Kahles, Julen; Torronen, Juha; Huuhtanen, Timo; Jung, Alexander

Automating Root Cause Analysis via Machine Learning in Agile Software Testing Environments

Published in:
Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019

DOI:
10.1109/ICST.2019.00047

Published: 01/04/2019

Document Version
Peer reviewed version

Automating Root Cause Analysis via Machine Learning in Agile Software Testing Environments

Julen Kahles
R&D
Ericsson Finland
Jorvas, Finland
julen.kahles@ericsson.com

Juha Törrönen
R&D
Ericsson Finland
Jorvas, Finland
juha.torronen@ericsson.com

Timo Huuhtanen
Dept. of Computer Science
Aalto University
Espoo, Finland
timo.huuhtanen@aalto.fi

Alexander Jung
Dept. of Computer Science
Aalto University
Espoo, Finland
alex.jung@aalto.fi

***Abstract*—We apply machine learning to automate the root cause analysis in agile software testing environments. In particular, we extract relevant features from raw log data after interviewing testing engineers (human experts). Initial efforts are put into clustering the unlabeled data, and despite obtaining weak correlations between several clusters and failure root causes, the vagueness in the rest of the clusters leads to the consideration of labeling. A new round of interviews with the testing engineers leads to the definition of five ground-truth categories. Using manually labeled data, we train artificial neural networks that either classify the data or pre-process it for clustering. The resulting method achieves an accuracy of 88.9%. The methodology of this paper serves as a prototype or baseline approach for the extraction of expert knowledge and its adaptation to machine learning techniques for root cause analysis in agile environments.**

***Keywords*-root cause analysis, software testing, log data analysis, machine learning, artificial neural networks, classification, clustering, automation**

I. Introduction

In modern agile software development environments, continuous integration and deployment (CI/CD) require numerous tests in fast-paced sprints [1], [2], [3]. These generate vast amounts of raw diagnostics (log data) [4] which need to be analyzed by the testing engineers in order to retrieve the failed tests and track down their root causes [5], [6]. The process with which this is undertaken is known as root cause analysis (RCA) [7], and it is traditionally carried out in a manual fashion: testing engineers generally rely on their accumulated experience to check what they consider to be suspicious log files, often by querying for predefined keywords [8].

Given the vast amount of log data being produced at high speed and due to its unstructured formatting, abstruse and unintuitive for human readers, software-analysis-based RCA outperforms manual RCA in terms of speed and cost; human expert work is simply too slow or expensive [9].

Existing approaches to software-analysis-based RCA come in the form of defining a rule-based system whose behavior on the log files is fully enclosed by its coded specifications [8]. This solution does not scale well to large software projects which involve many conditions and statements that need to be explicitly programmed.

A more reasonable, effective, and economical way of tackling this problem can be achieved by making use of machine learning (ML), enabling the computer in question to learn from the data without constraining its behavior with strict rules [10]. As an extra benefit, an ML-based system might even find hidden patterns that are not initially taken into account by either manual analysis or a rule-based system.

A considerable amount of attention has been put into applying ML and statistical techniques to software testing [11], [12], [13], [14], [15], [16]. Focusing on log data analysis, researchers have studied different techniques.

In an early attempt, Andrews [17] presents a finite-state machine that analyzes the events that take place while a program is running, tracked in the produced log files. This automaton, however, does not "learn" in an ML sense, as the log analyzer simply states whether a given log file conforms to a specification or not.

With respect to pattern recognition, Weiss and Hirsh [18] develop an ML system with the name of *Timeweaver* that detects rare events in sequential data by leveraging genetic algorithms. It uses predictions of present events from past events to determine whether a data piece is normal or rare (depending on whether the prediction matches the realization or not, respectively). Additionally, Vaarandi [19] envisions a clustering algorithm that groups in each cluster a particular line pattern that occurs frequently. His log-word-frequency-based algorithm is released via the *Simple Logfile Clustering Tool*.

Regarding failure prediction, Fulp et al. [20] develop a support vector machine that is fed with the frequency representation of sequences of system log messages to predict software failure events. In his doctoral dissertation, Salfner [21] presents a continuous-time extension of hidden Markov models that recognizes symptomatic patterns of error sequences. Fronza et al. [22] offer a different approach to support vector machines: random indexing feeds them with sequences of operations extracted from log files.

Concerning failure prediction, Lee [23] presents his research efforts on the field of inductive learning, where he explores data mining techniques for building intrusion detection models. Xu et al. [24] apply principal component analysis (PCA) combined with term-weighting to parsed console logs.

Moreover, commercial applications have been developed for serving the need of log data analysis, like the highly successful *Splunk* or *Elasticsearch-Logstash-Kibana Stack*, among others [25], [26], [27], [28].

With regard to the topic of applying ML to the task of identifying the root causes of failed tests using log data analysis, previous research has framed this scenario as either an anomaly detection problem or an abnormal log detection issue.

Along these lines, Stearley [29] compares the performance of a bioinformatic-inspired algorithm, known as *TEIRESIAS* [30], and the previously envisioned *Simple Logfile Clustering Tool* by Vaarandi [19]: the former outperforms the latter in terms of detecting anomalies and investigating cause-effect hypotheses at the cost of a non-scalable system memory demand, where the latter is able to tackle longer log files (that exceed 10000 lines).

Recently, Du et al. [31] develop a deep neural network model for anomaly detection and diagnosis of system logs known as *DeepLog*. It is based on a long short-term memory recurrent neural network that learns patterns during normal execution (modeling the log files as natural language sequences) and detects anomalies when the obtained patterns differ from the learned ones.

Lastly, Debnath et al. [32] envision *LogLens*, an anomaly detection system that analyzes log files in real time. It works with minimal or even no target system knowledge and user specification: it learns normal behavior and builds a finite-state machine that captures it, with which it then detects anomalies.

Thus, we are not aware of previous work on analyzing clustering and classification algorithms for root cause analysis in agile software testing environments.

We address three main research questions:

1) What are the most relevant feature definitions of software testing log data?
2) What are useful root-cause categories?
3) How can the root cause of a failed test be identified using ML?

We provide a methodology for incorporating expert knowledge to ML techniques for RCA [1]. We tackle the design of the feature extraction process, the definition of ground-truth categories, and the assessment of ML algorithms.

