Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH PAPER

Filesystem, Infrastructure Provisioning, and User Accounts: Operations in Defective Infrastructure as Code Scripts

Akond Rahman | Laurie Williams

[1]Department of Computer Science, North Carolina State University, North Carolina, USA

Correspondence
*Akond Rahman. Email: aarahman@ncsu.edu

Summary

Defects in infrastructure as code (IaC) scripts can have dire consequences, for example, creating large-scale system outages. One approach to mitigate defects in IaC scripts is to categorize the operations that appear in defective scripts, which may help practitioners to efficiently allocate validation and verification efforts. *The goal of this paper is to help practitioners prioritize validation and verification efforts for infrastructure as code (IaC) scripts by identifying the characteristics of defective IaC scripts*. We investigate if text features can be used to identify operations that characterize defective IaC scripts. We use two text mining techniques to extract text features from IaC scripts: the bag-of-words technique, and the term frequency-inverse document frequency (TF-IDF) technique. Using the extracted features and applying grounded theory, we characterize defective IaC scripts. We also use the text features to build defect prediction models. We apply our methodology on 2,439 IaC scripts downloaded from four organizations namely, Mirantis, Mozilla, Openstack, and Wikimedia Commons. We identify three properties that characterize defective IaC scripts: filesystem operations, infrastructure provisioning, and managing user accounts. Our constructed defect prediction models using text features yielded a median F-measure of 0.72, 0.64, 0.74, and 0.72, respectively, for Mirantis. Mozilla, Openstack, and Wikimedia Commons. Based on our findings, we advocate practitioners to allocate sufficient validation and verification efforts for IaC scripts which include any of the following operations: filesystem operations, infrastructure provisioning, or managing user accounts.

KEYWORDS:
configuration as code, configuration scripts, continuous deployment, continuous delivery, devops, empirical study, infrastructure as code, puppet, text mining

1 | INTRODUCTION

Continuous deployment (CD) is a software development methodology that helps information technology (IT) organizations to deploy software rapidly [49]. In CD, configurations and infrastructure specifics of the deployment environment is treated as code in form of scripts, known as infrastructure as code (IaC) scripts [24] [41]. IT organizations, such as Netflix [1], Ambit Energy [2], and Wikimedia Commons [3], use IaC scripts to automatically manage their software dependencies, and construct automated deployment pipelines [41] [24]. Commercial IaC tools, such as Ansible [4]

[1]https://www.netflix.com/
[2]https://www.ambitenergy.com/
[3]https://commons.wikimedia.org/wiki/Main_Page
[4]https://www.ansible.com/

and Puppet [5], provide multiple utilities to construct automated deployment pipelines. Use of IaC scripts has helped IT organizations to increase their deployment frequency. For example, Ambit Energy, used IaC scripts to increase their deployment frequency by a factor of 1,200 [6].

However, while developing IaC scripts practitioners may inadvertently introduce defects in IaC scripts [41]. Defects in IaC scripts can have serious consequences, for example, in January 2017, execution of a defective IaC script erased home directories of ∼270 users in cloud instances maintained by Wikimedia Commons [7]. Through qualitative and quantitative analysis we can identify operations that characterize defective IaC scripts. Such characterization can help IT organizations to efficiently allocate their validation and verification efforts by focusing on scripts that contain the identified operations.

The goal of this paper is to help practitioners prioritize validation and verification efforts for infrastructure as code (IaC) scripts by identifying the characteristics of defective IaC scripts.

We answer the following research questions:

RQ-1: What operations characterize defective infrastructure as code (IaC) scripts? How frequently do the identified characteristics appear in IaC scripts?

RQ-2: How can we build prediction models for defective infrastructure as code scripts using text features?

We characterize defective IaC scripts by extracting text features. We use two text mining techniques to extract text features: the 'bag-of-words (BOW)' technique [20] and the 'term frequency-inverse document frequency (TF-IDF)' technique [33]. We apply feature selection [58] on the extracted features using principal component analysis (PCA) to account for collinearity and identify text features that are correlated with defective IaC scripts. We apply the Strauss-Corbin Grounded Theory (SGT) [57] on text features that correlate with defective IaC scripts to characterize properties of defective IaC scripts. We quantify the count of each identified property that appear in IaC scripts. We construct defect prediction models using the text features and four statistical learners namely, Classification and Regression Tree, Logistic Regression, Naive Bayes, and Random Forest. We also tune the parameters of the four learners and the constructed text feature matrices to achieve better prediction performance. We evaluate the performance of the constructed prediction models using six metrics: precision; recall; accuracy; G-mean; area under the receiver operating characteristic curve (AUC); and F-Measure. We evaluate our methodology using 2,439 IaC scripts collected from four IT organizations: Mirantis, Mozilla, Openstack, and Wikimedia Commons.

We list our contributions as following:

- A list of operations that characterizes defective IaC scripts; and
- Defect prediction models that predict defective IaC scripts

In this paper, we extend our prior work titled "Characterizing Defective Configuration Scripts Used for Continuous Deployment", which was published at the International Conference on Software Testing, Verification and Validation (ICST)' 2018 [48]. The additional technical contributions of our extended work is summarized as following:

- Application of our methodology to a new dataset, the Mirantis dataset;
- In addition to Random Forest, use of three other learners namely Classification and Regression Trees, Logistic Regression, and Naive Bayes to construct defect prediction models;
- In addition to AUC and F-measure use of four other measures namely, precision, recall, accuracy and G-mean, to evaluate the constructed defect prediction models.
- Evaluation of how tuning of text feature matrices impact performance of the constructed prediction models.

We organize rest of the paper as following: in Section 2 we provide necessary background and related work. In Section 3 we describe our methodology. In Section 4 we describe our constructed datasets. We use Sections 5 and 6 respectively, to report our empirical findings, and discuss our findings. We present the limitations of our paper in Section 7. We finally conclude the paper in Section 8.

2 | BACKGROUND AND RELATED WORK

We provide necessary background information and describe related work in this section.

[5] https://puppet.com/
[6] https://puppet.com/resources/case-study/ambit-energy
[7] https://wikitech.wikimedia.org/wiki/Incident_documentation/20170118-Labs

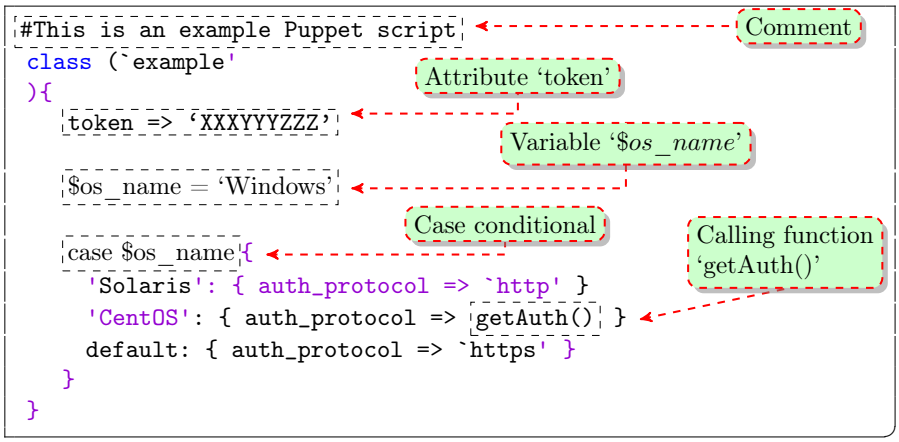


FIGURE 1 Annotation of an example Puppet script.

2.1 | Background

In this section, we provide background information on IaC and prediction performance measures to evaluate defect prediction models.

2.1.1 | Infrastructure as Code (IaC)

IaC is the practice of automatically defining and managing network and system configurations, and infrastructure through source code [24]. Companies widely use commercial tools such as Puppet, to implement the practice of IaC [24] [27] [53]. We use Puppet scripts to construct our dataset because Puppet is considered one of the most popular tools for configuration management [27] [53], and has been used by companies since 2005 [35]. Puppet provide several language constructs to automate configurations, system administration tasks, and manage package dependencies. For example, Puppet provides the 'user' construct to automatically create and manage user accounts. As another example, Puppet provides the 'sshkey' construct to manage SSH keys.

Typical entities of Puppet include modules and manifests [30]. A module is a collection of manifests. Manifests are written as scripts that use a .pp extension. Puppet provides the utility 'class' that can be used as a placeholder for the specified variables and attributes, which are used to specify configuration values. Similar to general purpose programming languages, code constructs such as functions/methods, comments, and conditional statements are also available for Puppet scripts. For better understanding, we provide a sample Puppet script with annotations in Figure 1.

2.1.2 | Performance Measures to Evaluate Defect Prediction Models

We use six measures to evaluate prediction performance of the constructed models:

- **Precision**: Precision measures the proportion of IaC scripts that are actually defective given that the model predicts as defective. We use Equation 1 to calculate precision.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

- **Recall**: Recall measures the proportion of defective IaC scripts that are correctly predicted by the prediction model. We use Equation 2 to calculate recall.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

- **Area Under The Receiver Operating Characteristic Curve (AUC)**: AUC uses the receiver operating characteristic (ROC). ROC is a two-dimensional curve that plots the true positive rates against false positive rates. An ideal prediction model's ROC curve has an area of 1.0. A random prediction's ROC curve has an area of 0.5. We refer to the *a*rea *u*nder the ROC curve as AUC throughout the paper. We consider AUC as this measure is threshold independent unlike precision and recall [17], and recommended by prior research [32].

- **F-Measure**: F-Measure is the harmonic mean of precision and recall. Increase in precision, often decreases recall, and vice-versa [36]. F-Measure provides a composite score of precision and recall, and is high when both precision and recall is high.

- **Accuracy**: We calculate accuracy using Equation 3. In Equation 3, TP, TN, P, and N respectively presents true positives, true negatives, number of defective scripts, and number of neutral scripts.

$$Accuracy = \frac{TP + TN}{P + N} \tag{3}$$

- **G-Mean**: We use Equation 4 to calculate G-Mean.

$$\textit{G-Mean} = \sqrt{(precision * recall)} \tag{4}$$

2.2 | Related Work

Our paper is related to empirical studies that have focused on IaC technologies, such as Puppet. Sharma et al. [54] investigated smells in IaC scripts and proposed 13 implementation and 11 design smells. Jiang and Adams [27] investigated the co-evolution of IaC scripts and other software artifacts, such as build files and source code. They reported changes to IaC scripts are tightly coupled with changes to test and source code files. Weiss et al. [61] proposed and evaluated 'Tortoise', a tool that automatically corrects erroneous configurations in IaC scripts. Hummer at al. [25] proposed a framework to enable automated testing of IaC scripts. Bent et al. [10] proposed and validated nine metrics to detect maintainability issues in IaC scripts. Rahman et el. [46] investigated which factors influence usage of IaC tools. Rahman et al. [47] investigated the questions that programmers ask on Stack Overflow to identify the potential challenges programmers face while working with Puppet. In another work, Rahman et al. [45] identified seven types of security smells that are indicative of security weaknesses in IaC scripts. They identified 21,201 occurrences of security smells that include 1,326 occurrences of hard-coded passwords. Rahman and Williams [48] characterized operations that appear in defective scripts, for example, setting up user account, file system operations and infrastructure provisioning.

Prior research has used text mining techniques such as BOW and TF-IDF to characterize and predict security and non-security defects. Scandariato et al. [52] mined text features from 20 Android applications and used the mined text features to predict vulnerabilities that appear in these applications. Walden et al. [60] applied the BOW-based text mining technique on web application source code to predict if the web applications contain vulnerabilities. They observed that, for their selection of web applications, text mining works better for vulnerability prediction than that of static code metrics. Perl et al. [43] applied the BOW model on commit messages extracted from version control repositories to predict security defects in 66 Github projects. Hovsepyan et al. [23] mined text features from source code of 18 versions of a mobile application, and used the text features to predict vulnerable files. They reported an average precision and recall of 0.85 and 0.88. Mizuno et al. [38] extracted patterns of tokens from source code using spam filter technique to predict fault prone modules for two projects: argUML and eclipse BIRT. Hata et al. [21] mined text features on source code from 10 releases of five projects using the spam filter technique. They observed with logistic regression, text-based features outperform source code metrics with respect to building fault prediction models. Gegick et al. [16] used text mining techniques upon the bug reports' content to predict if a bug report is related to security or not. Aversano et al. [3] applied text mining to transform code changes into vector spaces, and use the extracted vector spaces to predict which code changes introduce defects.

The above-mentioned studies demonstrate the usefulness of text features in software defect analysis. We take motivation from the above-mentioned research studies and extract text features via text mining techniques. We use the text features to characterize defective IaC scripts.

3 | METHODOLOGY

We first provide definitions, then we describe our methodology.

- **Defect**: An imperfection that needs to be replaced or repaired [26].
- **Defect-related commit**: A commit which includes an IaC script and whose message indicates that an action was taken related to a defect.
- **Defective script**: An IaC script which is listed in a defect-related commit.
- **Neutral script**: An IaC script with no reported defects.

3.1 | Dataset Construction

Our methodology of dataset construction involves three steps: repository collection (Section 3.1.1), commit message processing (Section 3.1.2), and determining defect-related commits (Section 3.1.3).

3.1.1 | Repository Collection

Researchers [59] [15], in prior work on defect prediction, used datasets from public software data archives such as Tera-PROMISE and NASA. But, these datasets are derived from OOP-based systems [40] [15], and not from IaC scripts. Thus, we need to construct IaC script-specific datasets to evaluate our methodology and build prediction models. We use open source repositories to construct our datasets. An open source repository contains valuable information about the development process of an open source project, but the project might have a short development period [39]. This observation motivates us to identify repositories for mining by using these inclusion criteria:

- **Criteria-1**: The repository must be available for download.
- **Criteria-2**: At least 11% of the files belonging to the repository must be IaC scripts. Jiang and Adams [27] reported that in open source repositories IaC scripts co-exist with other types of files, such as Makefiles and source code files. They observed a median of 11.1% of the files to be IaC scripts. By using a cutoff of 11%, we assume to collect a set of repositories that contain a sufficient amount of IaC scripts for analysis.
- **Criteria-3**: The repository must have at least two commits per month. Munaiah et al. [39] used the threshold of at least two commits per month to determine which repositories are active with respect to development activity. We use this threshold to filter repositories that are inactive with respect to development activity.

