. Each entry is the result of the analysis in Section 3.4 in 10 independent experiments using the same configuration.

model and its predictions. Most importantly, from this dashboard it is also possible to directly export all the relevant statistics for the evaluation performed in this paper.

4 Evaluation and discussion

In this section, we will elaborate in more detail on the insights we gathered with the use of AutoSklearn. In fact, we trained more than 8,700 AutoSklearn ensemble models, and these ensembles were composed of a total of about 1,200,000 individually trained base ML models. However, the goal of our evaluation is not to report which classifier achieved the best performance numbers, as these numbers are heavily dependent on (1) the underlying security technology and the logging data it produces (i.e., only XDR data), (2) the infrastructure being monitored for attacks, (3) the period of data collection, (4) the size of the training, validation, and test set.

Indeed, making any claims about generalization towards other experimental settings is almost impossible. Instead, in the following subsections, we will elicit

various lessons learned from the many experiments we carried out, in the hope that these are much more useful for the community. The results of our experiments have been summarized in Table 1, where each configuration produces one of the behaviors described in Section 3.4. The table shows the analysis of only $15 \times 9 = 135$ configurations. Our study covered 1710 configurations with different configuration parameter combinations. Additionally, we have tested the impact of the pre-processing step by repeating the experiments without it. Depending on AutoSklearn's choice to enable its automated data and feature pre-processing, the categorical fields may or may not be encoded and numerical fields may or may not be normalized. When using the data as is, one-hot encoding significantly increases the dimensionality of the data, with the ML model failing to generalize. Our additional experiments have demonstrated that under these conditions AutoSklearn failed to produce even one ML configuration that achieves consistent high utility.

4.1 Impact of manual feature engineering

Manual feature engineering is a time-consuming task that requires expertise and domain knowledge about the protected infrastructure. It is not always clear upfront whether this effort will pay off, or whether the automated data and feature pre-processing of AutoSklearn can obtain equally good results. Our analysis in Table 1 compares the impact in the columns **None**, **Pre-processing**, and **Encoding** for different metrics and exploration times. While equally good performance results in the test set exist for each of the combinations, they are not consistent. As exceptions, Weighted Precision seemingly produces remarkably good performance for the **Pre-processing** strategy, and the custom metric with $m = 0.002$ does the same for the **Encoding** strategy, but in all other cases, the metrics can perform well for some exploration times, and not consistently for others. From this comparison, we can conclude that feature representation has a big impact on performance. However, this observation is connected with the optimization metric, hereby recommending to try out experiments using different optimization metrics and feature representations aligned with the use case at hand. To further illustrate this, Fig. 4 compares different AutoSklearn exploration times while optimizing for the ROC-AUC metric. The impact of manual feature engineering is limited to 300 seconds of exploration time (see Figures (a) and (b)). However, for 1200 seconds, Figure (d) appears to demonstrate some benefits. For a larger safety margin, there are fewer false alerts with less variation across the 10 runs, yet with no guarantees regarding the presence of false negatives. Also, optimizing for the F1 error metric) results in a less outspoken impact of manual feature engineering.

4.2 Stochastic nature of repeated experiments

Table 1 shows a large variability in the presence of false negatives (*true alerts* classified as *false alerts*) in the test set, even after applying a conservative thresholding approach ($\theta_{5\%}$). This can be observed for all exploration times and feature

processing strategies (with some rare exceptions, such as 'Weighted Precision' with 'Pre-processing' feature engineering, and the custom error metric discussed later using $\theta_{0.002}$ thresholding and 'Encoding' as feature engineering).

Although our methodology is devoted to evaluating the consistency of the a priori performance of the different trained classifiers, we must consider the fact that the distribution of future data may significantly differ from those of the validation and test datasets, so it is not possible to guarantee consistent performance in the future. This is especially relevant given the big number of configurations labeled as Unreliable in Table 1. For many of the configurations with such a label, only one or two of the trained classifiers produced false negatives, which highlights how difficult it may be to ensure consistency.

