Machine Learning-Based Run-Time Anomaly Detection in Software Systems: An Industrial Evaluation

Fabian Huch*†, Mojdeh Golagha*, Ana Petrovska*, and Alexander Krauss†
* Technical University of Munich
Munich, Germany
{huch, golagha, petrovsk}@in.tum.de
† QAware GmbH
Munich, Germany
alexander.krauss@qaware.de

***Abstract*—Anomalies are an inevitable occurrence while operating enterprise software systems. Traditionally, anomalies are detected by threshold-based alarms for critical metrics, or health probing requests. However, fully automated detection in complex systems is challenging, since it is very difficult to distinguish truly anomalous behavior from normal operation. To this end, the traditional approaches may not be sufficient. Thus, we propose machine learning classifiers to predict the system's health status. We evaluated our approach in an industrial case study, on a large, real-world dataset of $7.5 \cdot 10^6$ data points for 231 features. Our results show that recurrent neural networks with long short-term memory (LSTM) are more effective in detecting anomalies and health issues, as compared to other classifiers. We achieved an area under precision-recall curve of 0.44. At the default threshold, we can automatically detect $70\,\%$ of the anomalies. Despite the low precision of $31\,\%$, the rate in which false positives occur is only $4\,\%$.**

I. Introduction

An enterprise software system is a software system designed as a part of a business that interacts with users, typically responding to web-based requests. During the operation of an enterprise software, anomalies might occur that impede the system's capabilities of serving properly. In this work, we define an *anomaly* as "any unintended state of a system that has or might have negative impact on its normal operation". This includes cases such as fatal conditions that bring nodes to a halt (e.g. out-of-memory errors) or more subtle issues such as hung threads that keep consuming resources, while the rest of the system is still up with reduced capacity. In both cases, we consider the system to be in an "unhealthy" state.

Fatal anomalies are obvious once a node stops responding, whereas non-fatal anomalies might continue unnoticed for a while. In any case, it is desirable that the anomaly is detected early on. If so, the system can react in time by automatically restarting affected components, and the problem does not impact the users' experience negatively. To achieve this goal, practitioners use monitoring techniques to constantly check the "health" status of the system.

Traditional monitoring approaches often employ a combination of a) fixed rules and thresholds for the values of the important system metrics, such as memory or CPU usage, b) probing requests that are expected to return successfully within a certain period of time, and c) customized health checks defined for each specific system. These approaches, whose precise rules must be defined *a priori*, may fail to detect complex and non-fatal anomalies. For example, a system might seem healthy based on the probing request, but exhibits low performance on certain types of user requests. In addition, in-depth expert knowledge about the system's internals is required to define the monitoring rules and thresholds.

For DevOps, the traditional methods may not be not sufficient, as anomalies need to be detected accurately with a very high degree of automation. We propose an alternative approach. Instead of relying on fixed rules, we build machine learning (ML) models for detecting health issues and anomalies. Together with the engineers at QAware[1], with whom we conducted this research, we have identified the problem statement and the solution as follows:

Problem. Insufficient or delayed detection of run-time anomalies in complex enterprise systems leads to performance issues and service failures. Is it possible to detect anomalies ahead of failures, in order to prevent negative user experiences?

Solution. We propose to use ML on operational data to create a model to predict the system's health status, to detect if any kind of anomaly is happening.

Contribution. We demonstrate our methodology on a complex real-world enterprise application, and discuss results of various supervised algorithms and the applicability of the approach. For this, we utilize a total of $7.5 \cdot 10^6$ measurements; our dataset is published in [1] and [2]. To the best of our knowledge, there is no prior work in incorporating data from a live real-world system into a ML model to detect operation anomalies.

Organization. In Section II, we explain our approach. We describe the experiment in Section III, including setup,

[1] https://www.qaware.de

evaluation metrics, and results. In Section IV, we review related work. Finally, in Section V, we discuss our findings and conclude.

II. Methodology

In the following, we describe our data collection, preparation, and ML steps.