3.1.2 | Commit Message Processing

Prior research (e.g. [51], [62]) leveraged open source repositories that use version control systems (VCS) for defect prediction studies. We use two artifacts from VCS of the selected repositories from Section 3.1.1, to construct our datasets: (i) commits that indicate modification of IaC scripts; and (ii) issue reports that are linked with the commits. We use commits because commits contain information on how and why a file was changed. Commits can also include links to issue reports. Issue report summaries may provide more insights on why IaC scripts were changed in addition to what is found in commit messages. We collect commits and other relevant information as following:

- First, we extract commits that were used to modify at least one IaC script. A commit lists the changes made on one or multiple files [1].
- Second, we extract the message of the commit. A commit includes a message, commonly referred as a commit message. The commit messages indicate why the changes were made to the corresponding files [1].
- Third, if the commit message included a unique identifier that maps the commit to an issue in the issue tracking system, we extract the identifier and use that identifier to extract the summary of the issue. We use regular expression to extract the issue identifier. We use the corresponding issue tracking API to extract the summary of the issue; and
- Fourth, we combine the commit message with any existing issue summary to construct the message for analysis. We refer to the combined message as 'extended commit message (XCM)' throughout the rest of the paper. We use the extracted XCMs to separate the defect-related commits from the non defect-related commits, as described in 3.1.3.

3.1.3 | Determining Defect-related Commits

We use defect-related commits to identify the defective IaC scripts, and the metrics that characterizes defective IaC scripts. We apply qualitative analysis to determine which commits were defect-related commits. Qualitative analysis provides the opportunity to improve the quality of the constructed dataset [22]. We perform qualitative analysis using the following three steps:

Categorization Phase: At least two raters with software engineering experience determine which of the collected commits are defect-related. We adopt this approach to mitigate the subjectivity introduced by a single rater. Each rater classifies an extended commit message (XCM) as defect-related if the XCM represents an imperfection in an IaC script. We provide raters with an electronic handbook on IaC scripts [30], and the IEEE publication on anomaly classification [26]. We record agreement between raters and compute the Cohen's Kappa [9] score, for all XCMs.

Resolution Phase: Raters can disagree if a XCM is defect-related. In these cases, we use an additional rater's opinion to resolve such disagreements. We refer to the additional rater as the 'resolver'.

Practitioner Agreement: To evaluate the ratings of the raters in the categorization and the resolution phase, we randomly select 50 XCMs for each dataset, and contact practitioners. We ask the practitioners if they agree to our categorization of XCMs. High agreement between the raters'

categorization and programmers' feedback is an indication of how well the raters performed. The percentage of XCMs to which practitioners agreed upon recorded and the Cohen's Kappa score should be computed. Practitioners do not provide any input to the raters' classifications.

Upon completion of these three steps, we can classify which commits and XCMs are defect-related. From the defect-related commits, we determine which IaC scripts are defective, similar to prior work [62]. Defect-related commits list which IaC scripts were changed, and from this list we determine which IaC scripts are defective. From the defective and neutral scripts we extract text features using two steps: text preprocessing and text feature extraction, respectively discussed in Section 3.1.4 and Section 3.1.5.

3.1.4 | Text Preprocessing

We apply text pre-processing in the following steps:

- First, we remove comments from scripts.
- Second, we split the extracted tokens according to naming conventions: camel case, pascal case, and underscore. These split tokens might include numeric literals and symbols, so we remove these numeric literals and symbols. We also remove stop words.
- Finally, we apply Porter stemming [44] on the collected tokens. After completing the text preprocessing step, we collect a set of pre-processed tokens for each IaC script in each dataset.

3.1.5 | Text Feature Extraction

We use two text mining techniques to extract text features: 'bag-of-words (BOW)' [20] and 'term frequency-inverse document frequency (TF-IDF)' [33]. The BOW technique which has been extensively used in software engineering (e.g. [43] [60] [23]), converts each IaC script in the dataset to a set of words or tokens, along with their frequencies. Similar to BOW, the TF-IDF technique is also popular in software engineering (e.g. [13] [12]), and prior research has demonstrated that TF-IDF can help in building better prediction models [33].

Bag-of-Words: Using the BOW technique, we use the tokens extracted from Section 3.1.4. We compute the occurrences of tokens for each script. By using the occurrences of tokens we construct a feature vector. Finally, for all the scripts in the dataset, we construct a feature matrix.

ScriptA	*ScriptB*
build, git, include, template	build, dir, file, include, os

⇩

	Feature Vector <build, dir, file, git, include, os, template>
ScriptA	<1, 0, 0, 1, 1, 0, 1>
ScriptB	<1, 1, 1, 0, 1, 1, 0>

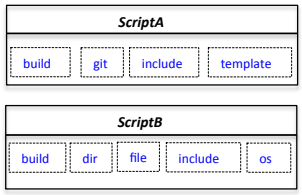
FIGURE 2 A hypothetical example to illustrate the BOW technique 3.1.5.

We use a hypothetical example shown in Figure 2 to illustrate the BOW technique. In our hypothetical example, our dataset has two IaC scripts ScriptA and ScriptB that respectively contain four and five pre-processed tokens. From the occurrences of tokens, we construct a feature vector where the the token 'build' appears once for ScriptA and ScriptB.

TF-IDF: The TF-IDF technique computes the relative frequency of a token compared to other tokens, across all documents [33]. In our experimental setting, the tokens that we apply TF-IDF on, are extracted from IaC scripts, as discussed in Section 3.1.4. The documents are the IaC scripts from which we extracted the tokens. For a script s, and a token t, we calculate the TF-IDF as following:

- **Calculate TF**: We calculate TF of token t in script s using Equation 5:

$$TF(t, s) = \frac{\text{occurences of token } t \text{ in script } s}{\text{total count of tokens in script } s} \quad (5)$$



a

	ScriptA	ScriptB
build	0.25	0.20
dir	0.00	0.20
file	0.00	0.20
git	0.25	0.00
include	0.25	0.20
os	0.00	0.20
template	0.25	0.00

b

Token	IDF
build	0.00
dir	0.30
file	0.30
git	0.30
include	0.00
os	0.30
template	0.30

c

	ScriptA	ScriptB
build	0.00	0.00
dir	0.00	0.06
file	0.00	0.06
git	0.07	0.00
include	0.00	0.00
os	0.00	0.06
template	0.07	0.00

d

FIGURE 3 A hypothetical example to illustrate out feature vectorization technique using TF-IDF. Figure 3a presents the pre-processed tokens of two scripts: ScriptA and ScriptB. Figure 3b presents the TF values for both scripts. Figure 3c presents the IDF scores for the unique seven tokens. Finally, TF-IDF scores for each script and token is presented in Figure 3d.

- **Calculate IDF**: We calculate IDF of token t using Equation 6:

$$\text{IDF(t)} = \log_{10}\left(\frac{\text{total count of scripts in the dataset}}{\text{count of scripts in which token } t \text{ appears at least once}}\right) \quad (6)$$

- **Calculate TF-IDF**: We calculate TF-IDF of token t using Equation 7:

$$TF - IDF(t, s) = TF(t, s) * IDF(t) \quad (7)$$

We use a hypothetical example to illustrate how the TF-IDF vectorization process works as shown in Figure 3. In our hypothetical example, our dataset has two IaC scripts ScriptA and ScriptB that respectively contain four and five pre-processed tokens. The total unique tokens in our hypothetical dataset is seven because two tokens appear in both scripts. Using Equation 5, we calculate the TF metric for each of these tokens and for both scripts: ScriptA and ScriptB. For example, the TF metric for token 'template' and ScriptA is 0.25, as the token 'template' appears once in ScriptA, and the total count of tokens in ScriptA is 4. Next, we show the calculation of metric IDF for all tokens using Equation 6. For the token 'template' we observe IDF to be 0.3, as it appears in one of the two scripts in our hypothetical dataset. Finally, using Equation 7, we determine the TF-IDF scores for token 'template'. For ScriptA and ScriptB token 'template' has a TF-IDF score of 0.07, and 0.0, respectively.

Upon completion of this step, we create a feature vector for each script in the dataset for both techniques: BOW and TF-IDF.

3.1.6 | Feature Selection

Feature selection is the process of eliminating features that have minimal influence on prediction performance [18] [58]. All of the extracted text features using the BOW and TF-IDF techniques may not be correlated with defective IaC scripts and may not contribute in building defect prediction models. The text features that have minimal correlation with defective IaC scripts can be eliminated via feature selection. We use principal component analysis (PCA) [58] for feature selection because PCA accounts for multi-collinearity amongst features [58] and identifies the strongest patterns in the data [58]. PCA creates independent linear combinations of the features that account for most of the co-variation of the features. PCA also provides a list of components and the amount of variance explained by each component. These principal components are independent and do not correlate or confound each other. For feature selection, we compute the total amount of variance accounted by the PCA analysis to determine what text features should be used for building prediction models. We select the count of principal components that account for at least 95% of the total variance to avoid overfitting. The identified components include text features that correlate with defective scripts.

We use the identified components using PCA analysis to answer RQ-1 and RQ-2. As will be described in Section 3.2, to answer RQ-1, we apply qualitative analysis on the text features that correlate with defective IaC scripts. Next, as will be described in Section 3.3 to answer RQ-2, we use the identified principal components as input to statistical learners for building defect prediction models.

3.2 | RQ-1: What operations characterize defective infrastructure as code (IaC) scripts? How frequently do the identified characteristics appear in IaC scripts?

The text features included in the identified principal components from Section 3.1.6 are correlated with defective IaC scripts. However, these text features are tokens, which might be insufficient to produce actionable information for practitioners. We address this issue by applying qualitative analysis on the identified tokens. We apply a qualitative analysis called Strauss-Corbin Grounded Theory (SGT) [57]. SGT is a variant of Grounded

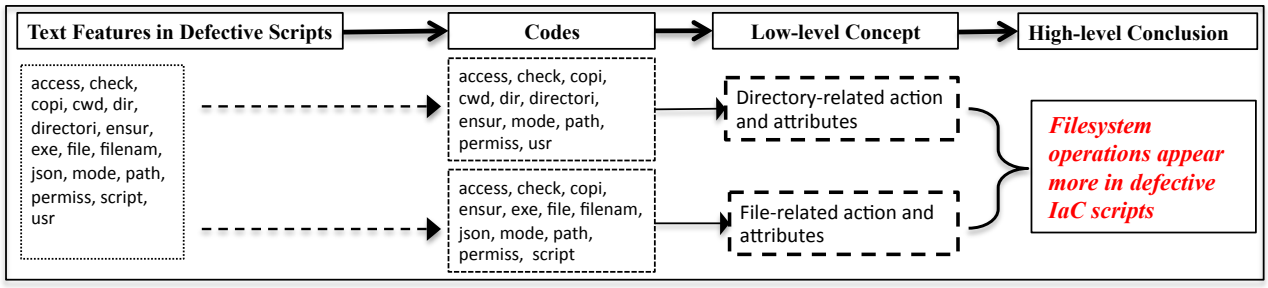


FIGURE 4 An example of how we use Strauss-Corbin Grounded Theory (SGT) to characterize defective IaC scripts.

Theory (GT) [57] [55] that allows for specification of research questions and is used to characterize properties from textual artifacts [57] [55]. SGT includes three elements: 'codes', 'low-level concept', and 'high-level conclusion'. In SGT, a 'high-level conclusion' represents an attribute or property [57], and by deriving these high-level conclusions, we identify properties that characterize defective scripts.

We use an example in Figure 4 to explain how we use the three SGT elements. We first start with text features that characterize defective IaC scripts determined by our PCA analysis to derive necessary codes. These codes are formed using text features that share a common attribute. As shown in Figure 4, using the commonality amongst attributes we construct two codes: one code is related to directories, and the other code is related to files. Next, we generate low-level concepts from the codes by creating a higher level of abstraction than text features. For example, the low-level concept 'directory-related action and attributes' was determined by the 11 tokens identified as codes. The final step is to draw high-level conclusions by identifying similarities between the low-level concepts. In the example, the two low-level concepts are related to performing filesystem operations. We use these two low-level concepts to determine a high-level conclusion 'Filesystem operations appear more in defective IaC scripts'. This high-level conclusion identifies one of the operations that characterize defective IaC scripts.

The second part of RQ1 is focused on the frequency of the identified properties that characterize defective scripts. By quantifying the frequency of the identified properties, we can identify how many scripts can be prioritized for verification and validation using that particular property. We determine the frequency of each property by counting for how many scripts the property appears at least once, in the following two-step process:

- **Step 1-Keyword Search**: First, we identify if any of the text features used as codes for a property, appears at least once in any of the IaC scripts. As a hypothetical example, if any of the following text features 'dir', and 'file', that are used as codes for a property, appear at least once in a script, then that script is considered for further analysis. Completion of Step 1 will provide a list of scripts which need further inspection in Step 2.
- **Step 2-Manual Examination**: The identified scripts in Step 1 can yield false positives, for example, a script can contain the text feature 'file', even though the script is unrelated with filesystem operations. We apply manual analysis to determine which scripts actually contain the property of interest. We consider a script to contain a property if:
  - the script uses the required IaC syntax to implement the property. (For example, to implement a filesystem operation in Puppet a script must use the 'file' syntax[8]); or
  - the comments in the script reveals the property of interest ('This script changes permission of file a.txt' is an example comment that reveals that the script performs a filesystem operation)

Upon completion of the above-mentioned two-step process, we will identify which properties appear in how many scripts.