For an imbalanced dataset, the ROC-AUC and F1 metrics are sensible optimization metrics for AutoSklearn. Nonetheless, we also explored other optimization metrics − including F1, Accuracy, Precision and Recall next to ROC-AUC − and we observed the same variation in false alert reduction across multiple runs, and keeping AutoSklearn exploring ML models for much longer does not necessarily help to achieve more consistent results.

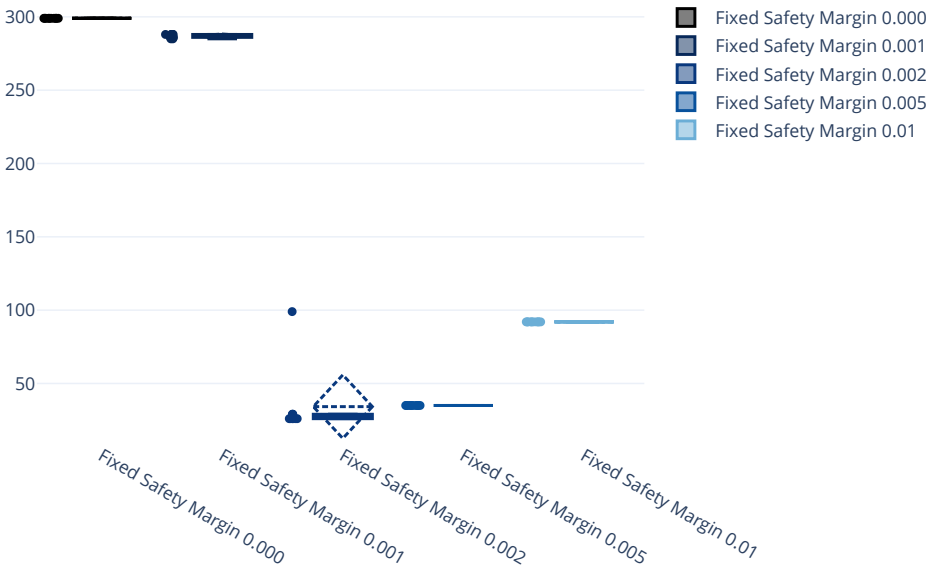


Fig. 5: Remaining false alerts with custom optimization metric.

The results in Fig. 5 show that AutoSklearn now produces a consistently high utility within the same configuration, i.e., there is hardly any variation across the 10 repeated experiments. Note that this does not guarantee that there are no false negatives in the test set! However, in many cases, this metric achieves not only the goal of reducing the number of false alerts, but also avoiding false negatives, as it may be observed in Table 1.

4.3 Out-of-distribution samples and temporal constraints on validation and test datasets

In the security landscape, new threats and attacks emerge on a daily basis, and ML models may perform poorly on data samples it was not trained upon. To account for out-of-distribution samples, the alerts in the training, validation, and test datasets should not be randomly sampled from the master dataset, but preserve their chronological order. For example, the alerts in the validation set should occur after all those in the training set, and the ones in the test set should have a timestamp later than those in the validation set. This is rather straightforward and a common best practice for ML in security applications.

However, when AutoSklearn exploits the training data to construct the best possible ensemble model, it starts by splitting up the original training data into two internal subsets where the chronological order of the alerts across these two subsets is not guaranteed. By default, it uses a holdout strategy where a 67% random subset of the original training data is used for internal training and the remaining 33% of the data is used for internal testing and optimization. Indeed, this holdout strategy or an alternative cross-validation strategy does not preserve the chronological order of the alerts in this internal training set and internal test set (i.e., some internal test samples may predate some of the internal training samples). We, therefore, developed a custom resampling technique so that AutoSklearn also internally preserves this property.

We compared the impact of a custom resampling strategy and noticed a consistent improvement in terms of true alerts that went unnoticed, while also having a slight improvement in the false alert reduction.

While we were able to implement and test our custom resampling strategy within the AutoSklearn framework, we noticed that there was a practical limitation with the PredefinedSplit API used in our custom resampling strategy. After loading the model from disk serialized as a Pickle file, we had to first 'refit' the model again against the training data. While this makes the proposed approach rather time-consuming and hence impractical, we hope that this practical concern can be alleviated in a future version of AutoSklearn.