A. Approach

To build an anomaly detection model, we collect related data from multiple sources first. We clean the collected data by removing/replacing missing values and eliminating invalid measurements. We define anomaly detection as a *binary classification* task. Thus, we label the data, using the "anomaly" and "healthy" class labels. Finally, we train and tune our ML models to find the best model for our case. In the following, we explain these steps in detail.

1) Collecting Data: To detect the operational anomalies of our complex enterprise application, we considered the operating system, and WebLogic Server monitoring beans, as our data sources. A total of twenty instances of the application were active on multiple hosts at the same time; as they are uniform, we do not differentiate between them when detecting anomalies. We measured 831 metrics in 1-minute time intervals, and used the time series of these measurements to form our dataset. As the result, our dataset contains $7.5 \cdot 10^6$ data points, or $5{,}209$ days of time series data. Traditionally, a few of the above mentioned metrics would be used to check the health status of the system. The metrics include CPU load, usages of different memory areas and thread pools throughout the application, connection delays to its different data sources, etc. A complete list can be found in [1].

2) Handling Missing Values: Incomplete data can occur when performing long-term measurements for different reasons. During system restarts, or when the system is unresponsive such that a restart is required, measurements cannot be made correctly. In addition, the set of measured metrics might change over time. Hence, the collected data might contain missing values.

To handle missing values, first we select a subset of metrics for which the majority of the values are present. Then, we replace missing values using linear interpolation [3]. This allows us to utilize ML algorithms that can not handle incomplete data. Additionally, we want to retain the information that measurements were unavailable at certain points in time, even if we could interpolate the missing values. For this, we introduce a feature that indicates at which points in time the measurements failed.

3) Creating Features: Some metrics like the *absolute* usage of a resource or *total* count of the occurrences of an event might not be very informative on their own. Combining these metrics or deriving a function can be more informative and also less influenced by configuration changes. With the help of the experts, we selected a set of metrics, combined some of them, and defined some functions, as our feature set. For example, the *size of the allocated heap* alone is not very informative, but combined with the *maximum heap size*, a feature for the *relative heap usage* is useful. Table I provides a summary of our features, the whole list can be found in [2].

TABLE I
Summarized Feature Set

Database connections	Currently active connections
	Database connection activity
	Database connection delay
	Failed reserve connection requests
	Relative unavailable connections
	Reserve connection request activity
	Started/Failed/Successful connections
Memory spaces	Activity in JVM memory spaces
	Physical memory activity
	Relative JVM memory space usages
	Relative physical memory usage
Transactions	Currently active transactions
	Transaction rollback/commit activity
Threads	Daemon/Total thread count
	Stuck threads
Swap	Relative swap usage
	Swap activity
Other	Class loading/unloading activity
	CPU usage/time
	Garbage collection activity/duration
	Hit rate of prepared statement caches
	JVM objects to be finalized
	Missing data
	Relative open file descriptors
	Relative physical memory usage

4) Labeling the Data: We define anomaly detection as a binary classification task. Thus, we need to assign our data into "anomaly" and "healthy" classes. Considering the huge amount of the data, scanning the whole dataset manually to find anomalies sounds very difficult and time-consuming, if not infeasible. To make this task faster and easier, we use restarts as an indicator for the possible health issues, since the application was restarted once an issue became apparent. With the help of experts, we analyze the behavior of the system preceding the restarts. If the behavior of the system is abnormal or anomalous, we label all the feature vectors in that specific period of time as "anomaly". The rest of data is then "healthy".

5) Balancing the Dataset: Imbalanced class distribution is a predominant problem in anomaly detection. In our case, the number of observations belonging to the "anomaly" class is significantly lower than those belonging to the "healthy" class. But most classifiers perform better when learning on more balanced data [4]. Since we have a huge amount of data, we balance the training set to some extent by random undersampling [5] of the "healthy" class. As a result, we have a $62.3\,\%$ "healthy" and $37.7\,\%$ "anomaly" distribution.

6) Normalizing the Data: Anomaly detection models perform better on normalized data [6]. In order to avoid a bias in the results, we normalize our features to zero mean and unit variance, i.e. feature X becomes $\frac{X-\mu}{\sigma}$ where μ is its mean and σ the variance.