3.3 | RQ-2: How can we build prediction models for defective infrastructure as code scripts using text features?

We answer RQ-2 by using the principal components identified from Section 3.1.6. Next, we use statistical learners to construct defect prediction models. We evaluate the performance of cont constructed defect prediction models using six measures.

3.3.1 | Statistical learners to Construct Defect Prediction Models

We use four statistical learners:

[8]https://puppet.com/docs/puppet/5.3/types/file.html

- **Classification and Regression Tree (CART)**: CART generates a tree based on the impurity measure and uses that tree to provide decisions based on input features [7]. We select CART because this learner does not make any assumption on the distribution of features, and is robust to model overfitting [58] [7].
- **Logistic Regression (LR)**: LR estimates the probability that a data point belongs to a certain class, given the values of features [14]. LR provides good performance for classification if the features are roughly linear [14]. We select LR because this learner performs well for classification problems [14], such as defect prediction [50] and fault prediction [19].
- **Naive Bayes (NB)**: The NB classification technique computes the posterior probability of each class to make prediction decisions. We select NB because prior research has reported that defect prediction models that use NB perform well [19].
- **Random Forest (RF)**: RF is an ensemble technique that creates multiple classification trees, each of which are generated by taking random subsets of the training data [6] [58]. Unlike LR, RF does not expect features to be linear for good classification performance. Researchers [17] recommended the use of statistical learners that use ensemble techniques to build defect prediction models.

3.3.2 | Automated Tuning

We apply two tuning strategies to investigate if tuning helps improve the performance of our constructed defect prediction models. The two tuning strategies are **tuning parameters of statistical learners** and **tuning of text feature matrices**, which we describe below:

Tuning Parameters of Statistical Learners

We use differential evolution (DE) [56], a search-based algorithm, to automatically tune the parameters of RF. We select DE because using DE as a parameter tuning technique, researchers have observed improved prediction performance for software defects [15]. For a given measure of quality and a set of input parameters that needs to be tuned, DE iteratively identifies the combination of parameter values for which the given measure of quality is optimal. Each combination of parameter value is referred as a 'candidate solution' in DE. DE achieves optimization by generating a population of candidate solutions and creating new candidate solutions by combining existing ones. Four attributes of DE can be set to control the generation of populations: GENERATION, POPULATION, cross-over probability (CR), and mutation constant (F). In our case, parameters of each statistical learner is the set of input parameters that need to be tuned. The given measure of quality is the prediction performance measure. We set GENERATION, POPULATION, CR, and F to respectively, 50, 10, 0.50, and 0.50. We repeat the above-mentioned process separately for the six prediction performance measures AUC, precision, recall, F-measure, accuracy, and G-Mean.

We determine the parameters for tuning by applying the following steps:

- **Step-1**: We search the ACM Digital Library, IEEEXplore, ScienceDirect, and Springer using keywords 'parameter tuning', 'defect prediction', and 'software engineering'.
- **Step-2**: Next, we read the collected publication from Step-1 to identify if the publication (i) reported the use of a statistical learner; and (ii) reported which parameters were tuned to report variation in prediction performance for the learner.
- **Step-3**: We read the publications collected from Step-2, to determine if they used CART, LR, or RF. If so, from these publications we determine which parameter of each statistical learner needs to be tuned and the ranges of values we can select.

Tuning of Text Feature Matrices

We take motivation from Krishna et al. [29]'s findings where they reported the impact of text feature count on text feature-based prediction. They observed that upon construction of a TF-IDF matrix, improved prediction performance can be obtained by selecting the top-1000 text features with the highest TF-IDF scores. In this experiment, similar to Krishna et al. [29], for each dataset we first sort the count-of-words and TF-IDF matrices. Then we select the top-i text features from the sorted matrices. We vary i from 250, 500, ... total number of text features existing in each dataset. Next, we construct defect prediction models using the selected text features and measure prediction performance with respect to the six metrics. We repeat the same process for both approaches: BOW and TF-IDF.

3.3.3 | 10×10-fold Cross Validation

We use 10×10-fold cross validation to evaluate the constructed prediction models. In the 10-Fold cross validation evaluation approach, the dataset is partitioned into 10 equal sized subsamples or folds [58]. The performance of the constructed prediction models are tested by using nine of the 10 folds as training data, and the remaining fold as test data. Similar to prior work [17], we repeat the 10-fold cross validation 10 times to assess prediction stability.

TABLE 1 Filtering Criteria to Construct Defect Datasets

Criteria	Dataset			
	Mirantis	Mozilla	Openstack	Wikimedia
Criteria-1	26	1,594	1,253	1,638
Criteria-2	20	2	61	11
Criteria-3	20	2	61	11
Final	20	2	61	11

4 | DATASETS

We construct datasets using Puppet scripts from OSS repositories maintained by four organizations: Mirantis, Mozilla, Openstack, and Wikimedia Commons. We select Puppet because it is considered as one of the most popular tools to implement IaC [27] [53], and has been used by organizations since 2005 [35]. Mirantis is an organization that focuses on the development and support of cloud services, such as OpenStack [9]. Mozilla is an OSS community that develops, uses, and supports Mozilla products, such as Mozilla Firefox [10]. The Openstack foundation is an open-source software platform for cloud computing where virtual servers and other resources are made available to customers [11]. The Wikimedia Foundation is a non-profit organization that develops and distributes free educational content [12].

4.1 | Repository Collection

We apply the three selection criteria presented in Section 3.1.1 to identify the repositories that we use for analysis. We describe how many of the repositories satisfied each of the three criteria in Table 1. Each row corresponds to the count of repositories that satisfy each criteria. For example, 26 repositories satisfy Criteria-1, for Mirantis. In total, we obtain 94 repositories to extract Puppet scripts from.

4.2 | Commit Message Processing

We report summary statistics on the collected repositories in Table 2. As shown we altogether collect 12,875 commits in which 2,424 Puppet scripts are modified. Of the 12,875 commits, 5,308 of them included links to issue reports. The amount of defect-related commits varied between 25.4% and 33.7%. The amount of defective scripts varied between 44.6% and 58.5%. The constructed datasets used for empirical analysis are available online [13].

4.3 | Determining Defect-related Commits

We use 89 raters to determine the defect-related commits, using the following phases:

- **Categorization Phase**:
  - **Mirantis**: We recruit students in a graduate course related to software engineering via e-mail. The number of students in the class was 58, and 32 students agreed to participate. We follow IRB#12130, in recruitment of students. We randomly distribute the 1,021 XCMs amongst the students such that each XCM is rated by at least two students. The average professional experience of the 32 students in software engineering is 1.9 years. On average, each student took 2.1 hours.
  - **Mozilla**: One second year PhD student and one fourth year PhD student separately apply qualitative analysis on 3,074 XCMs. The fourth and second year PhD student, respectively, have a professional experience of three and two years in software engineering. The fourth and second year PhD student, respectively, took 37.0 and 51.2 hours to complete the categorization.
  - **Openstack**: One second year PhD student and one first year PhD student separately, apply qualitative analysis on 7,808 XCMs from Openstack repositories. The second and first year PhD student respectively, have a professional experience of two and one years in

[9] https://www.mirantis.com/
[10] https://www.mozilla.org/en-US/
[11] https://www.openstack.org/
[12] https://wikimediafoundation.org/
[13] https://figshare.com/s/ad26e370c833e8aa9712

TABLE 2 Summary statistics of constructed datasets

Statistic	Dataset			
	Mirantis	Mozilla	Openstack	Wikimedia
Puppet Scripts	165	580	1,383	296
Puppet Code Size (LOC)	17,564	30,272	122,083	17,439
Defective Puppet Scripts	91 of 165, 55.1%	259 of 580, 44.6%	810 of 1383, 58.5%	161 of 296, 54.4%
Commits with Puppet Scripts	1,021	3,074	7,808	972
Commits with Report IDs	82 of 1021, 8.0%	2764 of 3074, 89.9%	2252 of 7808, 28.8%	210 of 972, 21.6%
Defect-related Commits	344 of 1021, 33.7%	558 of 3074, 18.1%	1987 of 7808, 25.4%	298 of 972, 30.6%

software engineering. The second and first year PhD student completed the categorization of the 7,808 XCMs respectively, in 80.0 and 130.0 hours.

- **Wikimedia**: 54 graduate students recruited from the 'Software Security' course are the raters. We randomly distribute the 972 XCMs amongst the students such that each XCM is rated by at least two students. According to our distribution, 140 XCMs are assigned to each student. The average professional experience of the 54 students in software engineering is 2.3 years. On average, each student took 2.1 hours to categorize the 140 XCMs. The IRB protocol was IRB#9521.

- **Resolution Phase**:

  - **Mirantis**: Of the 1,021 XCMs, we observe agreement for 509 XCMs and disagreement for 512 XCMs, with a Cohen's Kappa score of 0.21. Based on Cohen's Kappa score, the agreement level is 'fair' [31].
  - **Mozilla**: Of the 3,074 XCMs, we observe agreement for 1,308 XCMs and disagreement for 1,766 XCMs, with a Cohen's Kappa score of 0.22. Based on Cohen's Kappa score, the agreement level is 'fair' [31].
  - **Openstack**: Of the 7,808 XCMs, we observe agreement for 3,188 XCMs, and disagreements for 4,620 XCMs. The Cohen's Kappa score was 0.21. Based on Cohen's Kappa score, the agreement level is 'fair' [31].
  - **Wikimedia**: Of the 972 XCMs, we observe agreement for 415 XCMs, and disagreements for 557 XCMs, with a Cohen's Kappa score of 0.23. Based on Cohen's Kappa score, the agreement level is 'fair' [31].

  The first author of the paper was the resolver, and resolved disagreements for all four datasets. In case of disagreements the resolver's categorization is considered as final. We observe that the raters agreement level to be 'fair' for all four datasets.

  Practitioner Agreement: We report the agreement level between the raters' and the practitioners' categorization for randomly selected 50 XCMs as following:

  - **Mirantis**: We contact three practitioners and all of them respond. We observe a 89.0% agreement with a Cohen's Kappa score of 0.8. Based on Cohen's Kappa score, the agreement level is 'substantial' [31].
  - **Mozilla**: We contact six practitioners and all of them respond. We observe a 94.0% agreement with a Cohen's Kappa score of 0.9. Based on Cohen's Kappa score, the agreement level is 'almost perfect' [31].
  - **Openstack**: We contact 10 practitioners and all of them respond. We observe a 92.0% agreement with a Cohen's Kappa score of 0.8. Based on Cohen's Kappa score, the agreement level is 'substantial' [31].
  - **Wikimedia**: We contact seven practitioners and all of them respond. We observe a 98.0% agreement with a Cohen's Kappa score of 0.9. Based on Cohen's Kappa score, the agreement level is 'almost perfect' [31].

We observe that the agreement between ours and the practitioners' categorization varies from 0.8 to 0.9, which is higher than that of the agreement between the raters in the Categorization Phase. One possible explanation can be related to how the resolver resolved the disagreements. The first author of the paper has industry experience in writing IaC scripts, which may help to determine categorizations that are consistent with practitioners. Another possible explanation can be related to the sample provided to the practitioners. The provided sample, even though randomly selected, may include commit messages whose categorization are relatively easy to agree upon.

5 | EMPIRICAL FINDINGS

In this section we present our empirical findings.

5.1 | RQ-1: What operations characterize defective infrastructure as code (IaC) scripts? How frequently do the identified characteristics appear in IaC scripts?

We identify 1699, 1808, 3545, and 2398 unique text features from all IaC scripts, respectively, for Mirantis, Mozilla, Openstack, and Wikimedia. By applying PCA analysis, we observe that, respectively, for Mirantis, Mozilla, Openstack, and Wikimedia, 100, 393, 437, and 327 components account for at least 95% of the total variance when the bag-of-words technique (BOW) is applied. When TF-IDF is applied, we observe 100, 557, 485, and 662 components account for 95% of the total variance. For both BOW and TF-IDF, the components identify text features that are correlated with defective scripts. As described in Section 3.2, we use these text features to derive properties that characterize defective IaC scripts using SGT. We identify three properties that characterize defective IaC scripts. These properties are: 'filesystem operations', 'infrastructure provisioning', and 'managing user accounts'. All three properties are derived from text features using BOW and TF-IDF. Each of these properties correspond to an operation executed in an IaC script. We list the identified properties that characterize defective IaC scripts with example code snippets in Table 3. We list each property in the 'Characteristic' column, and a corresponding example code snippet in the 'Example Code Snippet' column. We briefly describe each property as following:

- ***Filesystem operations***: Filesystem operations are related to performing file input and output tasks, such as setting permissions of files and directories. For example, in Table 3, we report a code snippet that assigns permission mode '0444' to the file '/etc/firejail/thumbor.profile'. The file is assigned to owner 'root', and belongs to the group 'root'.
- ***Infrastructure provisioning***: This property relates to setting up and managing infrastructure for specialty systems, such as data analytics and database systems. From our qualitative analysis, we identify four types of systems that are provisioned: build systems, data analytics systems, database systems, and web server systems. Cito et al. [8] observed that IaC tools have become essential in cloud-based provisioning, and our finding provides further evidence to this observation. Vendors for IaC tools, such as Puppet [14], advertise automated provisioning of infrastructure as one of the major capabilities of IaC tools, but our results indicate that the capability of provisioning via IaC tools can introduce defects.
- ***Managing user accounts***: This property of defective IaC scripts is associated with setting up accounts and user credentials. In Table 3, we provide an example on how user 'puppetsync' is created. One of the major tasks of system administrators is to setup and manage user accounts in systems [2]. IaC tools, such as Puppet, provide API methods to create and manage users and their credentials in the system. According to some practitioners [28], IaC tools, such as Puppet, can only be beneficial for managing a small number of users, and managing large number of users increases the chances of introducing defects in scripts.