[1]The results this research presents are obtained within the Master's thesis work of the first author [33].

This paper is organized as follows: Section II formulates RCA as an ML problem; Section III discusses the software testing environment this work is based on; In Section IV, we explain the methodology that has been followed while undertaking this research; Section V tackles the ML feature extraction process; Sections VI and VII present the envisioned ML clustering and classification proposals, respectively, and Section VIII describes their implementation details; Section IX gathers the obtained results; Finally, Section X concludes this paper.

II. Problem Formulation

ML systems act on a numerical representation of the raw data whose single elements are known as features [34]. These are data points from the problem in question, and they need to be defined and extracted from the raw input data. Then, for the i^{th} example to be treated (i.e., for each failed test case), a vector $\boldsymbol{x}$ is obtained whose elements store the numerical values associated with every envisioned feature:

$$\boldsymbol{x}^{(i)} = \left(x_1^{(i)}, \ldots, x_n^{(i)}\right) \in \mathbb{R}^n, \tag{1}$$

where n is the number of features.

A final feature matrix $\boldsymbol{X}$ is built by horizontally concatenating all feature vectors: each row corresponds to a failed test case, and each column stands for a single feature. The dimension of matrix $\boldsymbol{X}$ will be $(m \times n)$, where m is the number of examples.

The machine is expected to carry out some kind of prediction or hypothesis on the data $\boldsymbol{X}$ so that it generalizes well to new data. The hypothesis function $h(\boldsymbol{X})$ is dependent on the feature matrix $\boldsymbol{X}$ and on a weight vector $\boldsymbol{w}$ that controls the contribution of each feature in the final prediction.

The primary goal of an ML system is to find the values of the vector $\boldsymbol{w}$ that obtain the best prediction, and thus, achieve the best results. The system's performance is assessed by means of a loss function whose task is to assign a cost value to a prediction. This function is also dependent on the weight vector $\boldsymbol{w}$: minimizing it ensures obtaining the optimal weight values. In contrast with the parameters $\boldsymbol{w}$ the ML system optimizes, the ones that determine the architecture of the algorithm in question and are kept fixed during the optimization process are known as "hyperparameters" [35].

Regarding the pursued prediction, on the one hand, if no ground-truth values are available, the dataset is said to be unlabeled; clustering algorithms are usually used in the process of discovering a structure in unlabeled data by crafting groups in the feature space. On the other hand, when labels are available (gathered in vector $\boldsymbol{y}$ of size $(m \times 1)$), classification algorithms can be used that aim at matching the prediction to the pursued ground-truth values (a situation that is accounted for in the loss function).

In order to find the optimal weights $\boldsymbol{w}$, an ML algorithm starts by iterating on a dataset known as the "training set": the cost function is computed for different weights, and the values are optimized until the solution that minimizes the loss function is found. In order to make sure that the weights generalize well to other previously unseen datasets, a "validation set" is used for making certain that the cost function is not only minimized at the training stage. Finally, once the optimal solution is found on both sets, a "test set" is used for reporting the final performance of the algorithm as it has not been used for enhancing the algorithm's performance.

One way to deal with the obstacle of "overfitting" the optimization to the training set consists in making use of regularization, a process that adds an extra term to the cost function such that it smoothens the final prediction: the added term penalizes the loss function with respect to its complexity, forcing the learning process to achieve simpler results on the training set.

The sub-branch of ML known as deep learning (DL) deals with algorithms that can be represented as cascaded connections of neurons [35], [36]. These entities are called artificial neural networks (ANNs), and their processing units receive the name of "neurons". The k^{th} neuron in a network can be represented in the form of a block diagram, as can be seen in Figure 1. It is fed with a set of inputs x_{k1} to x_{kp} that get scaled by their intrinsic weight values w_{k1} to w_{kp}. These are added together, where a fixed bias term b_k is also included in the summation. The final value is scaled to a given range, usually $[0, 1]$ or $[-1, 1]$, by applying a nonlinear activation function $\phi(\cdot)$:

$$y_k = \varphi\left(\sum_{j=1}^{p} w_{kj} x_{kj} + b_k\right) \tag{2}$$

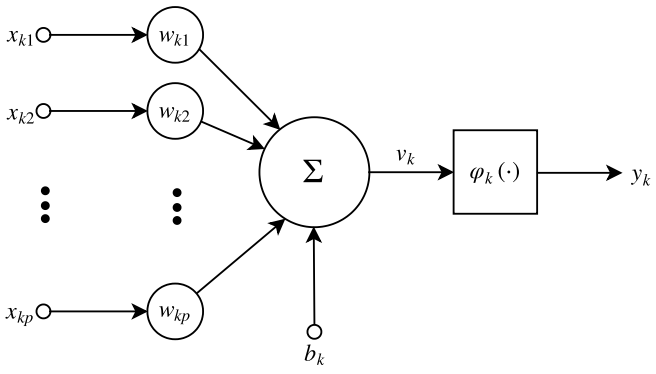


Figure 1. Block diagram of a neuron. Figure adapted from [36].

The general structure of ANNs consists of an input layer, an output layer, and a variable number of hidden layers that determines the "depth" of the ANN: deeper ANNs generally achieve more complex overall input-to-output mappings than shallower ones.

Specific to ANNs, the techniques of dropout and batch normalization can be used as regularizers. The former consists of the deactivation of several neurons at random throughout the training process, which makes it harder for the ANN in question to overfit, as communication between all neurons is blocked at certain moments, and thus, the network is less likely to achieve a more established structure from the data. The latter corresponds to standardizing the neurons' inputs in the hidden layer to be zero mean and unit variance; this results in a slight regularization effect due to the values of the inputs being bounded. Given the unsuitability of both methods acting simultaneously [37], they will be tried out separately in this work.

III. Testing Environment

This work is based on the development and testing environments of an agile team at Ericsson Finland: the former follows the agile methodology via the extreme programming framework and containerizes the produced software; the latter runs tests on the produced code (unit, integration, system, and regression tests). These are grouped in suites and are executed in dedicated servers that communicate employing the Hypertext Transfer Protocol (HTTP).