7) Building a ML Model: Our goal is to predict the system's health status y_t at time t based on a new mea-

surement $\mathbf{x}_t$ and the preceding sequence of measurements $(\mathbf{x}_{t-1}, ..., \mathbf{x}_{t-\tau})$, assuming only the last τ measurements are relevant. Mathematically, we need to find a predictor $\mathbb{L}$ so that $\mathbb{L}(\mathbf{x}_t, ..., \mathbf{x}_{t-\tau}) = y_t$ for as many measurement points as possible. To classify such sequences, we use recurrent neural networks (RNNs), i.e. layers of cells that feed the weighted sum of their inputs into a nonlinear activation function, and output their signal to other cells. Connections to previous layers, or the same layer, are *recurrent* and essentially allow the network to "remember" input from previous time steps. RNNs are trained by descending on the gradient of their error function and updating their weights [7]. However, in the calculation of the gradient for the error function for recurrent networks, a lot of multiplications occur. Since for bounded activation functions, the individual gradients are well below one, the total gradient becomes very small and *vanishes*. To solve this *vanishing gradient problem*, we utilize long short-term memory (LSTM) cells as proposed by Hochreiter and Schmidhuber [8], which have a linear self connection that prevents the gradient from vanishing. This yields a nonlinear anomaly detection model that takes into account more relevant information than traditional monitoring techniques such as thresholds.

8) Discretizing the Time Series Data: To be able to use a wider range of classifiers on the time series data, we can create feature vectors $\mathbf{z}_t(\delta)$ for a sliding time window δ from the sequence $(\mathbf{x}_t, ..., \mathbf{x}_{t-\delta})$, and train the predictor for $\mathbb{L}(\mathbf{z}_t(\delta)) = y'_t(\delta)$. The aggregated label $y'_t(\delta)$ in that window is the label with the most frequent occurrence there. This allows the use of simpler models; Sun et al. [9] pursue this simplification using support vector machines to detect synthetic failures.

B. Optimizing Parameters

We need to choose a set of optimal hyperparameters for all of the learning algorithms we use in our experiment. Typical examples include learning rate and number of cells for neural network classifiers [7]. To search the hyperparameter space for the best values, we use four-fold cross validation [7], using all data for the validation part, not only the (undersampled) balanced subset.

III. Experiment

We can now refine the research questions and describe evaluation metrics, evaluation results and threats to validity.

A. Research Questions

We aim to detect anomalies and health issues of a software system using ML models built on operating data. We need to answer the following questions to evaluate how successful our solution is in predicting the health status of the system:

- **RQ1**: Can our classification models effectively serve as a predictor for the health status of a software system?
- **RQ2**: Are LSTM neural networks more effective in classifying operating data sequences than other classification algorithms?

These questions boil down to determining how "well" our classification works. We start by defining what "well" means.

B. Evaluation Metrics

Typical objectives in classification are precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$) [7]. In our case, a true positive (TP) means that a measurement for an anomaly is correctly recognized as such; a false positive (FP) is normal behaviour wrongly classified as anomaly. False negatives (FN) and true negatives (TN) are defined accordingly.

For classifiers which output a probability distribution over a set of classes, we need a threshold for class assignment to be able to evaluate the results. For instance, the default threshold value of $50\,\%$ means that an instance would be classified as an "anomaly", if for that instance the probability of being in anomaly class is more than $50\,\%$. We utilize the precision/recall (PRC) plot to compare different threshold values. PRC plots are more informative than receiver operating characteristic (ROC) plots in showing the performance of classifiers on unbalanced data [10]. In addition, we utilize the area under the PRC (AUPRC) metric as a numerical performance indicator to compare different curves.

C. Results of the Industrial Case Study

Our experiment data is collected from August 2014 to October 2015. We used $54.5\,\%$ of the original dataset (all data before May 2015) for training. $1.38\,\%$ of the training data belonged to the "anomaly" class; by undersampling the "healthy" class, we achieved a more balanced distribution with $37.7\,\%$ anomaly instances. In total, the training set consists of $1.5 \cdot 10^5$ data points.