5.2 | Frequency of the Identified Characteristics

As described in Section 3.2, we apply a two-step process to calculate the frequency of the properties that characterize defective scripts. After executing the keyword search step (Step-1), we identify certain scripts that include the three operations: filesystem, infrastructure provisioning, and managing user accounts. We have presented these findings in Table 4. Each cell in the table presents how many of the total IaC scripts in a certain dataset include a certain operation based on keyword search.

Finally, after completion of manual examination (Step 2), we report the frequency of identified properties that characterize defective IaC scripts in Table 5. The 'Characteristics' column represents a property, and in the 'Frequency' column we report the frequency of each property. We observe that for Mozilla 21.7% of scripts contain filesystem operations. The 'Infrastructure provisioning (total)' row presents the summation of the four provisioning-related operations: provisioning of (i) build, (ii) data analytics, (iii) database, and (iv) web server systems. We observe the reduction in the count of scripts for which each property is observed, upon application of Step-2. Our findings suggest that the keyword-based matching technique can generate a lot of false positives, and manual inspection can filter out these false positives, as demonstrated by Bosu et al. [5], for detecting vulnerable code changes.

Table 5 also indicates how many scripts can be prioritized for verification and validation efforts. For example, considering filesystem operations for Mirantis, 15.1% of the total scripts can be prioritized. As shown in the 'Total' row, considering all three properties, namely filesystem, infrastructure provisioning, and managing user accounts, then instead of using all 165 IaC scripts for verification and validation, 104 (63.0%) can be prioritized.

[14] https://puppet.com/products/capabilities/automated-provisioning

TABLE 3 Operations in Defective IaC Scripts

Characteristic	Example Code Snippet

Filesystem operations	`file {'/etc/firejail/thumbor.profile':`
	`  ensure => present,`
	`  owner => 'root',`
	`  group => 'root',`
	`  mode  => '0444',`
	`  source => 'puppet:///modules/thumbor/thumbor.profile',`
	`}`

Infrastructure provisioning	*Build systems*
	`exec {`
	`  'add-builder-to-mock_mozilla':`
	`  command => "/usr/bin/gpasswd -a ${users::builder::username} mock_mozilla",`
	`  unless => "/usr/bin/groups ${users::builder::username} | grep '\\<mock_mozilla\\>'",`
	`  require => [Class['packages::mozilla::mock_mozilla'], Class['users::builder']];`
	`}`
	Data analytics systems
	`service {'elasticsearch':`
	`  ensure => running,`
	`  enable => true,`
	`  require => [`
	`  Package['elasticsearch', 'openjdk-7-jre-headless'],`
	`  File['/var/run/elasticsearch/'],`
	`  ]`
	`}`
	Database systems
	`mysql::user { $extension_cluster_db_user:`
	`  password => $extension_cluster_db_pass,`
	`  grant  => "ALL PRIVILEGES ON ${extension\_cluster\_shared\_db\_name}.*"`
	`}`
	Web server systems
	`file{'/etc/apache2/ports.conf':`
	`  content => template('apache/ports.conf.erb'),`
	`  require => Package['apache2'],`
	`  notify => Service['apache2'],`
	`}`

Managing User Accounts	`user {`
	`  'puppetsync':`
	`  managehome => true,`
	`  home => $homedir,`
	`  password => '*', # unlock the account without setting a password`
	`  comment => "synchronizes data between puppet masters";`
	`}`

TABLE 4 Frequency of Identified Operations Based on Keyword Search

Characteristics	Frequency			
	MIR	MOZ	OST	WIK
Filesystem operations	55.2%	37.5%	50.9%	67.2%
Infrastructure provisioning	43.7%	51.0%	42.1%	26.7%
Managing user accounts	16.6%	31.3%	65.0%	41.9%

Similarly, considering all three properties, 50.8%, 31.1%, 34.5% and 42.9% of all scripts respectively in Mirantis, Mozilla, Openstack and Wikimedia can be prioritized for verification and validation.

5.3 | RQ-2: How can we build prediction models for defective infrastructure as code scripts using text features?

We report our findings related to our DE-based prediction models in Section 5.3.1. We report our findings related to tuning text feature matrix in Section 5.3.2.

TABLE 5 Frequency of Identified Operations Based on Keyword Search and Manual Examination

Characteristics	Frequency			
	MIR	MOZ	OST	WIK
Filesystem operations	15.1%	21.7%	14.5%	23.4%
Infrastructure provisioning (build systems)	0.5%	2.8%	0.0%	0.0%
Infrastructure provisioning (data analytics systems)	1.1%	2.7%	6.2%	5.4%
Infrastructure provisioning (database systems)	10.5%	0.8%	7.7%	4.3%
Infrastructure provisioning (web server systems)	17.0%	0.6%	5.0%	8.2%
Infrastructure provisioning (total)	29.1%	6.9%	18.9%	17.9%
Managing user accounts	6.6%	2.5%	1.1%	1.6%
Total	50.8%	31.1%	34.5%	42.9%

TABLE 6 Parameters Selected for Tuning

Learner	Parameter	Description	Range
CART	max_depth [15]	Maximum depth of the tree. By default, CART expands the tree until all leaves belong to only one class [42].	[1, 50]
	max_features [15]	Number of features to consider for the best split. By default, CART uses all features to build a tree [42].	[0.01, 1.0]
	min_samples_split [15]	Minimum number of samples required to split an internal node. Default: 2	[2, 20]
	min_samples_leaf [15]	Minimum number of samples required to be at a leaf node. Default: 1	[1, 20]
LR	λ [11]	Penalty to increase the magnitude of features to reduce overfitting. Default: 1.0	[0.01, 1.0]
	penalty [11]	Penalty function. Default: 'l2'	'l1','l2'
NB	Distribution [34] [63]	Assumption on the underlying probabilistic distribution. Default: Gaussian	'Gaussian()', 'Bernoulli()'
RF	max_features [15]	Number of features to consider for the best split. By default, RF uses square root of features to build a tree [42].	[0.01, 1.0]
	max_leaf_nodes [15]	Grow trees with max_leaf_nodes in best first fashion. By default, RF does not limit the count of nodes to grow trees [42].	[1, 50]
	min_samples_split [15]	Minimum number of samples required to split an internal node. Default: 2	[2, 20]
	min_samples_leaf [15]	Minimum number of samples required to be at a leaf node. Default: 1	[1, 20]
	n_estimators [15]	Number of trees in the forest. Default: 100	[50, 150]

5.3.1 | Building Defect Prediction Models Using Differential Evolution (DE)

Using the steps described in Section 3.3.2, we obtain the list of parameters that need to be tuned. By executing **Step-1**, we collect 10, 2, 44, and 12 publications from IEEEXplore, ACM Digital Library, ScienceDirect, and Springer, respectively. From this total collection of 68 publications, we observe nine publications to report their use of statistical learner with at least one parameter (**Step-2**). Finally, in **Step-3**, we observe three of the nine publications, to use CART, LR, or RF. From these publications we determine what parameters need to be tuned and the range of values for each parameter. Of the reported parameters, we only select those parameters for tuning which are available as part of the Scikit Learn API [42].

In Table 6, we report the parameters we selected for tuning using DE. The 'Parameter' column presents the parameter name followed by the reference of the publication from which we derive the parameter and the value range. The 'Description' and 'Range' columns, respectively, describe a short description of the parameter and the range of values for the parameter. All parameters are numeric except for 'penalty', a parameter of LR. 'penalty' is string-based.

The median AUC values are presented in Table 7. The 'BOW' and 'TF-IDF' columns provides the median AUC values, respectively, for the BOW and TF-IDF technique. For AUC the BOW technique outperforms the TF-IDF technique for all four datasets.

We report the median precision, recall, F-measure, accuracy, and G-mean values for 10 $\times$ 10 cross validation, for all learners and all datasets respectively in Tables 8, 9, 10, 11, and 12. With respect to median accuracy, precision, and F-measure, the BOW technique outperforms the TF-IDF

TABLE 7 Median AUC for two model building techniques: BOW and TF-IDF.

Dataset	BOW				TF-IDF			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.69	0.61	0.64	**0.73**	0.66	0.66	0.61	0.71
MOZ	0.61	0.64	0.52	**0.65**	0.58	0.58	0.51	0.53
OST	0.58	**0.63**	0.61	0.59	0.54	0.57	0.57	0.55
WIK	0.63	0.62	**0.68**	0.67	0.56	0.56	0.55	0.56

TABLE 8 Median precision for two model building techniques: BOW and TF-IDF.

Dataset	BOW				TF-IDF			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.69	0.63	**0.75**	0.72	0.68	0.66	0.64	0.72
MOZ	0.61	0.64	0.63	0.65	0.56	0.51	0.46	**0.67**
OST	0.68	0.71	**0.76**	0.66	0.62	0.64	0.64	0.62
WIK	0.66	0.75	**0.76**	0.68	0.59	0.58	0.59	0.58

TABLE 9 Median recall for two model building techniques: BOW and TF-IDF.

Dataset	BOW				TF-IDF			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.76	0.53	0.67	0.84	0.74	**0.85**	0.68	0.84
MOZ	**0.61**	0.41	0.51	0.65	0.56	0.39	0.53	0.43
OST	0.67	0.61	0.71	0.70	0.72	**1.00**	0.71	0.98
WIK	0.68	0.47	0.75	0.80	0.69	**1.00**	0.94	0.82

technique for three datasets. The TF-IDF technique outperforms the BOW technique for three datasets with respect to median recall. With respect to median G-mean, for two datasets the bag-of-words technique outperforms TF-IDF.

5.3.2 | Building Prediction Models By Tuning Text Feature Matrix

We present our findings on how tuning text feature matrix can impact prediction performance for the two techniques: BOW and TF-IDF.

Tuning Text Feature Matrix with the Bag-of-words Technique

When we apply the text feature matrix tuning technique, we observe the impact on prediction performance. We present our findings in Figures 5 to 8. In Figures 5, 6, 7, and 8 we, respectively, present the impact of tuning text feature matrix on prediction performance for Mirantis, Mozilla, Openstack, and Wikimedia. In each of these figures, we report the prediction performance for the four learners CART, LR, NB, and RF. In each figure the

TABLE 10 Median F-measure for two model building techniques: BOW and TF-IDF.

Dataset	BOW				TF-IDF			
	CART	LR	NB	RF	CART	LR	NB	RF
MIR	0.71	0.59	0.67	**0.77**	0.67	0.72	0.65	0.72
MOZ	0.57	0.49	0.49	**0.64**	0.54	0.44	0.49	0.49
OST	0.65	0.65	0.69	0.67	0.69	0.74	0.66	**0.74**
WIK	0.66	0.56	0.71	**0.72**	0.59	0.70	0.71	0.65

TABLE 11 Median accuracy for two model building techniques: BOW and TF-IDF.

Dataset	BOW				TF-IDF			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.67	0.60	0.65	0.67	0.65	0.67	0.61	**0.71**
MOZ	0.62	0.62	0.63	**0.67**	0.58	0.54	0.51	0.61
OST	0.59	**0.63**	0.63	0.60	0.56	0.60	0.58	0.60
WIK	0.62	0.60	0.67	**0.68**	0.56	0.57	0.62	0.55

TABLE 12 Median G-mean for two model building techniques: BOW and TF-IDF.

Dataset	BOW				TF-IDF			
	CART	**LR**	**NB**	**RF**	**CART**	**LR**	**NB**	**RF**
MIR	0.67	0.70	0.67	0.69	0.67	**0.73**	0.65	0.76
MOZ	0.59	0.50	0.49	**0.64**	0.55	0.45	0.49	0.51
OST	0.65	0.66	0.69	**0.76**	0.69	**0.76**	0.66	0.75
WIK	0.71	0.59	0.71	**0.76**	0.60	0.73	0.73	0.66

x-axis corresponds to the varying amount of text features used to construct the text feature matrix, and y-axis presents the median prediction performance values. We use these figures to demonstrate how tuning of text features impact prediction performance i.e. show increasing, decreasing, or constant trends.

The impact of tuning text feature matrix however is different for different learners and the datasets. For the Mirantis dataset, for LR and NB the recall score decreases, but increases for CART. In case of the Mozilla dataset ,when we use LR, the F-measure, G-mean, precision and recall values decrease when all the text features are used. In a similar manner, we observe when we use Wikimedia with LR, the values decrease for accuracy, AUC, F-measure, G-mean, precision, and recall.

For the Openstack dataset we observe minor differences in prediction performance measures for all learners. However, this finding indicates that using smaller amount of text features can be used to obtain prediction performance measure similar to that of large amount of text features.

Tuning Text Feature Matrix with the TF-IDF Technique

With the TF-IDF technique, we also observe the impact of tuning text feature matrix on prediction performance. We present our findings in Figures 9- 12. In Figures 9, 10, 11, and 12 we, respectively, present the impact of tuning text feature matrix on prediction performance for Mirantis, Mozilla, Openstack, and Wikimedia. In each of these figures, we report the prediction performance for the four learners CART, LR, NB, and RF.

The impact of tuning text feature matrix is observable for specific learners. In the case of TF-IDF analysis, the change in precision and recall is observable for NB. For example, as shown in Figure 9, for NB the precision decreases from 0.78 to 0.64. In a similar manner, from Figure 10, we observe for LR and NB we observe precision to decrease. For LR, the precision when we use 1500 and 1808 text features is respectively 0.70 and 0.46. For NB, the precision when we use 1500 and 1808 text features is respectively 0.67 and 0.44.

For Mozilla, Openstack and Wikimedia, we notice NB to achieve the highest median recall when all text features in the dataset is used, indicating that even though false positives are generated (low precision), using all text features can be helpful to identify defective scripts in the dataset with the TF-IDF approach.