Once functional code has been produced, the developers commit it to a repository, from where it is both tested and taken to a manual inspection tool. The former runs automated unit and integration tests, whose outputs consist of a binary result (either pass or fail) and a collection of test logs that relate to their evolution (tracking, among others, variable values, function calls, inputs, and outputs). Additionally, the servers provide analytics on their HTTP communication during the tests' runs in the form of additional log data. The latter serves as a review environment, where other developers provide feedback for potential corrections. If any issue is encountered at any of these two stages (meaning that any test fails or that the code assessors require corrections), the author of the code is notified, and the process of RCA takes place, where the author works on retrieving the source of the problem. Once the code is approved, meaning that all tests pass and no corrections are required from the manual inspection tool, it is pushed to the main branch.

This iterative process is displayed in Figure 2 and corresponds to the commit cycle that takes place every time new code is committed. Additionally, daily regression tests and nightly system tests are run as to ensure the software works flawlessly in the final main branch.

The pace at which software is developed in a CI/CD environment causes a large number of tests to be carried out in short time spans, which, as a consequence, generate a continually growing amount of log data. This fact motivates the automation of the RCA process for failed test cases.

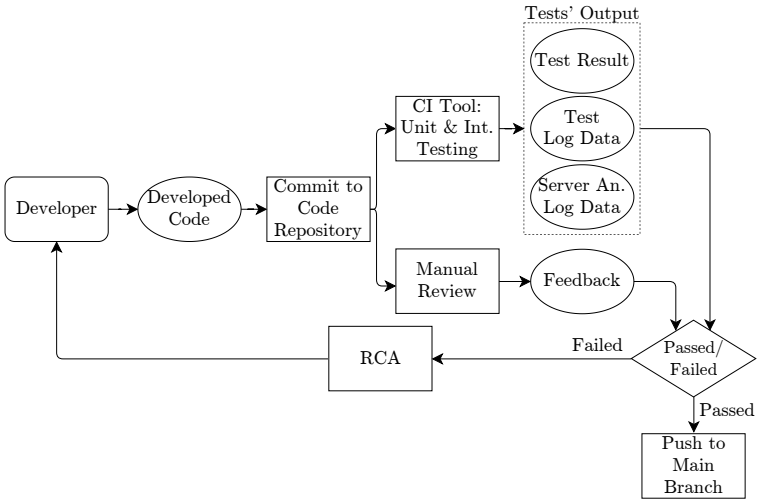


Figure 2. Commit cycle of the agile team of Ericsson Finland.

IV. Methodology

We gather log data for a period of 6 months, and we conceptualize a collection of features by interviewing the testing engineers. The features serve as the numerical data upon which the ML algorithms act.

The obtained test cases are then split into train, validation, and test sets at random, following a 60%/20%/20% division. 20 different random splits are formed with the aforementioned ratios so as to avoid overfitting to a given form of partitioning of the data. All algorithms are computed 20 times with these set divisions, and the average values and standard deviations of the evaluation metrics are reported (cf. Section IX).

Once this is achieved, several clustering algorithms are applied to the obtained feature matrix with a varying number of clusters, aiming at discovering a structure from the data: the cost in resources and time of obtaining labels justifies the initial consideration of clustering algorithms. Even though weak correlations are obtained between specific clusters and the failure causes they represent, the meaning of the remaining clusters remains vague.

Seeing that no clear structure is initially found from the data, the testing engineers are asked to conceptualize the highest-level failure root causes of the failed tests, an action that was initially attempted to be avoided, given its cost in time and resources. Five root-cause categories are thus obtained, and the failed test cases are manually labeled according to these categories:

- Functional errors, where the error lies in the product's developed code (i.e., development errors).
- Connectivity errors, whose failure corresponds to a problem in the communication between the servers.
- Infrastructure errors, being the failure caused by the testing environment (e.g., hardware errors).
- Software test code errors, corresponding to failures caused by the presence of bugs in the tests' code.
- Purposely interrupted tests, which gather tests whose execution was stopped by the testing engineers before completion.

The purpose of this is twofold: on the one hand, the results obtained from the clustering algorithms can now be quantitatively assessed, and on the other hand, classification methods can also be considered for this task.

V. Feature Extraction

We gather log data from February 2018 to July 2018, collecting a total amount of 1271 failed test cases. From these, a collection of features needs to be conceptualized and extracted, which will be later provided to the implemented ML solutions.

In order to extract features, the testing engineers are interviewed to gain insight into their analysis methodology, and eventually, map their expert knowledge into a feature matrix. After having explored the debugging process of several testing engineers, one could see how the pass/fail ratios and their occurrence across suites seemed to correlate with the origin of the failures. Additionally, the responses that the servers provided to the containers' requests and the number of times the containers were invoked gave hints on why certain tests failed. These findings lead to a conceptual categorization of four feature groups that best summarize the evolution and result of a test, as assessed by the testing engineers. These contain numerical values, and correspond to:

- Container activity (count of entries): number of times a given container is invoked.
- Server analytics (count of entries): HTPP 5xx responses, errors, tracebacks, and warnings.
- Success rate per test suite.

- Overall test success.

Table I displays the number of features in each group, as well as the scaling that is applied to each category. The success rates are obtained by dividing the number of passed tests in a given group by the total number of run tests in said group (either a given suite or the total amount of tests). Regarding the container invocations and server analytics, for each feature, the count of entries for each failed test case is divided by the maximum count of entries over all failed test cases (intra-feature normalization).

The normalized features have their values bounded between 0 and 1 to ensure an equal contribution of each feature to the ML algorithms and to increase the speed in the convergence of their optimization [38], [34].

A total number of 188 features are obtained from the aforementioned 1271 test cases. Nonetheless, the developed feature extraction software is programmed to retrieve the used containers and suites, and hence, depending on the analyzed failed test cases, their number (and thus, the amount of features in these two groups) is subjected to change. Their count is therefore marked with an asterisk (*) in Table I.

Table I
INFORMATION ON THE EXTRACTED FEATURES

Feature group	Number of features	Feature scaling
Container invocations (normalized count of entries)	56*	Intra-feature normalization
Server analytics (count of entries)	4	Intra-feature normalization
Success rate per test suite	127*	Ratio (passed/total)
Global success rate	1	Ratio (passed/total)

VI. Clustering Analysis

Clustering algorithms can be subdivided into hard clustering algorithms, which assign one and only one cluster to each example, and soft clustering algorithms, where the probability value of belonging to a cluster is given to each example.