Our cross validation results show that using two layers of 200 LSTM cells with the hyperparameters presented in Table II yields the best performance on the training dataset. Thus, we built the model using these settings and tested our model on the test dataset.

We compared the performance of the LSTM network with two other classifiers, namely naïve Bayes [11] and random forest [7]. Figure 1, Figure 2, and Figure 3 show the PRC plots for the LSTM network, naïve Bayes, and random forest respectively. The results are based on 1000 threshold steps.

The LSTM network had the highest performance of all classifiers with an AUPRC of 0.444. For reference, the corresponding area under ROC is as high as 0.892. At $50\,\%$ threshold , the LSTM network has $31.1\,\%$ precision and $69.5\,\%$ recall. The performance at that point is as good as random forest for fifteen minutes (if their threshold was adapted accordingly).

TABLE II
Hyperparameters for Best LSTM Network with Two Layers of 200 Cells Each, Using Ten Passes over the Training Set.

learning rate	0.0025
recurrent activation functions	tanh
output activation function	softmax
backpropagation length	600
L_2 norm constant	0.001
dropout regularization	40%
optimization algorithm	STGD with RMSPROP

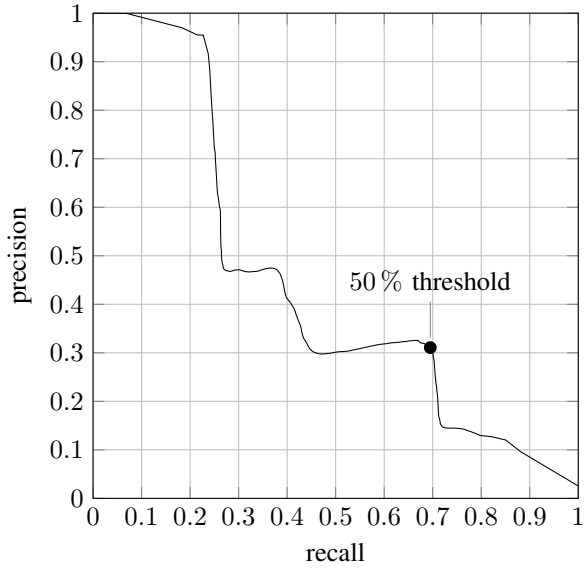

Fig. 1. PRC for best-performing LSTM network, threshold from 100 % (left) to 0 % (right).

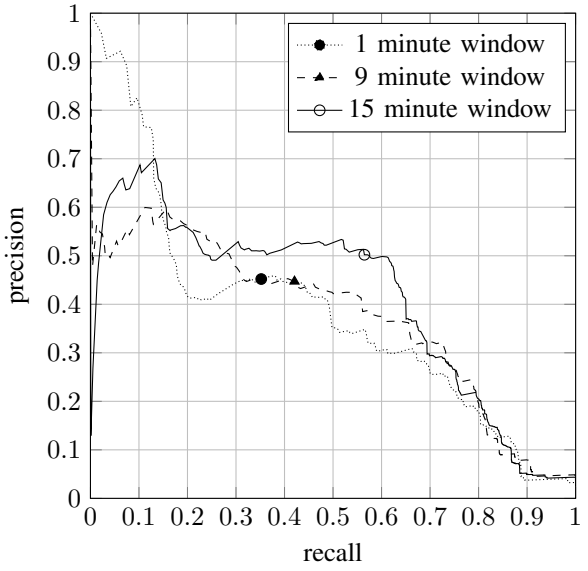

Fig. 3. PRC for random forest. Threshold ranges from 100 % (left) to 0 % (right), with markings at 50 %.

The naïve Bayes classifier has an AUPRC of 0.257 for the smallest time window and 0.334 for the largest. At the 50 % threshold, performance is similar for all the window sizes with about 19 % precision and 78 % recall. It is remarkable that performance is hardly affected by the window size in the high recall regions. In general, larger windows yield higher performance.

The random forest classifier indicates an AUPRC of 0.381 for a 1-minute window and 0.408 for the 15-minute window. This results in 50.2 % precision and 56.5 % recall (at 50 % threshold) for the 15-minute window setting.