4.4 Interpretability and explainability

The ML models that AutoSklearn constructs, can become fairly complex. By default these are ensemble models that combine up to 50 base ML models, and each of these base models can be any classifier that scikit-learn supports (e.g., a Decision Tree, Random Forest, Support Vector Machine, k-Nearest Neighbors, Multi-Layer Perceptron). Unfortunately, such ensemble models are hard to interpret by a human SOC analyst [6].

To address this concern, we modified the AutoSklearn classifier configuration to only produce ensembles with a single base ML model, and constrain that base model to be a Random Forest classifier. The results depicted in Fig. 6 were obtained after 300 seconds of exploration time and without any manual feature engineering. This particular model performed the best (i.e., false alerts

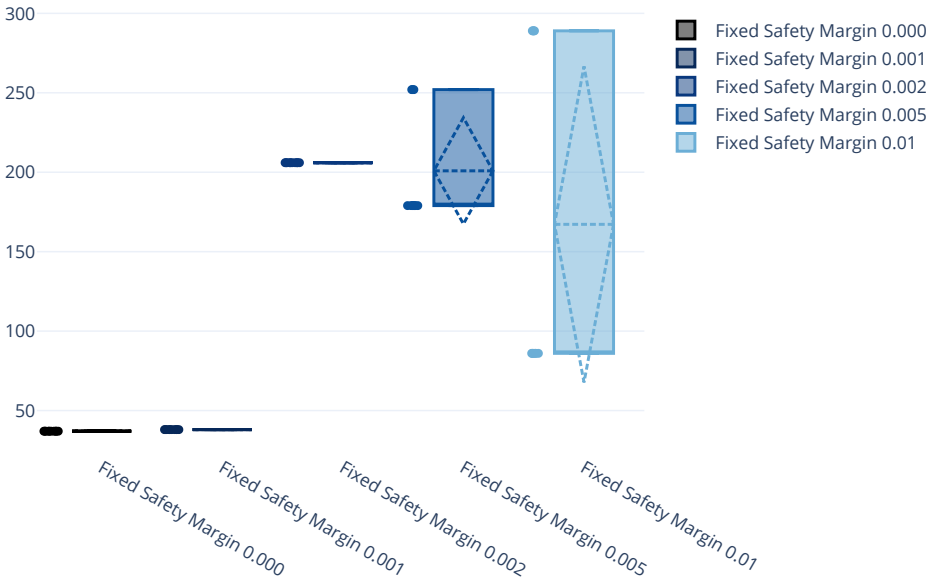


Fig. 6: Enforcing a single RandomForest base model in the ensemble.

reduced from 308 down to 19, and this without any missed true alerts), even without a fixed safety margin. The results for this classifier are shown in the last row of Table 1 for $\theta_{5\%}$. This particular setup performs consistently well for all exploration times (although a priori performance in test set is best for $ET = 300$).

However, by default, AutoSklearn also applies many data and feature pre-processing to assess whether these have a positive influence on optimizing the error metric. Some data pre-processing techniques include (a) dealing with imbalanced datasets, (b) imputing missing values, (c) one-hot encoding of categorical values, (d) rescaling for faster convergence, (e) variance thresholding to reduce dimensionality, etc. Other feature pre-processing that AutoSklearn attempts, include (a) Principle Component Analysis and related dimensionality reduction techniques, (b) univariate or (c) classification-based feature selection, (d) feature clustering, and many more. The automated pre-processing and transformation of the raw training data as the many dimensionality reduction methods make it hard to truly understand the context under which certain samples are correctly classified as a true alert (class 1) or a false alert (class 0), as the transformed input for a Random Forest classifier may no longer be semantically meaningful.

The advantage of Decision Tree-based models is that these are much easier to convert into rule- or signature-based schemes as implemented by the security technology used to detect the threats and raise alerts. An alternative model agnostic approach to explain the output of a model is via an eXplainable AI (XAI) techniques, such as its SHAP (SHapley Additive exPlanations) values [11] or LIME [14].

Obviously, when interpretability is not a concern, then there is no need to constrain the amount nor the type of the base ML models in the AutoSklearn ensemble, nor is it necessary to disable automated data and feature pre-processing. A valid scenario is the use of the ML model merely to prioritize the order in which alerts are handled by the SOC analyst, and not to automatically filter alerts.