Another finding we observe from the Openstack dataset is that the prediction performance measures remain overall consistent. For example, when we use CART and vary the number of text features selected from 100 to 3545, we observe AUC to vary between 0.51 and 0.55. The highest median AUC is observable when we use 100 test features. This finding indicates that using smaller amount of text features we may obtain the highest possible AUC for a certain learner.

Summary: We summarize or findings from Sections 5.3.1 and 5.3.2 in Table 13. In Table 13, we report the combination of techniques for which we observe the highest median prediction performance. Each cell in the entry is a tuple where we report the learner, feature extraction technique, and the tuning technique i.e. DE or text feature matrix (TFM). For example, for the Mirantis dataset we observe the highest median accuracy when the learner, feature extraction technique, tuning technique is respectively, Naive Bayes (NB), TF-IDF, and text feature matrix (TFM).

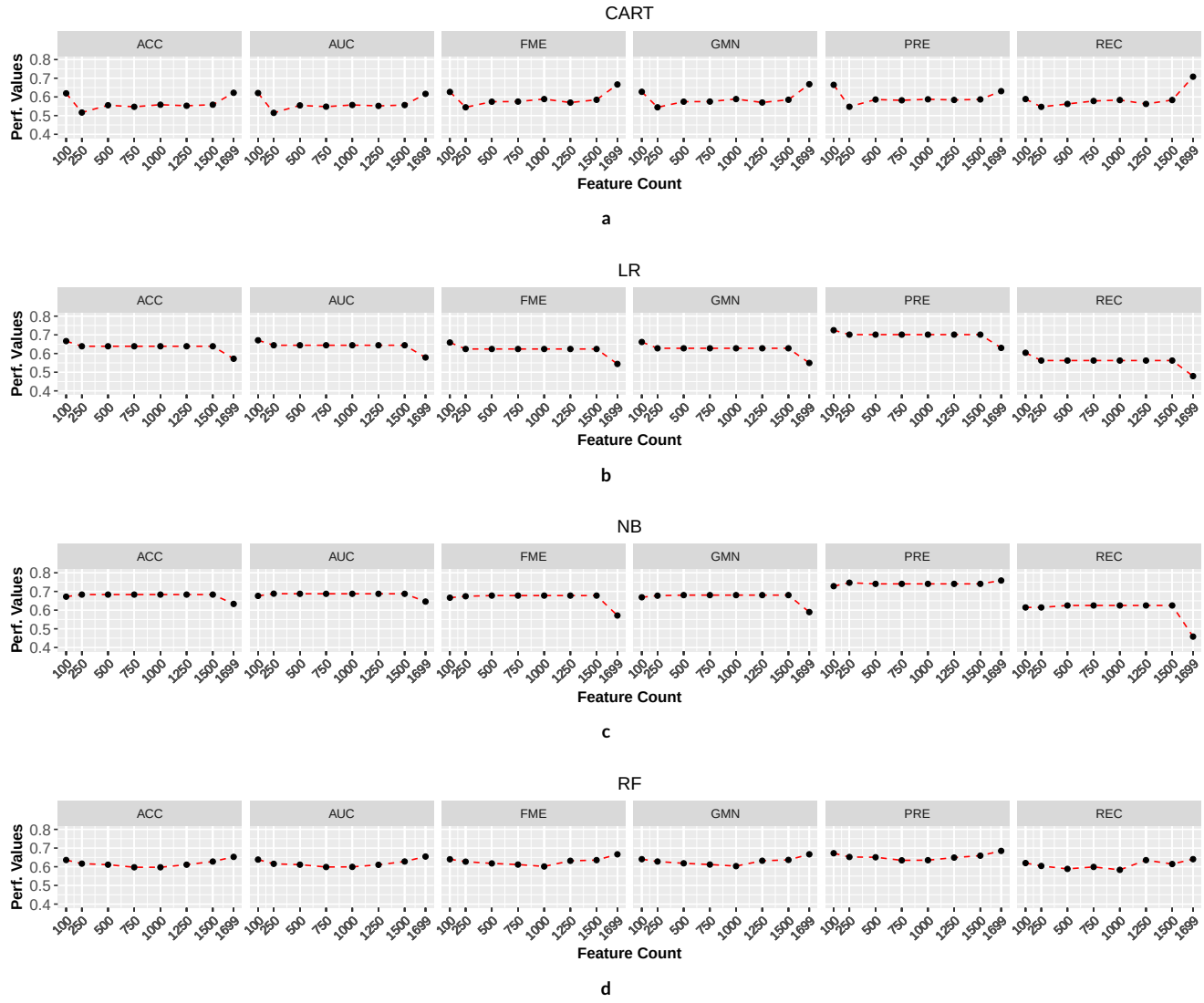


FIGURE 5 The impact of tuning text feature matrix on prediction performance for Mirantis when we use bag-of-words. Figures 5a, 5b, 5c, and 5d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

From Table 13, we observe that for F-measure and Recall the DE tuning technique provides the best performance for all datasets. For precision we observe the highest performance with respect to precision for all four datasets. We observe BOW technique to co-occur more with the DE-based tuning technique compared to the text feature matrix technique. For precision, the TF-IDF technique co-occurs more with the text feature matrix technique instead of BOW.

In short, based on prediction performance measure tuning techniques can impact differently. Along with paramter tuning of statistical learners we also observe the importance of tuning text feature matrices when text features are used.

6 | DISCUSSION

We discuss our findings with possible implications:

Prioritizing Verification and Validation Efforts: As shown in Table 3 and Table 5, one property that characterizes defective IaC scripts is performing file-related operations. Erroneous file mode and file path can make filesystem operations more susceptible to defects. Filesystem operations

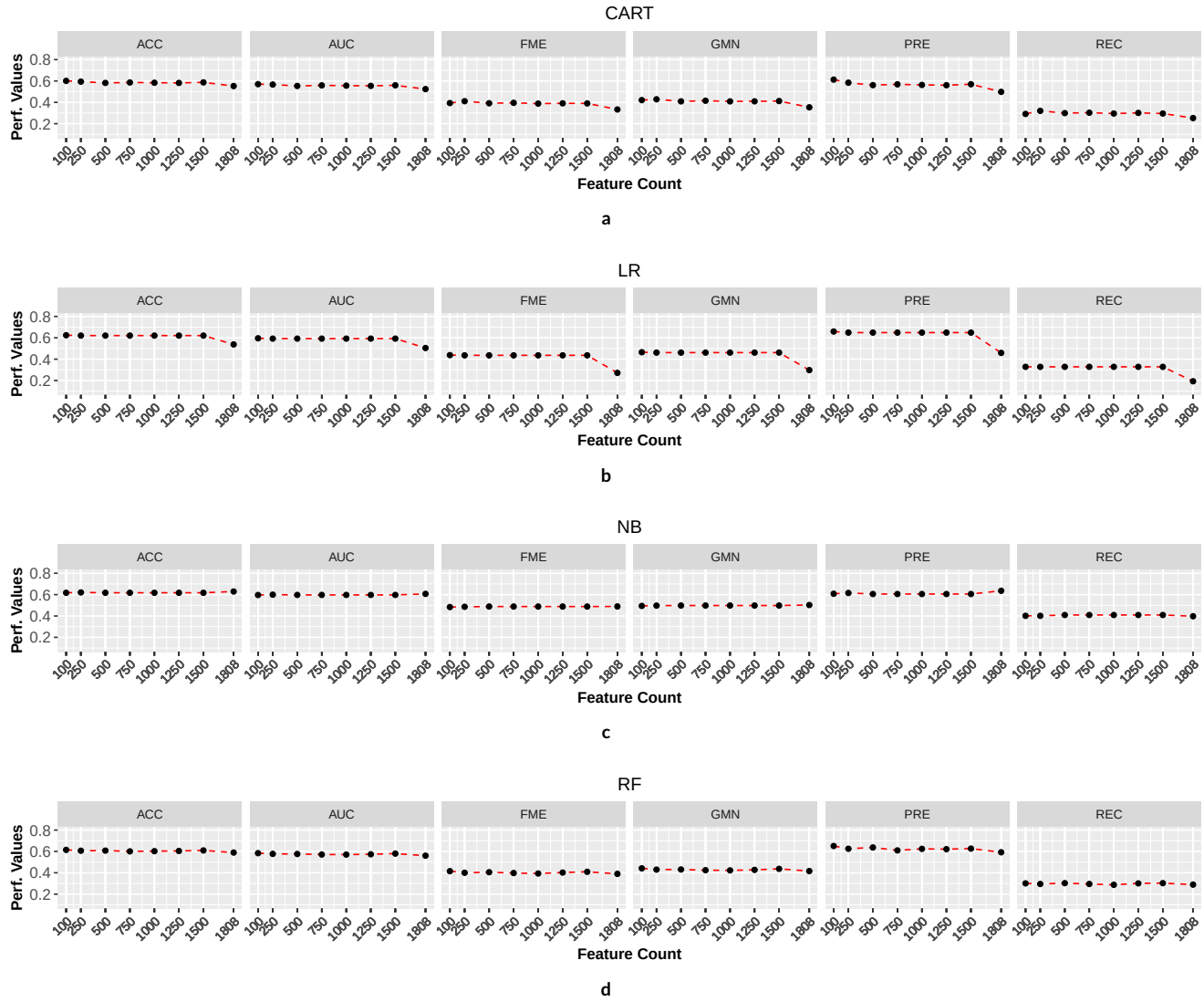


FIGURE 6 The impact of tuning text feature matrix on prediction performance for Mozilla when we use bag-of-words. Figures 6a, 6b, 6c, and 6d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

are performed in 15.1%, 21.7%, 14.5%, and 23.4% of the scripts respectively for Mirantis, Mozilla, Openstack, And Wikimedia. To perform file operations, practitioners have to provide configuration values such as file location and permissions. While assigning these values practitioners might be inadvertently making mistakes and introducing defects to IaC scripts. Software teams can take our findings into account, and prioritize their verification and validation efforts accordingly. They can write tests dedicated for scripts that are used to perform filesystem operations such as setting file locations and permissions. They can also benefit from extra inspection efforts to check if proper configuration values are being assigned in these particular scripts.

Cito et al. [8] interviewed practitioners and observed that in cloud-based application development, use of IaC tools such as Puppet, is fundamental to automated provisioning of development and deployment infrastructure. Findings from our research provides further evidence to their observations. We also observe that infrastructure provisioning can be a source of defects for IaC scripts. Infrastructure provisioning using IaC scripts involves executing a sequence of complex steps, for example installation of third-party packages, ensuring scalability, and handling the sensitive information of systems [35]. While implementing these steps, practitioners might be introducing defects inadvertently. As shown in Table 5, infrastructure provisioning appears respectively for 29.1%, 6.9%, 18.9%, and 17.9% of Mirantis, Mozilla, Openstack, and Wikimedia scripts.

Similar recommendations apply for managing user accounts as well. Similar to filesystem operations, IaC tools provide the options to setup and manage users [30], and practitioners have to provide the proper configuration values in the required format. Our research indicates the practice

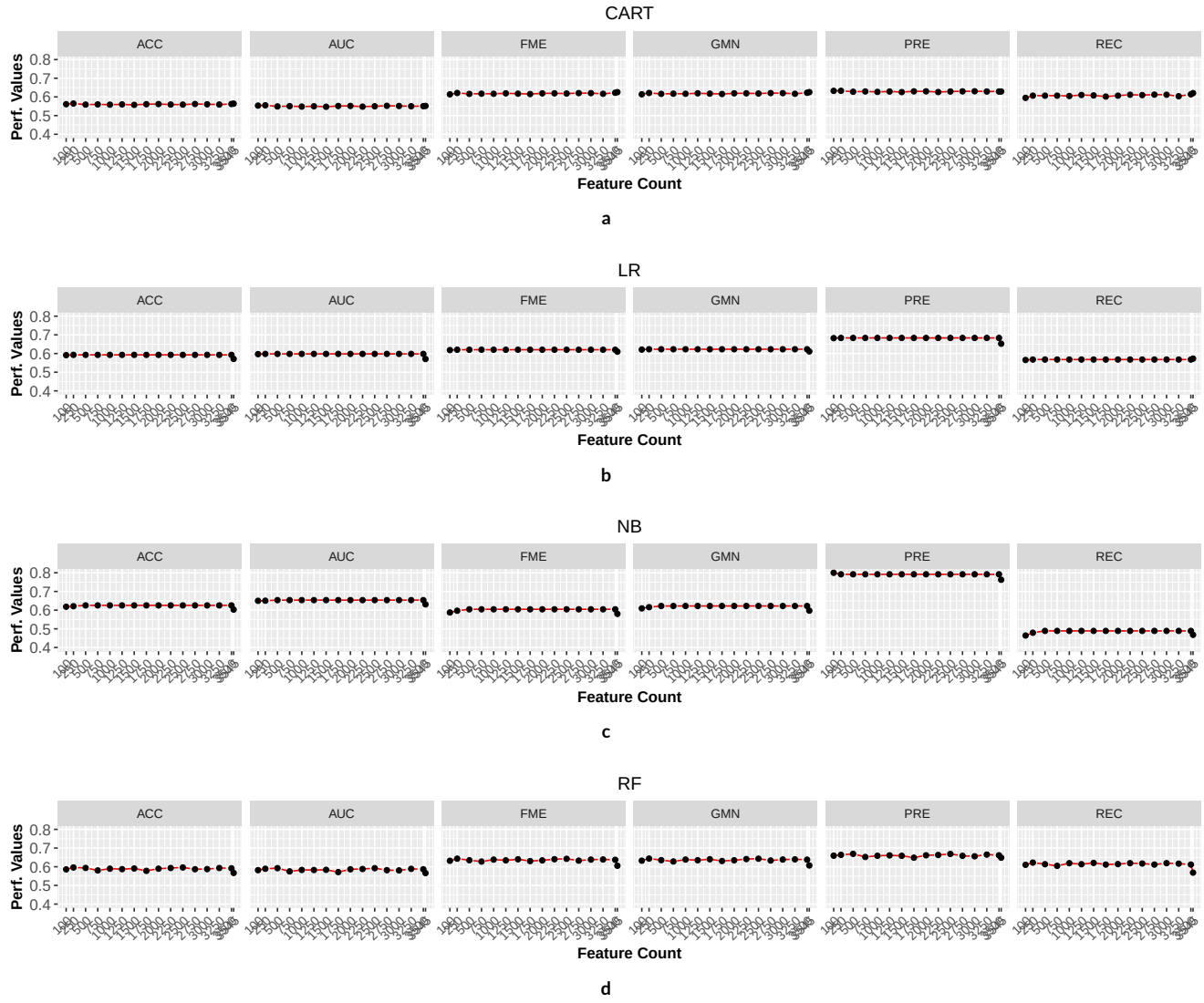


FIGURE 7 The impact of tuning text feature matrix on prediction performance for Openstack when we use bag-of-words. Figures 7a, 7b, 7c, and 7d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

of setting up user accounts is susceptible to defects, and IaC scripts that are used for user account management should be prioritized for more verification and validation. Compared to filesystem operations, scripts used for managing users is smaller: 6.6%, 2.5%, 1.1%, and 1.6% respectively, for Mirantis, Mozilla, Openstack, and Wikimedia.