Both types will be assessed in this work, the first via the k-means algorithm, the latter by means of Gaussian mixture models (GMMs); these algorithms correspond to prime examples of their corresponding categories [34]. The k-means algorithm is simpler in its implementation, a fact that comes at the cost of achieving rather strict boundaries between clusters in the feature space, whereas GMMs are more flexible and might find more adaptive nonlinear clusters.

Both algorithms are fed with several variations of the original feature matrix:

- Raw data, i.e., the feature matrix without any further alteration.
- Data preprocessed with PCA, a statistical technique that reduces the dimensionality of the original feature matrix by retaining as much variation in the original data as possible [39], [34]. The number of principal components (i.e., the new dimension of the data) is selected so that it retains 95% of the original variance on the training set.
- Data preprocessed by an autoencoder (AE), an ANN that encodes and decodes the original data, constrained to reconstruct the data to be as close as possible to the original representation. AEs can generalize the action of PCA to nonlinear mappings that outperform it [40], [35]. The encoded representation of the original feature matrix is thus fed to the clustering algorithms.

For the data preprocessed by an AE, two different ANN variations are considered: a standard AE, and a recent proposal by Song et al. [41] which includes the original reconstruction action and, at the same time, aims at grouping the encoded representation in clusters. For the latter, the contribution of the second term in the custom loss function is controlled utilizing parameter η.

VII. Classification

Once the data is labeled by the testing engineers, multilayer perceptron (MLP) ANNs are used for classification: their input layer consists of as many neurons as features, the output layer contains as many neurons as conceptual categories are pursued, and the number of hidden layers, as well as the number of neurons in each hidden layer, determines their design.

The final layer applies the softmax activation function, causing the output neurons to store, for each input example, values bounded in the $[0, 1]$ range, where their sum adds up to 1. The results can be then interpreted as the probability the ANN assigns to each example of belonging to a given category. The conceptual category of the output neuron with the highest value is then selected as the predicted class. The objective of the ANN is to adjust the weights of the neurons so that the classification maximizes the matching between the predicted and the ground-truth categories.

VIII. Tested algorithms

Both clustering algorithms are initialized ten times and iterated at maximum 300 times, should they not converge before (where the convergence is defined as no changes happening between successive assignments). The best result from those ten initializations is kept in terms of the pursued loss function (so that a suboptimal local minimum is avoided).

Given that we end up dealing with labeled data, and the number of pursued root cause categories is 5, the number of clusters is set to be $k = 5$.

Table II
HYPERPARAMETER TUNING FOR THE STANDARD AE: ITEMIZED NUMBER OF RUNS.

Hyperparameters	Number of runs
Loss functions	1
Clustering algorithms	2
Network structures	28
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**15120**

Table III
HYPERPARAMETER TUNING FOR THE CUSTOM-LOSS AE: ITEMIZED NUMBER OF RUNS.

Hyperparameters	Number of runs
Loss functions	1
Weighing parameter η values	10
Clustering algorithms	2
Network structures	28
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**151200**

The AEs' optimal hyperparameters are found via grid search: all combinations of architectures with a set of defined hyperparameters are assessed, and the best-performing combination of hyperparameters is selected. The number of tested hyperparameters is chosen as a compromise between completeness and runtime: increasing the number of hyperparameters has a multiplicative effect on the number of runs to be carried out.

For both the standard and custom-loss AE, the following hyperparameters are assessed:

- Loss functions: Standard AE and Song et al.'s loss functions, respectively.
- Clustering algorithms: k-means and GMMs.
- Network structures: 28 different network structures, consisting of the symmetric permutations of 1 to 7 encoder and 1 to 7 decoder layers, whose number of neurons are exponents of base 2, where the exponents range from 1 to 7.
- Optimizers: Stochastic gradient descent (SGD) and Adam.
- Activation functions: Rectified linear unit (ReLU), sigmoid, and hyperbolic tangent (tanh).
- Dropout of neurons with ratios of 0.3, 0.5, 0.8, and no dropout. Batch normalization is tested for when no dropout takes place, as well as no batch normalization.
- Regularizer functions: L1 and L2 regularization, each with 4 regularization values (0.01, 0.1, 1, 10), as well as no regularization.

Table IV
HYPERPARAMETER TUNING FOR THE CLASSIFICATION MLP: ITEMIZED NUMBER OF RUNS.

Hyperparameters	Number of runs
Loss functions	1
Network structures	55
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**14850**

Table V
HYPERPARAMETER TUNING FOR THE CLASSIFICATION MLP (EXTENSIVE ANALYSIS WITH 7 HIDDEN LAYERS): ITEMIZED NUMBER OF RUNS.

Hyperparameters	Number of runs
Loss functions	1
Network structures	6435
Optimizers	2
Activation functions	3
Dropout or batch norm.	5
Regularizer functions	2
Regularizer values	4
Total regularization combinations	$(4 \times 2) + 1 = 9$
Total count	**1737450**

For Song et al.'s proposal, the contribution of the clustering term in the custom loss function is evaluated with values of the parameter η of -1, -0.7, -0.5, -0.2, -0.1, 0.1, 0.2, 0.5, 0.7, and 1.

The counts of all different AE hyperparameter possibilities are gathered in Tables II and III for the standard AE and the custom-loss AE proposal, respectively. The best results for each case are kept.

With respect to classification MLPs, the same hyperparameters are assessed but for the network structures: this time, up to 10 layers are tested, so that the number of neurons ranges all the permutations of powers of 2 whose exponents range from 1 to 10, yielding 55 different architectures. The count of hyperparameter possibilities is shown in Table IV.

Given that the MLPs provide the best results with an ANN consisting of 7 hidden layers, a more in-depth grid search is carried out: all permutations of values in decreasing order starting from 1000 neurons to 20 neurons with decrements of 70 are assessed over 7 hidden layers, meaning that 6435 structures are tested. The new count of hyperparameter possibilities can be seen in Table V.

The vast amount of runs were executed for 27 days on a dedicated machine, yielding the results gathered in Section IX.

IX. Results

As mentioned in Section IV, the manually obtained labels are needed for quantifying the performance of the ML algorithms.