4.5 On safety margins

The specific use case analyzed in this paper imposes some constraints that need to be taken into account. The most important one is the high cost of false negatives, which must be avoided. Thus, the working point of the classifier needs to be set accordingly to a region where no false negatives are expected. In our experiments, we compute this threshold in two different manners, but with the same objective, i.e., to avoid false negatives while maximizing True Negatives. The fixed thresholding strategy simply moves the threshold slightly by a fixed amount, while the statistical margin thresholding moves it according to the True Negatives distribution, thus allowing for a fixed loss in utility for the sake of security. Although this strategy seems sensible, from the results shown in Table 1 it is clear that the absence of false negatives in the test set cannot be guaranteed. However, from our analysis with the two different strategies and parameters, it is clear that the use of these safety margins does provide additional assurance in that regard.

4.6 Discussion

Among all tested configurations, we found the best consistent utility for a single Random Forest with 300s exploration time, only AutoSklearn automatic feature processing, and using our custom metric. However, other configurations achieve similar utility and consistency, such as ensembles using Weighted Precision with manual feature pre-processing or our custom metric with $m = 0.002$.

Obviously, the impact of (manual) data and feature pre-processing is very specific to this use case. We observed that by applying one-hot encoding on various columns in our dataset the dimensionality simply exploded by almost two orders of magnitude. In principle, this is not a problem as long as one has sufficient data. Otherwise, there is a risk that a model easily overfits, which was the case in our scenario.

Also, the adopted a priori evaluation methodology aids to observe consistency of each classifier configuration. The observed variability in behavior and performance of the various classifiers raises concerns, but also warns developers about the lack of generalization capability of many of the trained classifiers, which would remain undiscovered otherwise. From our evaluation, we can conclude that even for classifiers with a consistent high utility we could benefit from a further assessment

1. by using larger datasets (to increase the representation of True Alerts)
2. originating from diverse sources (as our datasets come from the same source)

3. and that span a wider temporal frame (to further assess concept drift issues)

in order to achieve the necessary confidence on the performance over future data collected after deployment.

As future work, we will additionally explore the adversarial robustness of the aforementioned ensemble models. While the literature is rife with studies on how to create adversarial examples and how to defend models with adversarial training, our case study is significantly more complex as our models are trained on the logs of security monitoring tools, and not on the raw data (e.g., network traffic) directly. The fact that model explanations can be manipulated too [15] further complicates this challenge.

5 Conclusion

We investigated the benefits and drawbacks of automated machine learning (AutoML) to address alert fatigue in a security operation centers (SOC) environment. The goal was to increase the efficiency of a SOC by automatically filtering the false alerts produced by various security monitoring and threat detection technologies. We evaluated the use of AutoSklearn against a real-world case study and found that the framework has its benefits, but one must be aware of possible pitfalls.

We proposed a methodology designed to cope with a difficult dataset, with scarce true positives, large dimensionality in comparison to its size, and temporal variability. Using this methodology, we offer critical reflections on the use of such automation frameworks in this security context. In this exploration, we found different configurations that offer equivalent and consistent good utility in terms of false alerts reduction, while not producing false negatives.

Further work will investigate the benefits of model explainability methods that may be more tailored to the needs of a SOC analyst, who require understanding the reasons behind the classifiers' outputs.

Acknowledgment

This research is partially funded by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by VLAIO through the CS ICON project "Cyber Security Artificial Intelligence" (CSAI). We would like to thank Spotit, a member of the CSAI project consortium, for permission to use their dataset.