Tools: Our answer to RQ-2 provides evidence that text features can be a strategy to build defect prediction models for IaC scripts. Building defect prediction models for IaC scripts also provide the opportunity of creating new tools and services for IaC scripts. For software production code, such as C++ and Java code, tools and services such as DevOps Insights [15] exist that predict which source code files can be defect-prone. Toolsmiths can apply text mining on IaC scripts to build defect prediction models for IaC scripts.

Future Research: We investigate two techniques to mine text features. Researchers can investigate if other techniques such as topic modeling [4] and word2vec [37] can be applied to extract text mining features for defect prediction of IaC scripts. Future research can investigate how to improve the accuracy of text feature-based defect prediction models. Researchers can also investigate how text-based features compare with code metrics and process metrics.

[15] https://www.ibm.com/cloud/devops-insights

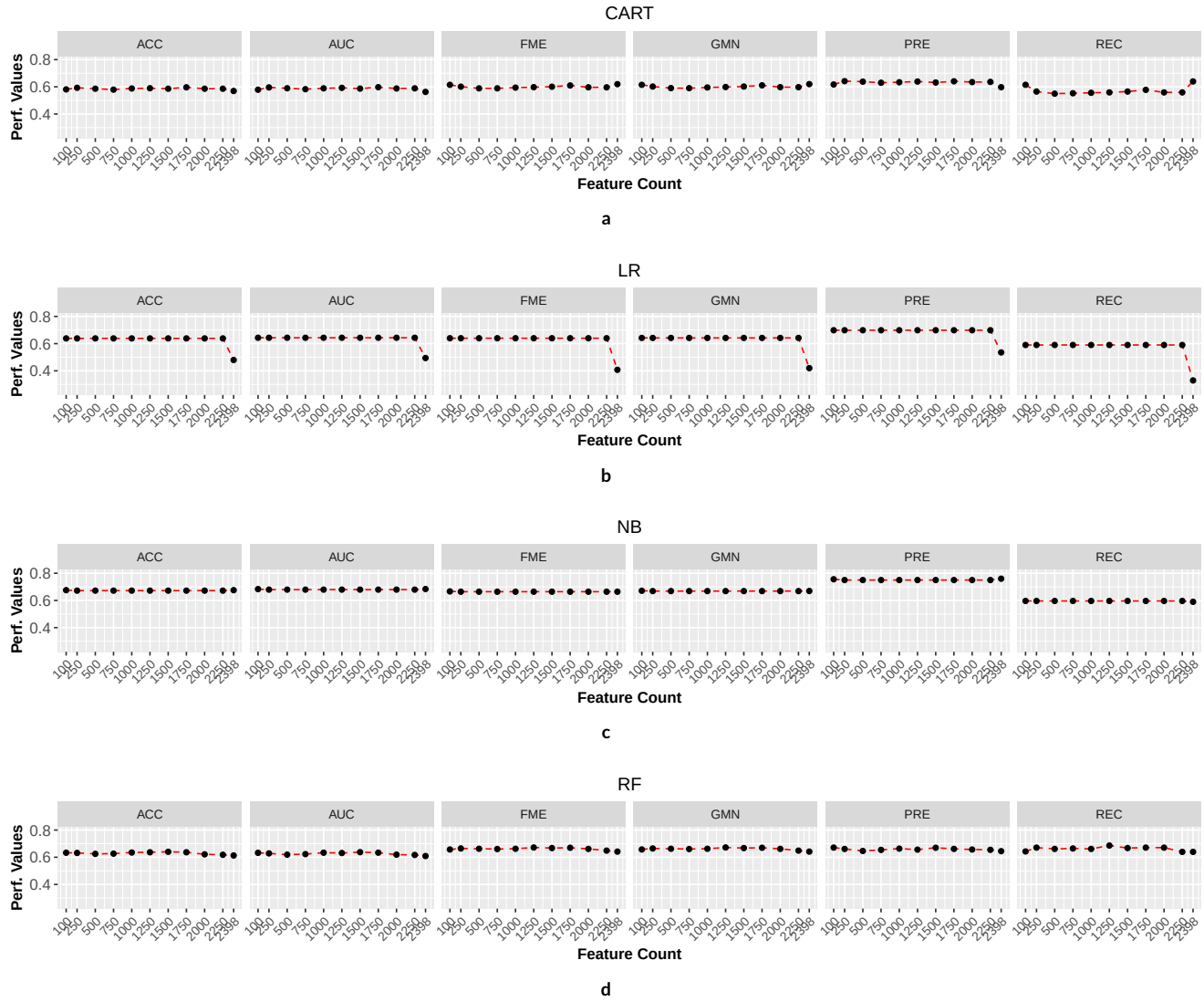


FIGURE 8 The impact of tuning text feature matrix on prediction performance for Wikimedia when we use bag-of-words. Figures 8a, 8b, 8c, and 8d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

Researchers can also benefit from our tuning analysis where we have tuned the parameters of the statistical learners. We have observed that tuning of statistical learners yielded the best prediction performance with respect to F-measure, G-Mean, and recall. On the other hand, for precision the tuning of text feature matrix helped to achieve the highest precision. When using text features, we recommend researchers to explore tuning of text feature matrices as well as tuning of learner parameters.

7 | THREATS TO VALIDITY

We discuss the limitations of our paper as following:

- **Conclusion Validity**: Our approach is based on qualitative analysis, where raters categorized XCMs, and assigned defect categories. We acknowledge that the process is susceptible human judgment, and the raters' experience can bias the categories assigned. The accompanying human subjectivity can influence the distribution of the defect category for IaC scripts of interest. We mitigated this threat by assigning multiple raters for the same set of XCMs. Next, we used a resolver, who resolved the disagreements. Further, we cross-checked our categorization with practitioners who authored the XCMs, and observed 'substantial' to 'almost perfect' agreement.

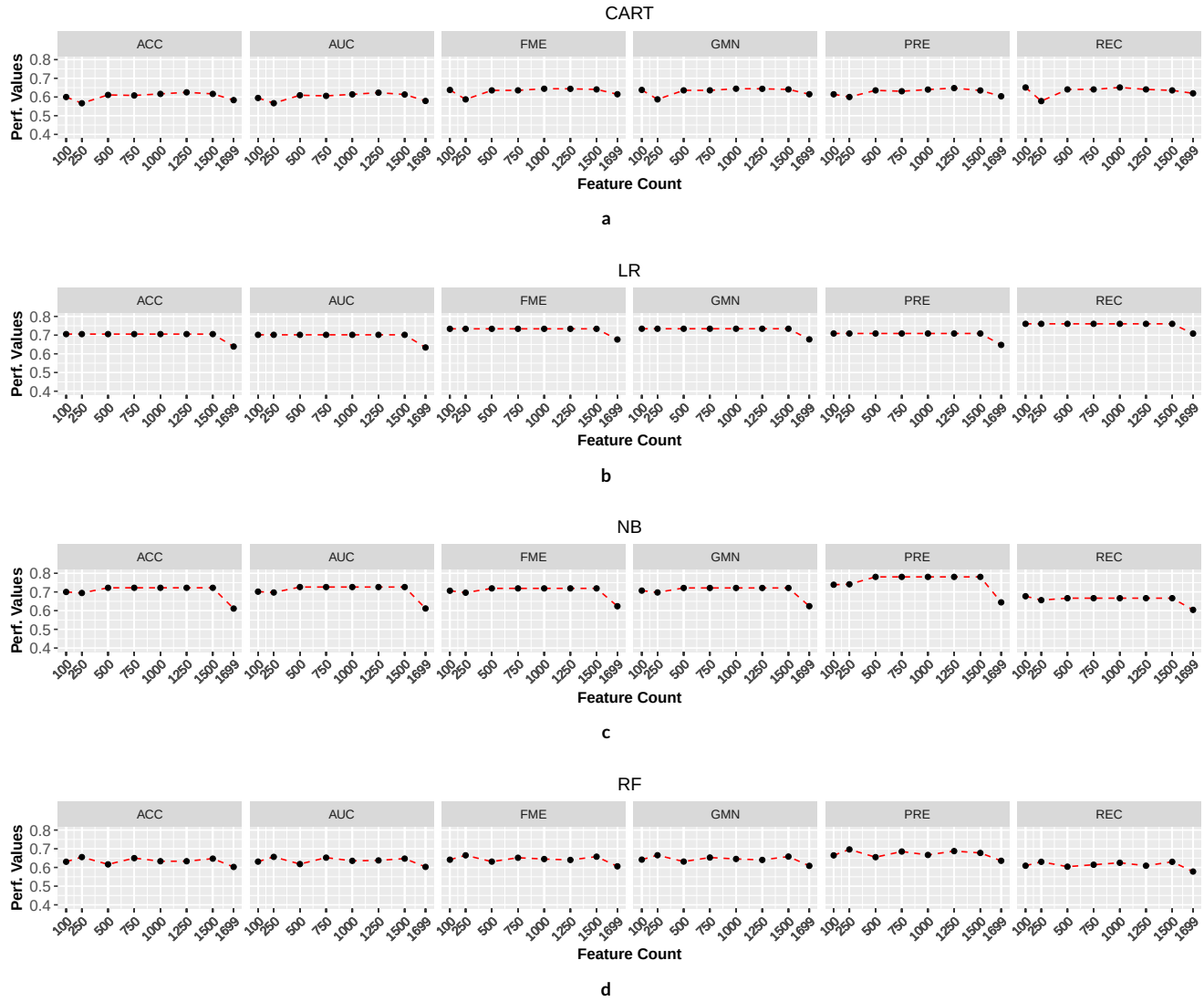


FIGURE 9 The impact of tuning text feature matrix on prediction performance for Mirantis when we use TF-IDF. Figures 9a, 9b, 9c, and 9d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

- **Internal Validity**: We have used a combination of commit messages and issue report descriptions to determine if an IaC script is associated with a defect. We acknowledge that these messages might not have given the full context for the raters. Other sources of information, such as practitioner input, and code changes that take place in each commit could have provided the raters better context to categorize the XCMs.

  We have used two text mining techniques, and we acknowledge that our use of two techniques is not comprehensive. We observe the opportunity to apply sophisticated text mining techniques, such as deep learning for text-based feature discovery. We also acknowledge, some defects such as incorrect file paths may not be captured using text features. We advocate for mining new sets of metrics such as code metrics, and process metrics.

- **Construct validity**: Our process of using human raters to determine defect categories can be limiting, as the process is susceptible to mono-method bias, where subjective judgment of raters can influence the findings. We mitigated this threat by using multiple raters.

  Also, for Mirantis and Wikimedia, we used graduate students who performed the categorization as part of their class work. Students who participated in the categorization process can be subject to evaluation apprehension, i.e. consciously or sub-consciously relating their performance with the grades they would achieve for the course. We mitigated this threat by clearly explaining to the students that their performance in the categorization process would not affect their grades.