For the classification algorithms, the simple measure of accuracy returns a ratio of correct assignments with respect to all processed examples [42]:

$$\text{accuracy}\,(\Omega, \mathbb{C}) = \frac{1}{K}\sum_{i=1}^{k} |\omega_i \cap c_i|, \tag{3}$$

where $\mathbb{C} = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots, \boldsymbol{c}_k\}$ is the set of ground-truth classes, $\Omega = \{\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \ldots, \boldsymbol{\omega}_k\}$ is the set of predicted classes, k is the number of distinct ground-truth categories, and K is the total number of processed samples. Its value will be bounded between 0 and 1, the former being a result of total mismatch, and the latter being the consequence of a perfect match.

For clustering algorithms, the mutual information (MI) is a more suitable metric. It measures the agreement or amount of information that the prediction and ground-truth assignments share, ignoring permutations [43].

The mutual information between two random variables (RVs) A and B is defined as [44]:

$$\text{MI}\,(A, B) = \sum_{i=1}^{N}\sum_{j=1}^{N} \text{P}\,(a_i, b_j) \log\left(\frac{\text{P}\,(a_i, b_j)}{\text{P}\,(a_i)\,\text{P}\,(b_j)}\right), \tag{4}$$

where RV A conceptualizes the assignment for each element of the original dataset $\mathbb{D}$ to the ground-truth categories belonging to $\mathbb{C}$, and RV B refers to the matching from the original dataset $\mathbb{D}$ to the predicted categories Ω.

It represents the measure of mutual dependence between both RVs, i.e., how much information one RV conveys about the other one: the higher its value, the higher the dependence between the prediction and the ground-truth assignments.

The values of MI increase in proportion to the number of clusters and samples, a fact that can be accounted for with the adjusted MI (AMI) [45], [43]:

$$\text{AMI}\ \ (A, B) = \frac{\text{MI}\ \ (A, B) - \ \text{E}\ \ [\,\text{MI}\ \ (A, B)]}{\max\left(\text{H}\,(A), \text{H}\,(B)\right) - \ \text{E}\ \ [\,\text{MI}\ \ (A, B)]}, \tag{5}$$

where E is the expectation operator, and H is the information or Shannon entropy of an RV V, which states the average surprise or uncertainty when sampling V, and is defined as:

$$\text{H}\,(V) = \text{E}\,[\text{I}\,(V)] = \ \text{E}\ \left[\log\left(\frac{1}{\text{P}\,(V)}\right)\right] \tag{6}$$

$$= \ \text{E}\ \left[-\log\left(\text{P}\,(V)\right)\right] \tag{7}$$

$$= -\sum_{i=1}^{n} \text{P}\,(v_i) \log\left(\text{P}\,(v_i)\right). \tag{8}$$

When the probability distribution of the RV V is close to uniform, the entropy is maximized (having a value close to 1) and the uncertainty when sampling the RV is high. When the probability distribution of V tends to be deterministic, the entropy is minimized (having a value close to 0) and the uncertainty when sampling V is low.

The AMI extends its validity to classification [46], and thus, when carrying out grid searches over the ANN structures presented in Section VI, the best results in terms of AMI are collected for each category of tested algorithms. Additionally, the values of accuracy are included in the classification algorithms given their intuitive meaning.

The ANNs' parametrization is gathered in Table VI and their associated visual structure can be found in Figures 3 and 4 for clustering and classification, respectively (where all the layers are depicted as rectangles with constant width and variable height, proportional to the number of neurons, which is shown as text inside the rectangles).

Concerning clustering, the best AEs display shallow architectures: but for the custom-loss case, where the number of hidden layers is equal to 3, the best AEs used in conjunction with k-means and GMMs have only one hidden layer, the former with 128 and the latter with 32 neurons. The dropout rate in both scenarios is 0.5; hence, no batch normalization is used. The AE that combines k-means with Song et al.'s custom loss has three hidden layers of sizes 64, 32 and 64, respectively, and it neither regularizes with dropout nor implements batch normalization. The optimizers and activation functions for all AEs are Adam and ReLU, respectively.

Regarding classification, the best base-2 architecture has seven hidden layers, and the extended architecture tested the same number of hidden layers but with different counts of neurons per layer (cf. Section VI). But for batch normalization, which the base-2 architecture does not implement (while the extended architecture does), all other parameters are shared in both networks: ReLU as an activation function, no dropout, and SGD as an optimizer.

The performance of the tested algorithms is shown in Tables VII and VIII.

For clustering, applying PCA to the raw data does not show any improvement with respect to clustering it directly with both k-means and GMMs. The results with GMMs are in fact worse than those obtained with k-means.

With the best AE, k-means still outperforms GMMs. There is a slight improvement when using k-means and a greater one when using GMMs with respect to no pre-processing or to pre-processing with PCA.

Using the AE with Song et al.'s custom loss function, the results reach the maximum average AMI value of 0.451, thus proving to be the most suitable solution for clustering.

However, classification MLPs significantly outperform clustering analysis, reaching an average accuracy value of 0.880 with the base-2 architecture and of 0.889 with the

extended architecture (where the associated average AMI values are 0.654 and 0.665, respectively).

X. Conclusion

In this paper, ML clustering and classification algorithms have been explored for accurately categorizing the root causes of failed tests from log data.

We found out that the most feasible way to extract meaningful features from the log data is to interview the testing engineers and follow their debugging activity. We then obtain four distinct conceptual feature groups that summarize the information the testing engineers make use of when dealing with manual RCA. Given the satisfactory results obtained with the classification MLPs, we conclude that the envisioned feature groups adequately transfer the testing engineers' knowledge to a feature matrix.

In order to map the existing human experts' knowledge into distinctive ground-truth root-cause categories, we carried out further interviews with the testing engineers: we asked them to frame the conceptual ground-truth categories they envision when dealing with failed tests. These interviews prove to be simpler than the ones that pursue crafting the feature matrix: the sought information is not latent, as the concept of "ground truth category" is used in their everyday debugging process.

Lastly, concerning the use of ML for accurately identifying the root causes of the failed tests, this work experimentally shows how MLP-based classification proves to be the best-performing solution, significantly surpassing the action of clustering analysis (no matter how intricate its foundation): the best ANN correctly assigns an example with its failure root cause, on average, 88.9% of the time. While a manual revision of the obtained results is needed and full automation is not achieved at this stage, the testing engineers benefit from this as their activity changes from manually analyzing the root causes of all failed tests to reviewing the output obtained from the MLP and checking whether it actually maps to the real root cause for each test, correcting the 11 out of every 100 failed tests that are misclassified.