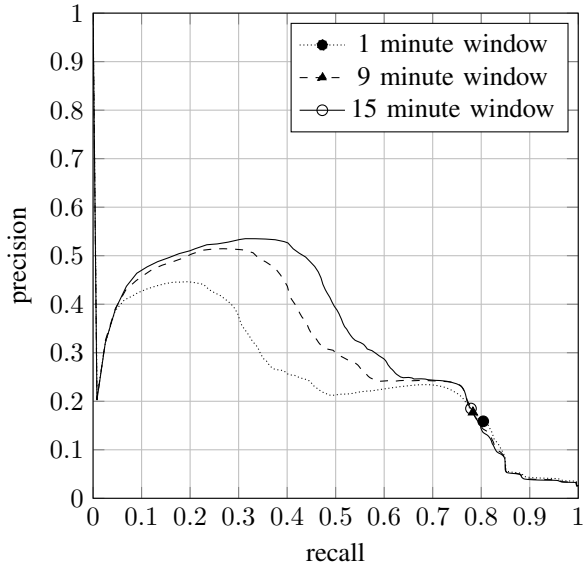

Fig. 2. PRC for naïve Bayes classifier. Threshold ranges from 100 % (left) to 0 % (right), with markings at 50 %.

Apart from the models discussed above, we also employed gradient-boosted trees and multi-layer perceptrons, but did not achieve good results. The multi-layer perceptron achieved at most 41.3 % precision and 37.8 % recall; gradient-boosted trees were the worst models with a maximum of 27.6 % precision and 16.6 % recall. Our reasoning is that due to the relatively high degree of error in the data labeling, only models that are robust to label noise could perform well. For instance, while gradient-boosted trees usually perform better than random forests on low-noise data, they are found to be much more sensitive to label noise [12, 13]. The naïve Bayes classifier, which performs reasonable in our test, is also known to be less affected by label noise [14].

Based on the above results, we answer the research questions as follows:

- **RQ1**: Although a precision of 30 % and 70 % recall (that we could achieve with multiple models) do not sound very high, the FP rate is actually only about 4 %. This very low rate of FPs makes the results promising for our industry partner, especially because in DevOps environments, restarts are not too expensive. Compared to manual inspection of metrics, utilizing our classification technique to detect about 70 % of the anomalies automatically, is a big step forward.
- **RQ2**: The best overall performance was achieved by LSTM networks with an AUPRC of 0.44. However, the figures show that LSTM networks only perform significantly better than the other classifiers when using higher thresholds. At lower thresholds, random forests are mostly better, and both models go through the 70 % recall, 30 % precision point.

D. Threats to Validity

We offered a methodology for adapting ML-based anomaly detection to an industrial context. We do not claim that our experiment setup and results will be valid for all other contexts. The first important factor that might impact the results is our data labeling. There might be some mistakes since we did not have any ground truth to compare the labeling results with. To mitigate the threats to the validity, we discussed our labeling and findings with experts.

In addition, since we use system restarts as indicators for problems, anomalies not severe enough to lead to a restart are not taken into account. This problem is inherent when using real-world data, and can hardly be mitigated.

Furthermore, it is possible that the set of features we used might not be ideal.

Finally, since anomaly detection is done only in retrospect, we ignore the back coupling of the operation decisions. When performing automated anomaly handling in practice, one would have to account for this, but it is likely that this can be solved programmatically.

IV. Related Work

In this paper, we focus on applying ML techniques in order to detect software operation anomalies, as opposed to static anomalies. Although ML is vastly used to solve various problems in many different fields, there is almost no work when it comes to applying ML techniques for detecting operation anomalies in distributed software systems. Thus, we discuss related literature in the field of intrusion detection in addition to already existing ML approaches for detecting anomalies for DevOps, and correlation-based anomaly detection.