References

1. Ahmed, T., Shah, A., Kolla, M., Yellasiri, R.: Reduction of alert fatigue using extended isolation forest. In: 2021 International Conference on Forensics, Analytics, Big Data, Security (FABS). vol. 1, pp. 1–5 (2021). https://doi.org/10.1109/FABS52071.2021.9702617

2. Aminanto, M.E., Ban, T., Isawa, R., Takahashi, T., Inoue, D.: Threat alert prioritization using isolation forest and stacked auto encoder with day-forward-chaining analysis. IEEE Access **8**, 217977–217986 (2020). https://doi.org/10.1109/ACCESS.2020.3041837
3. Aminanto, M.E., Zhu, L., Ban, T., Isawa, R., Takahashi, T., Inoue, D.: Combating threat-alert fatigue with online anomaly detection using isolation forest. In: Gedeon, T., Wong, K.W., Lee, M. (eds.) Neural Information Processing. pp. 756–765. Springer International Publishing, Cham (2019)
4. Ban, T., Samuel, N., Takahashi, T., Inoue, D.: Combat security alert fatigue with ai-assisted techniques. In: Cyber Security Experimentation and Test Workshop. pp. 9–16 (2021)
5. Bertino, E., Kantarcioglu, M., Akcora, C.G., Samtani, S., Mittal, S., Gupta, M.: Ai for security and security for ai. In: Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy. pp. 333–334 (2021)
6. Došilović, F.K., Brčić, M., Hlupić, N.: Explainable artificial intelligence: A survey. In: 2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO). pp. 0210–0215. IEEE (2018)
7. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems 28 (2015). pp. 2962–2970 (2015)
8. Hassan, W.U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., Bates, A.: Nodoze: Combatting threat alert fatigue with automated provenance triage. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society (2019)
9. He, X., Zhao, K., Chu, X.: Automl: A survey of the state-of-the-art. Knowledge-Based Systems **212**, 106622 (2021)
10. Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., Zhang, G.: Learning under concept drift: A review. IEEE Transactions on Knowledge and Data Engineering **31**(12), 2346–2363 (2018)
11. Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. Advances in neural information processing systems **30** (2017)
12. Ndichu, S., Ban, T., Takahashi, T., Inoue, D.: A machine learning approach to detection of critical alerts from imbalanced multi-appliance threat alert logs. In: 2021 IEEE International Conference on Big Data (Big Data). pp. 2119–2127 (2021). https://doi.org/10.1109/BigData52589.2021.9671956
13. Pang, R., Xi, Z., Ji, S., Luo, X., Wang, T.: On the security risks of AutoML. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3953–3970. USENIX Association, Boston, MA (Aug 2022)
14. Ribeiro, M.T., Singh, S., Guestrin, C.: "why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 1135–1144 (2016)
15. Slack, D., Hilgard, A., Lakkaraju, H., Singh, S.: Counterfactual explanations can be manipulated. Advances in Neural Information Processing Systems **34**, 62–75 (2021)

Appendix: AutoSklearn implementation

```
# Min-max data normalization and ordinal encoding of categories (if needed)
preprocessor = make_preprocessor(X_train)

# Enforce temporal splits
selected_indices = ((X_train['timestamp'] > split_time)
  & (X_train['timestamp'] < train_time)).astype(int)
custom_resampling_strategy = PredefinedSplit(test_fold=selected_indices)

# Configure the AutoML classifier
automl_model = AutoSklearnClassifier(
  time_left_for_this_task=1200,     # Exploration time (in seconds)
  per_run_time_limit=240,           # Exploration time / 5
  memory_limit=65536,               # 64 GB
  tmp_folder="tmp",
  metric=f1,                        # accuracy, roc_auc, recall, ...
  scoring_functions=[accuracy, f1, precision, recall],
  n_jobs=8,
  ensemble_size=1,
  include={"classifier": ["random_forest"],
      "data_preprocessor": ["NoPreprocessing"],
      "feature_preprocessor": ["no_preprocessing"]
  },
  resampling_strategy=custom_resampling_strategy) # enforce temporal splits

model = make_pipeline(preprocessor, automl_model)
# Fit the model on training set
model.fit(X_train, Y_train)

# Evaluate on test set
Y_pred = model.predict(X_test)

print("Accuracy:  ", accuracy_score(Y_test, Y_pred))
print("F1 score:  ", f1_score(Y_test, Y_pred, average='weighted'))
print("Precision: ", precision_score(Y_test, Y_pred, average='weighted'))
print("Recall:    ", recall_score(Y_test, Y_pred, average='weighted'))
```

Fig. 7: Code snippet of AutoSklearn ML pipeline, configuring a single Random Forest classifier in the ensemble without any data and feature pre-processing by AutoSklearn.