Future research directions aimed at improving the obtained results (which might make the machine's output less dependent on an *a posteriori* human validation) could contemplate the following:

- A more extensive hyperparameter optimization could be run by utilizing a parallelized implementation running on several machines, searching over a broader range of hyperparameters.
- More granular ground-truth failure categories could be considered, taking as labels more specific categories than the highest-level ones. For this to be achieved, the testing engineers would need to be interviewed more extensively, and a diagram with their envisioned ground-truth categories would need to be constructed.
- The extraction of new features might improve the algorithms' outcome, whose envisioning could be carried out by means of extensive interviewing of the testing engineers (for example, including code-change-derived features); in fact, we did some trials with features related to CPU usage in the servers that run the tests, but they were feasible only in some cases, hence why we could not include them in this work.
- The use of natural language processing could be tested for mining the log files directly and obtaining a representation in a new feature space, which could then be used for classification, as presented by Bertero et al. [47]. Additionally, more research could be carried out in this area with the aim of finding more intricate proposals, and their performance could be compared to traditional classification with a set of defined features, as has been carried out in this research.

Acknowledgment

This work has been funded by Ericsson Finland.

Table VI
BEST-PERFORMING ANN ARCHITECTURES.

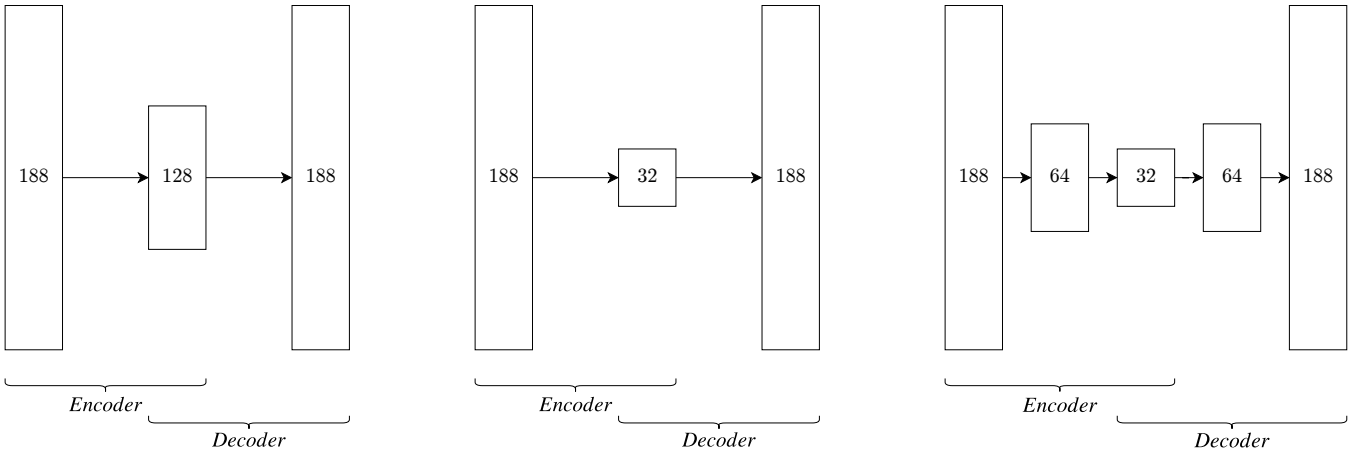
	Number of hidden layers	Structure of hidden layers	Activation functions in hidden layers	Dropout rate	Batch normalization	Optimizer
AE: k-means	1	[128]	ReLU	0.5	No	Adam
AE: GMMs	1	[32]	ReLU	0.5	No	Adam
AE: k-means (custom loss)	3	[64, 32, 64]	ReLU	0	No	Adam
MLP: base-2 architecture	7	[512, 256, 128, 64, 32, 16, 8]	ReLU	0	No	SGD
MLP: 7-layer architecture	7	[1000, 930, 860, 790, 230, 160, 90]	ReLU	0	Yes	SGD

Table VII
CLUSTERING AMI RESULTS.

	k-means	GMMs
Raw data	0.401 ± 0.050	0.295 ± 0.037
PCA (95% of original variance)	0.400 ± 0.052	0.290 ± 0.038
Clustering (via the standard AE)	0.415 ± 0.031	0.405 ± 0.036
Clustering (via the custom-loss AE)	0.451 ± 0.041	—