Detecting anomalies on a predefined time interval (called window), for long DevOps operations, has been introduced in [9]. Support Vector Machines (SVMs) are being used to train multiple classifiers from monitored data, in a way that a SVM is created for each phase of the DevOps operation, on which the logging information can indicate the best suitable classifier at any time. For further anomaly detection improvement, the authors introduce moving average over the windows, and the entropy of metrics, as additional features to the SVM. Randomly injected faults can be accurately classified with larger time windows of three minutes and above. Yet the approach introduces several minute delays when the measurements can't be made more frequently than once per minute. The work focuses on SVMs and dedicates a lot of effort to anomaly detection during a rolling upgrade. In this paper, on the other hand, a wider range of different models are examined and put into comparison. Additionally, we employ sequence-based models that have the potential of avoiding the delay from larger time windows.

To assist in the reliable assurance of correct execution of cloud operations, an approach based on regression-based correlation analysis technique has been derived in [15]. The approach models the correlation between operations, like redeployment or scaling, and their effects on the resources of the underlying cloud platform. The correlation can be used as the basis for generating run-time assertions in order to detect anomalies in running operations. Operations are inferred from their event log messages, while resource usage is measured by metrics. Behaviour rules are extracted from the model and the predicted behaviour is compared with the actual behaviour; if they differ, an anomaly is assumed to have occurred. This approach performs well after injecting random, artificial faults on a initially trained healthy system. However, in this paper, we focus on anomaly detection for already faulty systems.

Intrusion detection is a research area that has been broadly studied for nearly 20 years, and is still a contemporary subject of widespread interest by researchers. Anomaly detection is closely related to intrusion detection, as disturbances of normal behaviour indicate a presence of intended or unintended induced attacks, faults or defects [16]. Since both research fields are very similar, there is an enormous potential in utilizing ML techniques for accurate anomaly detection. However, this also means that the problems are likely the same across these two fields. Kaur et al. [17] highlight the advantage of employing ML based techniques for detecting unknown anomalies, as well as known ones. Still, they describe the problem of separating normal from truly abnormal behaviour as *extremely challenging*. Omar et al. [16] are going a step further and describe the problem of distinguishing between normal and truly abnormal even as yet unsolved. Furthermore, they find that if the evaluation consists only of known attacks, supervised methods significantly outperform unsupervised techniques. This is in line with research in other problem domains that suggests that supervised methods outperform unsupervised approaches for classification problems in the majority of the time [18, 19, 20]. Hence, in this paper, we focus on supervised techniques for classification. While the intrusion detection research field enjoys great popularity, Sommer and Paxson [21] have analyzed that there is a wide discrepancy to the actual use of such systems in practice. This industrial evaluation aims to help close the gap and evaluate performance and challenges on a real-world application.

V. Conclusion and Future Work

Detecting operation anomalies for complex distributed software systems is quite a challenging task. The large amount of data ($7.5 \cdot 10^6$ data points), class imbalance of a mere $1.91\,\%$ anomalies, and prevalent label noise made it hard to find a high-performing classifier. Using LSTM networks to classify sequences, we achieved the highest area under precision-recall curve of 0.444. However, performance was only good at some parts of the curve; at a default $50\,\%$ threshold, we achieved $31.1\,\%$ precision and $69.5\,\%$ recall. When classifying single feature vectors from a sliding time window, larger windows did correspond to higher performance. For the largest window of fifteen minutes, the random forest classifier had an area under PRC of 0.408, which is slightly lower than the LSTM network. This corresponds to $50.2\,\%$ precision and $56.5\,\%$ recall at default threshold. The naïve Bayes classifier performed worse with an area under curve of 0.257; we also tested gradient-boosted trees and multilayer perceptrons, but did not discuss

performance in detail as it was significantly worse due to labeling noise.

In many applications, unnecessary node restarts are not too expensive, so low precision might be acceptable. But recall has to be very high to allow the use in a live system, since unrecognized anomalies can be expensive. As the precision-recall curves show, further increasing recall by lowering threshold is very costly to precision and thus not a suitable solution. Hence, the models we found in our research have to be combined with other detection mechanisms for a sufficient anomaly detection.