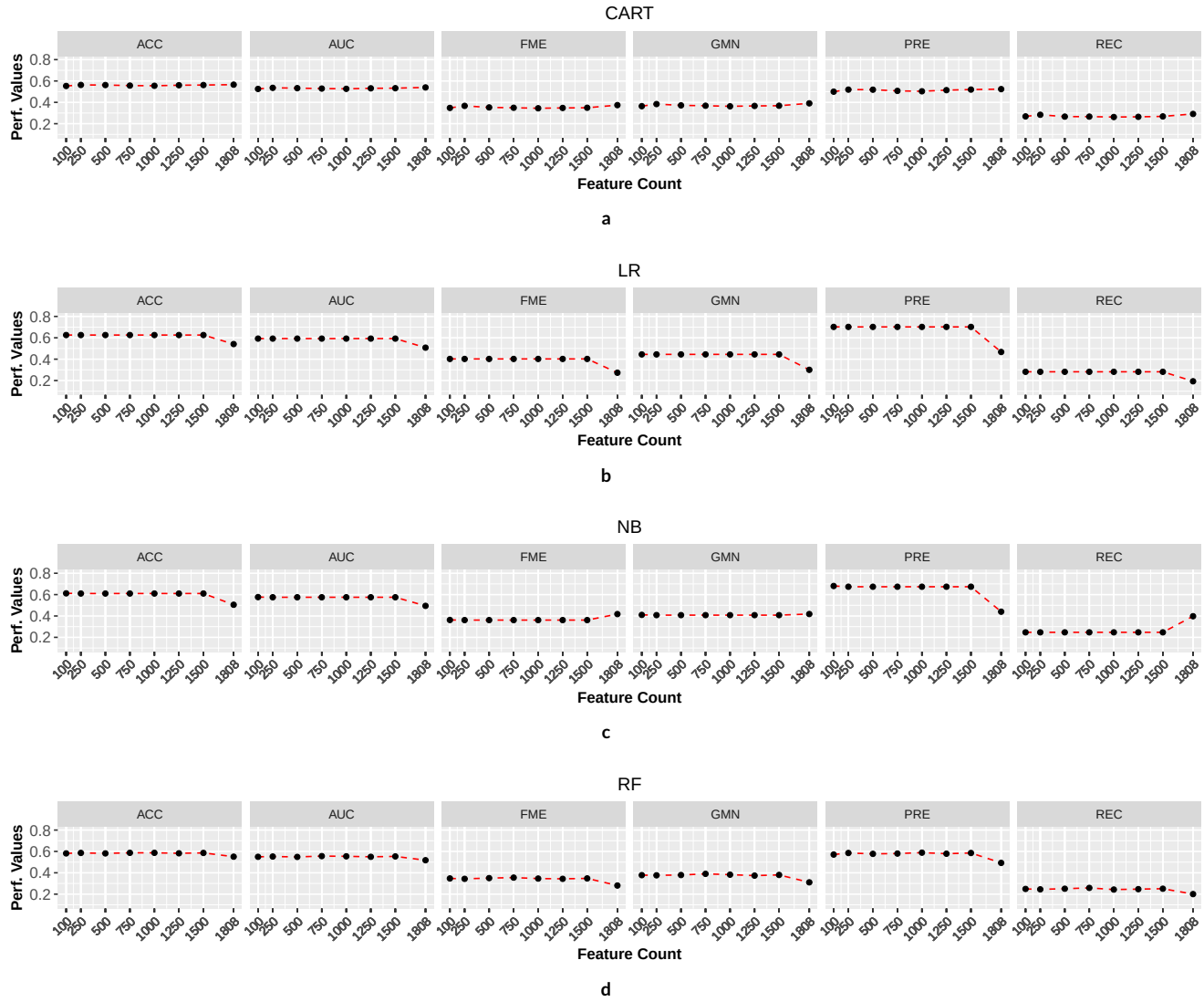


FIGURE 10 The impact of tuning text feature matrix on prediction performance for Mozilla when we use TF-IDF. Figures 10a, 10b, 10c, and 10d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

The raters involved in the categorization process had professional experience in software engineering for at two years on average. Their experience in software engineering may make the raters curious about the expected outcomes of the categorization process, which may effect the distribution of the categorization process. Furthermore, the resolver also has professional experience in software engineering and IaC script development, which could influence the outcome of the defect category distribution.

- **External Validity**: Our scripts are collected from the OSS domain, and not from proprietary sources. Our findings are subject to external validity, as our findings may not be generalizable.

8 | CONCLUSION

IaC scripts provide practitioners the opportunity to build automated deployment pipelines. Similar to software code, IaC scripts can be defective. We focus on identifying characteristics of defective IaC scripts. By applying text mining techniques, and qualitative analysis, we identify three properties that characterize defective scripts: filesystem operations, infrastructure provisioning, and managing user accounts. We observe these three properties appear, respectively, in 31.1%, 34.5%, and 42.9% scripts of the Mozilla, Openstack, and Wikimedia dataset. Next, we build prediction

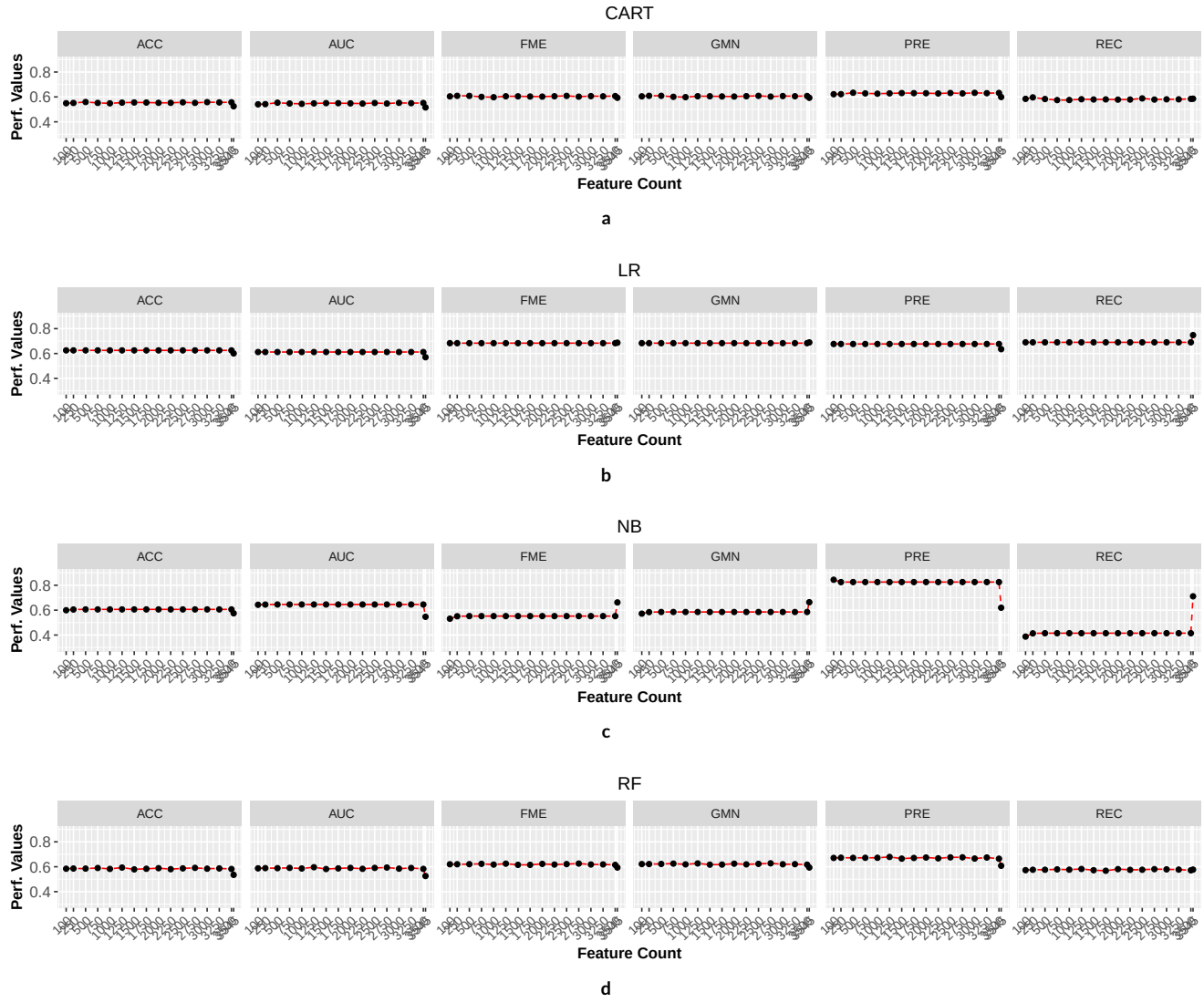


FIGURE 11 The impact of tuning text feature matrix on prediction performance for Openstack when we use TF-IDF. Figures 11a, 11b, 11c, and 11d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

models using statistical learners and parameter tuning of statistical learners. Our constructed defect prediction models using text features yielded a median F-measure of 0.72, 0.64, 0.74, and 0.72, respectively, for Mirantis. Mozilla, Openstack, and Wikimedia Commons. Based on our findings, we advocate practitioners to allocate sufficient validation and verification efforts for IaC scripts which include any of the following operations: filesystem operations, infrastructure provisioning, or managing user accounts. We also investigated two tuning strategies: tuning parameters of statistical learners and tuning text feature matrices. Based on our findings we observe the importance of tuning text feature matrices when text features are used along with paramter tuning of statistical learners.

ACKNOWLEDGMENTS

We thank the members of the RealSearch group for their valuable feedback. Our project is partially funded by the Science of Security Lablet at North Carolina State University.

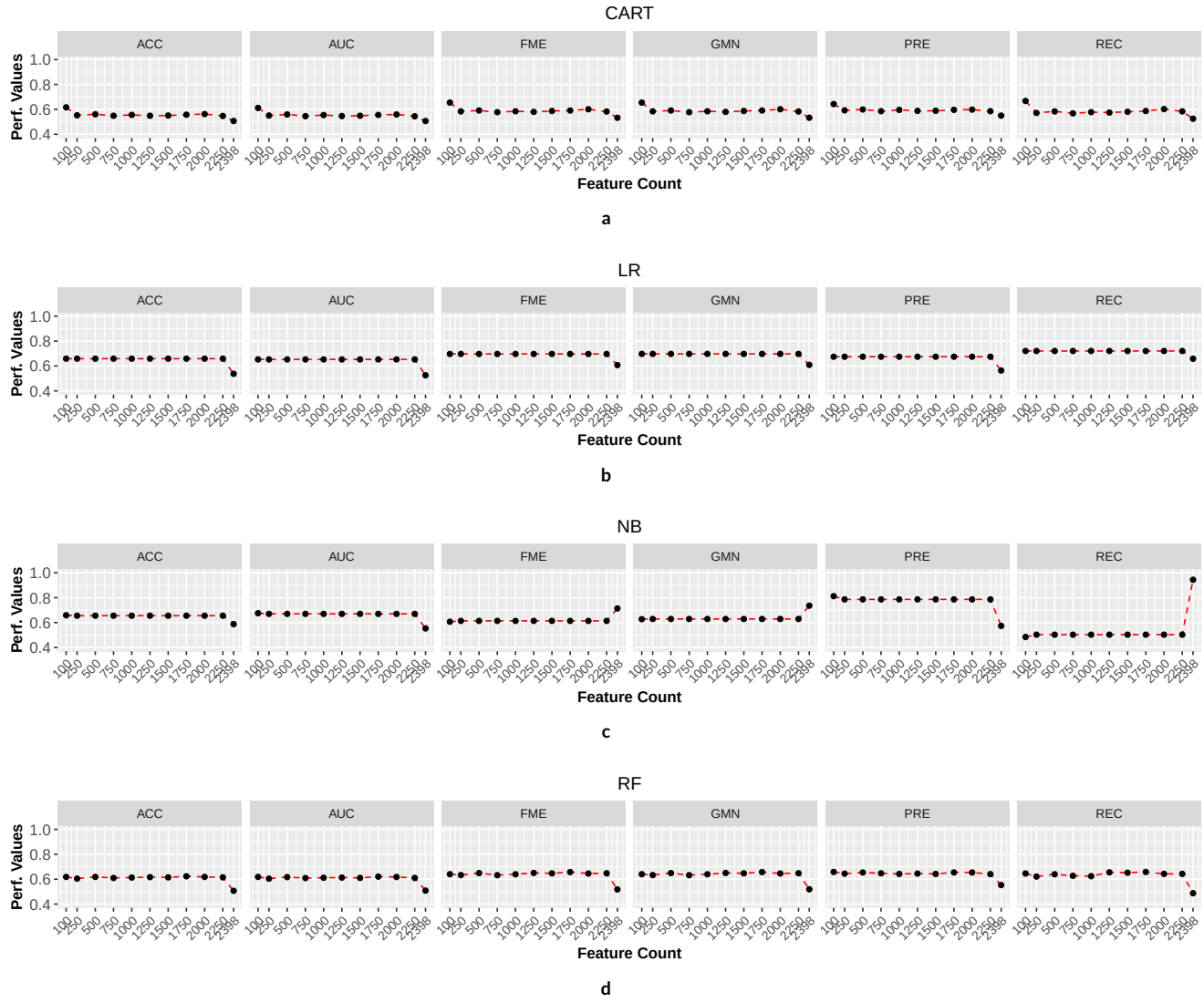


FIGURE 12 The impact of tuning text feature matrix on prediction performance for Wikimedia when we use TF-IDF. Figures 12a, 12b, 12c, and 12d respectively presents the impact of tuning text feature matrix for CART, LR, NB, and RF. Each subfigure presents the performance measures for the six prediction performance measures.

References

[1] Alali, A., H. Kagdi, and J. I. Maletic, 2008: What's a typical commit? a characterization of open source software repositories. *2008 16th IEEE International Conference on Program Comprehension*, 182–191.

[2] Anderson, E. A., 2002: *Researching System Administration*. Ph.D. thesis, aAI3063285.

[3] Aversano, L., L. Cerulo, and C. Del Grosso, 2007: Learning from bug-introducing changes to prevent fault prone code. *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ACM, New York, NY, USA, IWPSE '07, 19–26. URL http://doi.acm.org/10.1145/1294948.1294954

[4] Blei, D. M., A. Y. Ng, and M. I. Jordan, 2003: Latent dirichlet allocation. *Journal of machine Learning research*, **3**, no. Jan, 993–1022.

[5] Bosu, A., J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, 2014: Identifying the characteristics of vulnerable code changes: An empirical study. *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, New York, NY, USA, FSE 2014,

TABLE 13 The technique for which we observe the highest median prediction performance for each dataset.