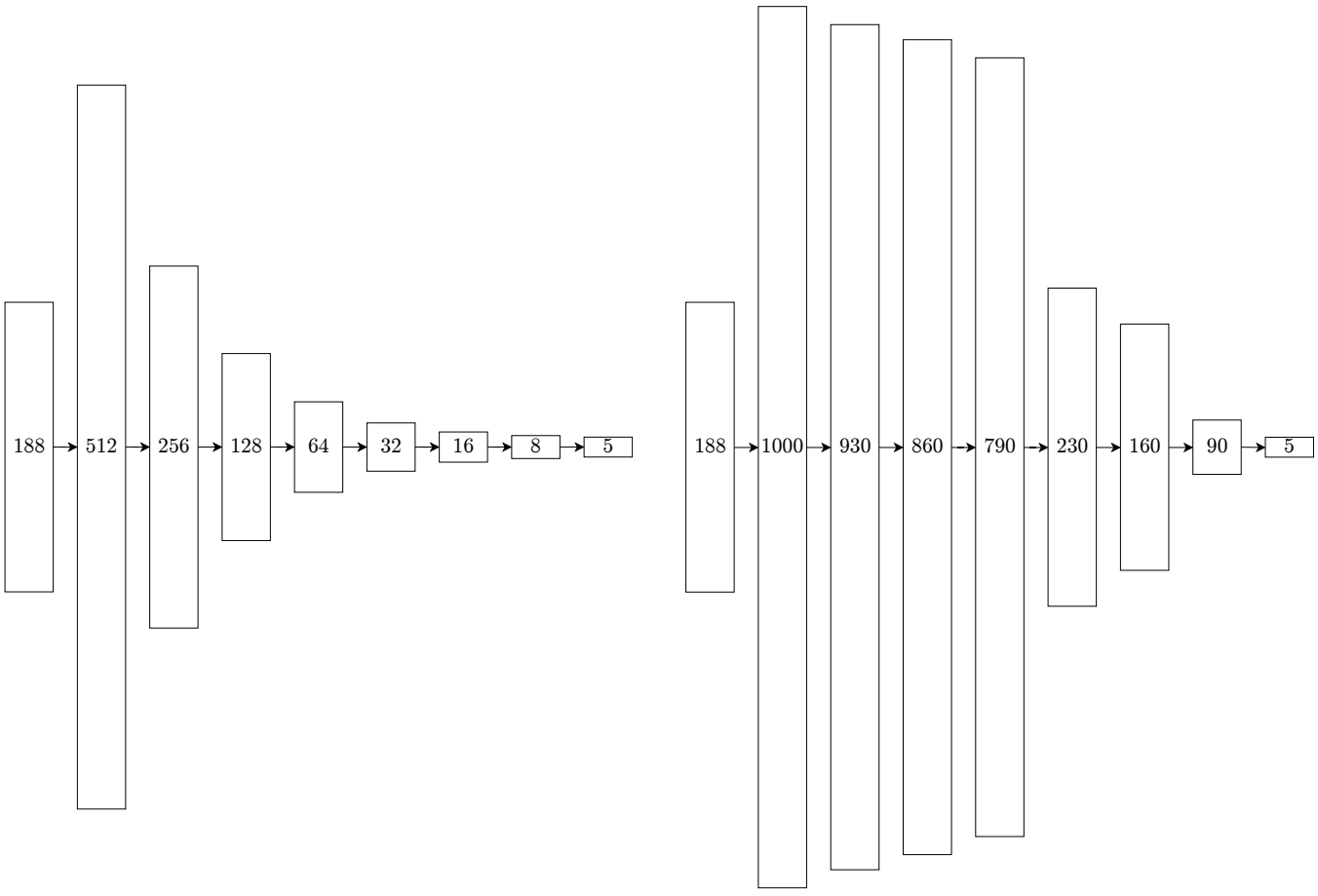
Table VIII
CLASSIFICATION AMI AND ACCURACY RESULTS.

	AMI	Accuracy
Base-2 architecture	0.654 ± 0.043	0.880 ± 0.021
7-layer architecture	0.665 ± 0.043	0.889 ± 0.020



(a) AE: k-means. (b) AE: GMMs. (c) AE: k-means (custom loss).

Figure 3. AE architectures for clustering.



(a) MLP: base-2 architecture. (b) MLP: 7-layer extended architecture.

Figure 4. MLP architectures for classification.

REFERENCES

[1] Amazon Web Services, "What is Continuous Integration?" https://aws.amazon.com/devops/continuous-integration/, accessed: 2019-01-16.

[2] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, 1st ed., ser. Addison-Wesley Signature Series. Boston, MA, USA: Pearson Education, July 2007.

[3] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909–3943, March 2017.

[4] E. Dustin, J. Rashka, and J. Paul, *Automated Software Testing: Introduction, Management, and Performance*, 1st ed. Reading, MA, USA: Addison-Wesley Professional, July 1999.

[5] J. J. Rooney and L. N. V. Heuvel, "Root Cause Analysis for Beginners," *Quality Progress*, vol. 37, no. 7, pp. 45–53, July 2004.

[6] R. J. Latino, K. C. Latino, and M. A. Latino, *Root Cause Analysis: Improving Performance for Bottom Line Results*, 4th ed. Boca Raton, FL, USA: CRC Press, July 2011.

[7] T. O. Lehtinen, M. V. Mäntylä, and J. Vanhanen, "Development and evaluation of a lightweight root cause analysis method (ARCA method) – Field studies at four software companies," *Information and Software Technology*, vol. 53, no. 10, pp. 1045–1061, October 2011.

[8] W. Li, "Automatic Log Analysis using Machine Learning: Awesome Automatic Log Analysis version 2.0," Master's thesis, Uppsala University, Uppsala, Sweden, November 2013.

[9] G. Tassey, "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards and Technology, RTI Project Number 7007.011, Research Triangle Park, NC, Tech. Rep., May 2002, available at: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf. Accessed: 2019-01-16.

[10] T. M. Mitchell, *Machine Learning*, 1st ed., ser. Computer Science. New York, NY, USA: McGraw-Hill Education, March 1997.

[11] A. Denise, M.-C. Gaudel, and S.-D. Gouraud, "A Generic Method for Statistical Testing," in *Proceedings of the 15^{th} International Symposium on Software Reliability Engineering (ISSRE 04)*. Saint-Malo, France: IEEE Computer Society, November 2004, pp. 25–34.

[12] A. X. Zheng, "Statistical Software Debugging," Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, USA, December 2005.

[13] N. Baskiotis, M. Sebag, M.-C. Gaudel, and S.-D. Gouraud, "A Machine Learning Approach for Statistical Software Testing," in *Proceedings of the 20^{th} International Joint Conference on Artificial Intelligence (IJCAI 07)*. Hyderabad, India: International Joint Conferences on Artificial Intelligence, January 2007, pp. 2274–2279.

[14] L. C. Briand, "Novel Applications of Machine Learning in Software Testing," in *Proceedings of the 8^{th} International Conference on Quality Software (QSIC 08)*. Oxford, UK: IEEE, August 2008, pp. 3–10.

[15] M. Noorian, E. Bagheri, and W. Du, "Machine Learning-based Software Testing: Towards a Classification Framework," in *Proceedings of the 23^{rd} International Conference on Software Engineering and Knowledge Engineering (SEKE 11)*. Miami Beach, FL, USA: Knowledge Systems Institute, July 2011, pp. 225–229.

[16] J. Kim, J. W. Ryu, H.-J. Shin, and J.-H. Song, "Machine Learning Frameworks for Automated Software Testing Tools: A Study," *International Journal of Contents*, vol. 13, no. 1, pp. 38–44, March 2017.

[17] J. H. Andrews, "Testing using Log File Analysis: Tools, Methods, and Issues," in *Proceedings of the 13^{th} Annual International Conference on Automated Software Engineering (ASE 98)*. Honolulu, HI, USA: IEEE, October 1998, pp. 157–166.

[18] G. M. Weiss and H. Hirsh, "Learning to predict rare events in event sequences," in *Proceedings of the 4^{th} International Conference on Knowledge Discovery and Data Mining (KDD 98)*. New York, NY, USA: American Association for Artificial Intelligence, August 1998, pp. 359–363.