Of course, classification performance can still be improved in several ways. To better capture the process happening in a time window, different time window lengths could be used: Automated feature selection is found to increase prediction accuracy [22], and can yield more useful features on a feature set for multiple time window lengths. Additionally, one could utilize more sophisticated hyperparameter tuning (for example random search) as well as means to counteract label noise.

Another potential solution could be an interactive environment for developers (e.g., using a chat-bot), where the ML system proposes an assessment of the situation that can be confirmed or corrected by developers. This way, over time a more precise dataset is built. It is likely that this way, classification performance would increase steadily up to the point where completely autonomous error handling is possible at times when developers are not available, e.g., at night.

References

[1] F. Huch. Raw data repository. [Online]. Available: https://www.kaggle.com/anomalydetectionml/rawdata

[2] ——. Repository for feature data. [Online]. Available: https://www.kaggle.com/anomalydetectionml/features

[3] S. Moritz and T. Bartz-Beielstein, "imputets: Time series missing value imputation," *R package version 0.4*, 2015.

[4] G. M. Weiss and F. Provost, "The effect of class distribution on classifier learning: an empirical study," *Rutgers Univ*, 2001.

[5] N. V. Chawla, "Data mining for imbalanced datasets: An overview," in *Data mining and knowledge discovery handbook*. Springer, 2009, pp. 875–886.

[6] W. Wang, X. Zhang, S. Gombault, and S. J. Knapskog, "Attribute normalization in network intrusion detection," in *Pervasive systems, algorithms, and networks (ISPAN), 2009 10th international symposium on*. IEEE, 2009, pp. 448–453.

[7] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] D. Sun, M. Fu, L. Zhu, G. Li, and Q. Lu, "Non-intrusive anomaly detection with streaming performance metrics and logs for devops in public clouds: a case study in aws," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 2, pp. 278–289, 2016.

[10] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the roc plot when evaluating binary classifiers on imbalanced datasets," *PloS one*, vol. 10, no. 3, p. e0118432, 2015.

[11] I. Rish, "An empirical study of the naive bayes classifier," in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22. IBM, 2001, pp. 41–46.

[12] T. G. Dietterich, "An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization," *Machine learning*, vol. 40, no. 2, pp. 139–157, 2000.

[13] P. Melville, N. Shah, L. Mihalkova, and R. J. Mooney, "Experiments on ensembles with missing and noisy data," in *International Workshop on Multiple Classifier Systems*. Springer, 2004, pp. 293–302.

[14] D. F. Nettleton, A. Orriols-Puig, and A. Fornells, "A study of the effect of different types of noise on the precision of supervised learning techniques," *Artificial intelligence review*, vol. 33, no. 4, pp. 275–306, 2010.

[15] M. Farshchi, J.-G. Schneider, I. Weber, and J. Grundy, "Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 24–34.

[16] S. Omar, A. Ngadi, and H. H. Jebur, "Machine learning techniques for anomaly detection: an overview," *International Journal of Computer Applications*, vol. 79, no. 2, 2013.

[17] H. Kaur, G. Singh, and J. Minhas, "A review of machine learning based anomaly detection techniques," *International Journal of Computer Applications Technology and Research*, vol. 2, no. 2, pp. 185–187, 2013.

[18] L. Guerra, L. M. McGarry, V. Robles, C. Bielza, P. Larranaga, and R. Yuste, "Comparison between supervised and unsupervised classifications of neuronal cell types: a case study," *Dev Neurobiol*, vol. 71, no. 1, pp. 71–82, 2011.

[19] K. Lee, D. Booth, and P. Alam, "A comparison of supervised and unsupervised neural networks in predicting bankruptcy of korean firms," *Expert Systems with Applications*, vol. 29, no. 1, pp. 1–16, 2005.

[20] B. C. Love, "Comparing supervised and unsupervised category learning," *Psychonomic Bulletin & Review*, vol. 9, no. 4, pp. 829–835, 2002. [Online]. Available: https://doi.org/10.3758/BF03196342

[21] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 305–316.

[22] H. Osman, M. Ghafari, and O. Nierstrasz, "Automatic feature selection by regularization to improve bug prediction accuracy," in *Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Workshop on*. IEEE, 2017, pp. 27–32.