	MIR	MOZ	OST	WIK
Acc.	NB,TFIDF,TFM	RF,BOW,DE	NB/LR,BOW,DE	RF,BOW,DE
AUC	RF,BOW,DE	RF,BOW,DE	NB,BOW,TFM	NB,BOW,DE
F-Mea.	RF,BOW,DE	RF,BOW,DE	LR/RF,TFIDF,DE	RF,BOW,DE
G-Mean	RF,TFIDF,DE	RF,BOW,DE	RF,BOW,DE	RF,BOW,DE
Pre.	NB,TFIDF,TFM	LR,TFIDF,TFM	NB,TFIDF,TFM	NB,TFIDF,TFM
Recall	LR,TFIDF,DE	RF,BOW,DE	LR,TFIDF,DE	LR,TFIDF,DE

257–268.
URL http://doi.acm.org/10.1145/2635868.2635880

[6] Breiman, L., 2001: Random forests. *Machine Learning*, **45**, no. 1, 5–32, doi:10.1023/A:1010933404324.
URL http://dx.doi.org/10.1023/A:1010933404324

[7] Breiman, L., J. Friedman, R. A. Olshen, and C. J. Stone, 1984: *Classification and Regression Trees*. 1stnd ed., Chapman & Hall, New York, 358 pp.
URL http://www.crcpress.com/catalog/C4841.htm

[8] Cito, J., P. Leitner, T. Fritz, and H. C. Gall, 2015: The making of cloud applications: An empirical study on software development for the cloud. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, New York, NY, USA, ESEC/FSE 2015, 393–403.
URL http://doi.acm.org/10.1145/2786805.2786826

[9] Cohen, J., 1960: A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, **20**, no. 1, 37–46, doi:10.1177/001316446002000104.
URL http://dx.doi.org/10.1177/001316446002000104

[10] der Bent, E. V., J. Hage, J. Visser, and G. Gousios, 2018: How good is your puppet? an empirically defined and validated quality model for puppet. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 164–174.

[11] El-Koka, A., K. H. Cha, and D. K. Kang, 2013: Regularization parameter tuning optimization approach in logistic regression. *2013 15th International Conference on Advanced Communications Technology (ICACT)*, 13–18.

[12] Escobar-Avila, J., E. Parra, and S. Haiduc, 2017: Text retrieval-based tagging of software engineering video tutorials. *Proceedings of the 39th International Conference on Software Engineering Companion*, IEEE Press, Piscataway, NJ, USA, ICSE-C '17, 341–343.
URL https://doi.org/10.1109/ICSE-C.2017.121

[13] Florea, A.-C., J. Anvik, and R. Andonie, 2017: *Spark-Based Cluster Implementation of a Bug Report Assignment Recommender System*, Springer International Publishing, Cham. 31–42.
URL https://doi.org\/10.1007\/978-3-319-59060-8_4

[14] Freedman, D., 2005: *Statistical Models : Theory and Practice*. Cambridge University Press.

[15] Fu, W., T. Menzies, and X. Shen, 2016: Tuning for software analytics: Is it really necessary? *Information and Software Technology*, **76**, 135 – 146, doi:https://doi.org/10.1016/j.infsof.2016.04.017.
URL http://www.sciencedirect.com/science/article/pii/S0950584916300738

[16] Gegick, M., P. Rotella, and T. Xie, 2010: Identifying security bug reports via text mining: An industrial case study. *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, 11–20.

[17] Ghotra, B., S. McIntosh, and A. E. Hassan, 2015: Revisiting the impact of classification techniques on the performance of defect prediction models. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, IEEE Press, Piscataway, NJ, USA, ICSE '15, 789–800.
URL http://dl.acm.org/citation.cfm?id=2818754.2818850

[18] Guyon, I. and A. Elisseeff, 2003: An introduction to variable and feature selection. *J. Mach. Learn. Res.*, **3**, 1157–1182.
URL http://dl.acm.org/citation.cfm?id=944919.944968

[19] Hall, T., S. Beecham, D. Bowes, D. Gray, and S. Counsell, 2012: A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, **38**, no. 6, 1276–1304, doi:10.1109/TSE.2011.103.

[20] Harris, Z. S., 1954: Distributional structure. *WORD*, **10**, no. 2-3, 146–162, doi:10.108000437956.1954.11659520.

[21] Hata, H., O. Mizuno, and T. Kikuno, 2010: Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering*, **15**, no. 2, 147–165, doi:10.1007/s10664-009-9117-9.
URL https://doi.org/10.1007/s10664-009-9117-9

[22] Herzig, K., S. Just, and A. Zeller, 2013: It's not a bug, it's a feature: How misclassification impacts bug prediction. *2013 35th International Conference on Software Engineering (ICSE)*, 392–401.

[23] Hovsepyan, A., R. Scandariato, W. Joosen, and J. Walden, 2012: Software vulnerability prediction using text analysis techniques. *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, ACM, New York, NY, USA, MetriSec '12, 7–10.
URL http://doi.acm.org/10.1145/2372225.2372230

[24] Humble, J. and D. Farley, 2010: *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1stnd ed., Addison-Wesley Professional.

[25] Hummer, W., F. Rosenberg, F. Oliveira, and T. Eilam, 2013: Automated testing of chef automation scripts. *Proceedings Demo:38; Poster Track of ACM/IFIP/USENIX International Middleware Conference*, ACM, New York, NY, USA, MiddlewareDPT '13, 4:1–4:2.
URL http://doi.acm.org/10.1145/2541614.2541632

[26] IEEE, 2010: Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, 1–23, doi:10.1109/IEEESTD.2010.5399061.

[27] Jiang, Y. and B. Adams, 2015: Co-evolution of infrastructure and source code: An empirical study. *Proceedings of the 12th Working Conference on Mining Software Repositories*, IEEE Press, Piscataway, NJ, USA, MSR '15, 45–55.
URL http://dl.acm.org/citation.cfm?id=2820518.2820527

[28] Keller, G., 2012: *Incident documentation20160204-Phabricator*. https://jumpcloud.com\/blog\/why-user-management-in-chef-and-puppet-is-a-mistake/, [Online; accessed 10-October-2017].

[29] Krishna, R., Z. Yu, A. Agrawal, M. Dominguez, and D. Wolf, 2016: The "bigse" project: Lessons learned from validating industrial text mining. *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*, ACM, New York, NY, USA, BIGDSE '16, 65–71.
URL http://doi.acm.org/10.1145/2896825.2896836

[30] Labs, P., 2017: *Puppet Documentation*. https://docs.puppet.com/, [Online; accessed 10-October-2017].

[31] Landis, J. R. and G. G. Koch, 1977: The measurement of observer agreement for categorical data. *Biometrics*, **33**, no. 1, 159–174.
URL http://www.jstor.org/stable/2529310

[32] Lessmann, S., B. Baesens, C. Mues, and S. Pietsch, 2008: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, **34**, no. 4, 485–496, doi:10.1109/TSE.2008.35.
URL http://dx.doi.org/10.1109/TSE.2008.35

[33] Manning, C. D., P. Raghavan, and H. Schütze, 2008: *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

[34] McCallum, A. and K. Nigam, 1998: A comparison of event models for naive Bayes text classification. *Learning for Text Categorization: Papers from the 1998 AAAI Workshop*, 41–48.
URL http://www.kamalnigam.com/papers/multinomial-aaaiws98.pdf

[35] McCune, J. T. and Jeffrey, 2011: *Pro Puppet*. 1nd ed., Apress, 336 pp.
URL https://www.springer.com/gp/book/9781430230571

[36] Menzies, T., A. Dekhtyar, J. Distefano, and J. Greenwald, 2007: Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'". *IEEE Trans. Softw. Eng.*, **33**, no. 9, 637–640, doi:10.1109/TSE.2007.70721.
URL http://dx.doi.org/10.1109/TSE.2007.70721

[37] Mikolov, T., I. Sutskever, K. Chen, G. Corrado, and J. Dean, 2013: Distributed representations of words and phrases and their compositionality. *Proceedings of the 26th International Conference on Neural Information Processing Systems*, Curran Associates Inc., USA, NIPS'13, 3111–3119.
URL http://dl.acm.org/citation.cfm?id=2999792.2999959

[38] Mizuno, O., S. Ikami, S. Nakaichi, and T. Kikuno, 2007: Spam filter based approach for finding fault-prone software modules. *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 4–4.

[39] Munaiah, N., S. Kroh, C. Cabrey, and M. Nagappan, 2017: Curating github for engineered software projects. *Empirical Software Engineering*, 1–35, doi:10.1007/s10664-017-9512-6.
URL http://dx.doi.org/10.1007/s10664-017-9512-6

[40] Nunez-Varela, A. S., H. G. Perez-Gonzalez, F. E. Martinez-Perez, and C. Soubervielle-Montalvo, 2017: Source code metrics: A systematic mapping study. *Journal of Systems and Software*, **128**, 164 – 197, doi:https://doi.org/10.1016/j.jss.2017.03.044.
URL http://www.sciencedirect.com/science/article/pii/S0164121217300663

[41] Parnin, C., E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, 2017: The top 10 adages in continuous deployment. *IEEE Software*, **34**, no. 3, 86–95, doi:10.1109/MS.2017.86.

[42] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, 2011: Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, **12**, 2825–2830.
URL http://dl.acm.org/citation.cfm?id=1953048.2078195

[43] Perl, H., S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, 2015: Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, New York, NY, USA, CCS '15, 426–437.
URL http://doi.acm.org/10.1145/2810103.2813604

[44] Porter, M. F., 1997: Readings in information retrieval. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter An Algorithm for Suffix Stripping, 313–316.
URL http://dl.acm.org/citation.cfm?id=275537.275705

[45] Rahman, A., C. Parnin, and L. Williams, 2019: The seven sins: Security smells in infrastructure as code scripts. *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, to appear. Pre-print: https://akondrahman.github.io/papers/ist18_iac_sms.pdf.

[46] Rahman, A., A. Partho, D. Meder, and L. Williams, 2017: Which factors influence practitioners' usage of build automation tools? *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering*, IEEE Press, Piscataway, NJ, USA, RCoSE '17, 20–26.
URL https://doi.org/10.1109/RCoSE.2017..8

[47] Rahman, A., A. Partho, P. Morrison, and L. Williams, 2018: What questions do programmers ask about configuration as code? *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, ACM, New York, NY, USA, RCoSE '18, 16–22.
URL http://doi.acm.org/10.1145/3194760.3194769

[48] Rahman, A. and L. Williams, 2018: Characterizing defective configuration scripts used for continuous deployment. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 34–45.

[49] Rahman, A. A. U., E. Helms, L. Williams, and C. Parnin, 2015: Synthesizing continuous deployment practices used in software development. *Proceedings of the 2015 Agile Conference*, IEEE Computer Society, Washington, DC, USA, AGILE '15, 1–10.
URL http://dx.doi.org/10.1109/Agile.2015.12

[50] Rahman, F. and P. Devanbu, 2013: How, and why, process metrics are better. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press, Piscataway, NJ, USA, ICSE '13, 432–441.
URL http://dl.acm.org/citation.cfm?id=2486788.2486846

[51] Ray, B., V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, 2016: On the "naturalness" of buggy code. *Proceedings of the 38th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE '16, 428–439.
URL http://doi.acm.org/10.1145/2884781.2884848

[52] Scandariato, R., J. Walden, A. Hovsepyan, and W. Joosen, 2014: Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, **40**, no. 10, 993–1006, doi:10.1109/TSE.2014.2340398.

[53] Shambaugh, R., A. Weiss, and A. Guha, 2016: Rehearsal: A configuration verification tool for puppet. *SIGPLAN Not.*, **51**, no. 6, 416–430, doi:10.1145/2980983.2908083.
URL http://doi.acm.org/10.1145/2980983.2908083

[54] Sharma, T., M. Fragkoulis, and D. Spinellis, 2016: Does your configuration code smell? *Proceedings of the 13th International Conference on Mining Software Repositories*, ACM, New York, NY, USA, MSR '16, 189–200.
URL http://doi.acm.org/10.1145/2901739.2901761

[55] Stol, K.-J., P. Ralph, and B. Fitzgerald, 2016: Grounded theory in software engineering research: A critical review and guidelines. *Proceedings of the 38th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE '16, 120–131.
URL http://doi.acm.org/10.1145/2884781.2884833

[56] Storn, R. and K. Price, 1997: Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, **11**, no. 4, 341–359, doi:10.1023/A:1008202821328.
URL http://dx.doi.org/10.1023/A:1008202821328

[57] Strauss, A. and J. Corbin, 1998: *Basics of qualitative research techniques*. Sage publications.

[58] Tan, P.-N., M. Steinbach, and V. Kumar, 2005: *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[59] Tantithamthavorn, C., S. McIntosh, A. E. Hassan, and K. Matsumoto, 2016: Automated parameter optimization of classification techniques for defect prediction models. *Proceedings of the 38th International Conference on Software Engineering*, ACM, New York, NY, USA, ICSE '16, 321–332.
URL http://doi.acm.org/10.1145/2884781.2884857

[60] Walden, J., J. Stuckman, and R. Scandariato, 2014: Predicting vulnerable components: Software metrics vs text mining. *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 23–33.

[61] Weiss, A., A. Guha, and Y. Brun, 2017: Tortoise: Interactive system configuration repair. *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, Piscataway, NJ, USA, ASE 2017, 625–636.
URL http://dl.acm.org/citation.cfm?id=3155562.3155641

[62] Zhang, F., A. Mockus, I. Keivanloo, and Y. Zou, 2016: Towards building a universal defect prediction model with rank transformed predictors. *Empirical Softw. Engg.*, **21**, no. 5, 2107–2145, doi:10.1007/s10664-015-9396-2.
URL http://dx.doi.org/10.1007/s10664-015-9396-2

[63] Zhang, H., 2004: The optimality of naive bayes. *In FLAIRS2004 conference.*

9 | AUTHOR BIOGRAPHY

Akond Rahman. Akond Rahman is a PhD candidate at North Carolina State University. His research interests include DevOps, Software Security, and Applied Software Analytics. He graduated with a M.Sc. in Computer Science and Engineering from University of Connecticut and a B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology.

Laurie Williams. Laurie Williams is the Interim Department Head of Computer Science and a Professor in the Computer Science Department of the College of Engineering at North Carolina State University (NCSU). Laurie is a co-director of the NCSU Science of Security Lablet sponsored by the National Security Agency. Laurie's research focuses on software security; agile software development practices and processes; software reliability, and software testing and analysis.