[19] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns From Event Logs," in *Proceedings of the 3^{rd} IEEE Workshop on IP Operations and Management (IPOM 2003)*. Kansas City, MO, USA: IEEE, October 2003, pp. 119–126.

[20] E. W. Fulp, G. A. Fink, and J. N. Haack, "Predicting Computer System Failures Using Support Vector Machines," in *Proceedings of the USENIX Workshop on the Analysis of System Logs (WASL 08)*. San Diego, CA, USA: USENIX, December 2008, pp. 5–5.

[21] F. Salfner, "Event-based Failure Prediction: An Extended Hidden Markov Model Approach," Ph.D. dissertation, Humboldt University of Berlin, Berlin, Germany, February 2008.

[22] I. Fronza, A. Sillitti, G. Succi, M. Terho, and J. Vlasenko, "Failure prediction based on log files using Random Indexing and Support Vector Machines," *Journal of Systems and Software*, vol. 86, no. 1, pp. 2–11, January 2013.

[23] W. Lee, "Applying Data Mining to Intrusion Detection: the Quest for Automation, Efficiency, and Credibility," *SIGKDD Explorations*, vol. 4, no. 2, pp. 35–42, December 2002.

[24] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," in *Proceedings of the 22^{nd} ACM Symposium on Operating Systems Principles (SOSP 09)*. Big Sky, MT, USA: ACM, October 2009, pp. 117–132.

[25] Alex Zhitnitsky, "Splunk vs ELK: The Log Management Tools Decision Making Guide," available at: https://blog.takipi.com/splunk-vs-elk-the-log-management-tools-decision-making-guide/, accessed: 2019-01-16.

[26] Splunk Inc., "Splunk Inc. Announces Fiscal First Quarter 2019 Financial Results," available at: https://www.splunk.com/en_us/newsroom/press-releases/2018/splunk-inc-announces-fiscal-first-quarter-2019-financial-results.html, accessed: 2019-01-16.

[27] J. Stearley, S. Corwell, and K. Lord, "Bridging the Gaps: Joining Information Sources with Splunk," in *Proceedings of the USENIX Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML 10)*. Vancouver, BC, Canada: USENIX, October 2010, pp. 8–8.

[28] Elasticsearch BV, "Elasticsearch Grows to 6 Million Downloads, Adds Former Box SVP as CMO," available at: https://www.elastic.co/fr/blog/press/elasticsearch-grows-6-million-downloads-adds-former-box-svp-cmo, accessed: 2019-01-16.

[29] J. Stearley, "Towards Informatic Analysis of Syslogs," in *Proceedings of the 5^{th} IEEE International Conference on Cluster Computing (CLUSTER 2004)*. San Diego, CA, USA: IEEE, September 2004, pp. 309–318.

[30] I. Rigoutsos and A. Floratos, "Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm," *Bioinformatics*, vol. 14, no. 1, pp. 55–67, February 1998.

[31] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *Proceedings of the 24^{th} ACM SIGSAC Conference on Computer and Communications Security (CCS 17)*. Dallas, TX, USA: ACM, November 2017, pp. 1285–1298.

[32] B. Debnath, M. Solaimani, M. A. G. Gulzar, N. Arora, C. Lumezanu, J. Xu, B. Zong, H. Zhang, G. Jiang, and L. Khan, "LogLens: A Real-time Log Analysis System," in *Proceedings of the 38^{th} IEEE International Conference on Distributed Computing Systems (ICDCS 2018)*. Vienna, Austria: IEEE, July 2018, pp. 1052–1062.

[33] J. Kahles, "Applying Machine Learning to Root Cause Analysis in Agile CI/CD Software Testing Environments," Master's thesis, Aalto University, Espoo, Finland, February 2019.

[34] A. Jung, "Machine Learning: Basic Principles," arXiv preprint, available at: https://arxiv.org/abs/1805.05052v8, October 2018, accessed: 2019-01-16.

[35] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, 1st ed. Cambridge, MA, USA: MIT Press, November 2016, available at: http://www.deeplearningbook.org. Accessed: 2019-01-16.

[36] S. O. Haykin, *Neural Networks and Learning Machines*, 3rd ed. Upper Saddle River, NJ, USA: Pearson, November 2009.

[37] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift," arXiv preprint, available at: https://arxiv.org/abs/1801.05134v1, January 2018, accessed: 2019-01-16.

[38] S. Raschka, "About Feature Scaling and Normalization (and the effect of standardization for Machine Learning algorithms)," available at: https://sebastianraschka.com/Articles/2014_about_feature_scaling.html, accessed: 2019-01-16.

[39] I. Jolliffe, *Principal Component Analysis*, 2nd ed., ser. Springer Series in Statistics. New York, NY, USA: Springer Science+Business Media, October 2002.

[40] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, no. 5786, pp. 504–507, July 2006.

[41] C. Song, F. Liu, Y. Huang, L. Wang, and T. Tan, "Auto-encoder Based Data Clustering," in *Proceedings of the 18^{th} Iberoamerican Congress on Pattern Recognition (CIARP 2013)*. Havana, Cuba: Springer, November 2013, pp. 117–124.

[42] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, 1st ed. Cambridge, UK: Cambridge University Press, July 2008.

[43] Scikit-learn, "Clustering performance evaluation," available at: http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation, accessed: 2019-01-16.

[44] K. Murphy and F. Bach, *Machine Learning: A Probabilistic Perspective*, 1st ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, August 2012.

[45] N. X. Vinh, J. Epps, and J. Bailey, "Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance," *Journal of Machine Learning Research*, vol. 11, pp. 2837–2854, October 2010.

[46] H. Bao-Gang, H. Ran, and Y. Xiao-Tong, "Information-theoretic Measures for Objective Evaluation of Classifications," *Acta Automatica Sinica*, vol. 38, no. 7, pp. 1169–1182, July 2012.

[47] C. Bertero, M. Roy, C. Sauvanaud, and G. Trédan, "Experience Report: Log Mining Using Natural Language Processing and Application to Anomaly Detection," in *Proceedings of the 28^{th} International Symposium on Software Reliability Engineering (ISSRE 2017)*. Toulouse, France: IEEE, October 2017, pp. 351–360.