Within-project Defect Prediction of Infrastructure-as-Code Using Product and Process Metrics

Stefano Dalla Palma*, Dario Di Nucci*, Fabio Palomba†, and Damian A. Tamburri‡
*Tilburg University, Jheronimous Academy of Data Science, Netherlands
†Software Engineering (SeSa) Lab — University of Salerno, Italy
‡Eindhoven University of Technology, Jheronimous Academy of Data Science, Netherlands

Abstract—Infrastructure-as-code (IaC) is the DevOps practice enabling management and provisioning of infrastructure through the definition of machine-readable files, hereinafter referred to as *IaC scripts*. Similarly to other source code artefacts, these files may contain defects that can preclude their correct functioning. In this paper, we aim at assessing the role of *product* and *process* metrics when predicting defective IaC scripts. We propose a fully integrated machine-learning framework for IaC Defect Prediction, that allows for repository crawling, metrics collection, model building, and evaluation. To evaluate it, we analyzed 104 projects and employed five machine-learning classifiers to compare their performance in flagging suspicious defective IaC scripts. The key results of the study report RANDOM FOREST as the best-performing model, with a median AUC-PR of 0.93 and MCC of 0.80. Furthermore, at least for the collected projects, product metrics identify defective IaC scripts more accurately than process metrics. Our findings put a baseline for investigating IaC Defect Prediction and the relationship between the product and process metrics, and IaC scripts' quality.

Index Terms—Infrastructure-as-code; Defect Prediction; Empirical Software Engineering.

1 INTRODUCTION

Modern software relies on the use of automation at both development and operations levels, an engineering strategy known as DevOps [1]; the software code driving such automation is collectively known as Infrastructure-as-Code (IaC). Infrastructure-as-Code *"promotes managing the knowledge and experience inside reusable scripts of infrastructure code instead of traditionally reserving it for the manual-intensive labor of system administrators, typically slow, time-consuming, effort-heavy, and often even error-prone"* [2].

IaC tactic emphasizes consistent, repeatable routines for provisioning, deploying, and orchestrating systems. Infrastructure managers can build systems that automatically change the infrastructure. Specifically, they can develop, test, and deploy these systems using the same engineering practices and tooling proven effective for application development, such as version control systems and automated testing. On the one hand, this requires an infrastructure to cope with continuous and rapid changes common in established software development practices, like Agile development. On the other hand, this opens the door to exploit development practices such as test-driven development, continuous integration, and continuous delivery.

Although IT infrastructures are constantly evolving and growing in size and complexity, little is known concerning how to maintain best, speedily evolve, and continuously improve infrastructure code, and yet it is picking up more and more traction in different domains [3]. This is problematic for organizations where software is essential. Infrastructure failures are even more demanding in environments where IT systems are more than just business-critical and where there is no tolerance for downtime. For example, Amazon's systems handle hundreds of millions of dollars in transactions every day. In that context, only in 2012, the estimated average cost of one-minute service downtime for Amazon alone was $66,000[1], even after extensive manual and semi-automatic service continuity practices such as service hot stand-by [4], or elastic provisioning [5].

Software Defect Prediction [6] supports the software development life-cycle testing process by identifying the parts of the system that are failure-prone and require extensive testing. As shown by previous research in the field, like any other source code artifact, infrastructure configuration management scripts can be failure-prone [7], [8], [9]. The effective prediction of failure-prone IaC scripts may enable organizations embracing the DevOps methodology to focus on such critical scripts during Quality Assurance activities and allocate effort and resources more efficiently.

Therefore, this work aims *to help software practitioners prioritize their inspection efforts for IaC scripts by proposing prediction models of failure-prone IaC scripts and investigating the role of product and process metrics for their prediction*.

To this end, we propose the RADON FRAMEWORK FOR IAC DEFECT PREDICTION, a fully integrated Machine-Learning-based framework that allows for repository crawling, metrics collection, model building, and evaluation. The HORIZON-2020 RADON project (https://radon-h2020.eu/) [10] aims to unlock the benefits of serverless Function-as-a-Service (FaaS) for the European software industry by

1. As determined by Forbes based on Amazon's 2012 net sales

developing a model-driven DevOps framework for creating and managing applications based on serverless computing. RADON applications consist of fine-grained and independent microservices that can efficiently and optimally exploit FaaS and container technologies. In this context, our framework for IaC Defect Prediction strives to tackle correctness in designing such applications.

The framework assessment led to the definition of several research questions, namely:

RQ$_1$ *To what extent does the classifier selection impact the performance of Machine-Learning models to **predict the failure-proneness of IaC scripts**?*

RQ$_2$ *How is the prediction performance affected by the choice of the **metric sets**?*

RQ$_3$ *Which metrics are **good defect predictors**? That is, what are the most selected predictors and their combinations?*

RQ$_1$ aims at identifying the effect that the choice of classifiers (e.g., Naive Bayes and Random Forest) has on the prediction performance. We gathered a comprehensive and meaningful set of failure-prone IaC scripts and metrics to implement and assess different classifiers for predicting the failure-proneness of an IaC script. Afterward, we compared their performance and focused on RANDOM FOREST as the best performing model. The contribution is a *set of classifiers suitable for the detection of suspicious failure-prone IaC scripts*. **RQ$_2$** aims at identifying the effect that the choice of metric sets (i.e., code and process metrics, and groups thereof) has on the prediction performance. Finally, **RQ$_3$** aims to identify and rank the measures that highly affect the prediction performance. A recursive feature selection method is performed to find the optimal number of features and to rank them according to their importance for the prediction. The contribution is *a set of metrics for the detection of suspicious failure-prone IaC scripts* that DevOps engineers and researchers can use to further understand and assess the quality of IaC scripts.

To evaluate the RADON FRAMEWORK FOR IAC DEFECT PREDICTION, we trained and tested five Machine-Learning techniques on 104 open-source Ansible-based projects. We focus on Ansible as (i) it is the most popular IaC language on GitHub to date[2] and in industry [11], and (ii) at the best of our knowledge, there is no previous work on predicting defects in the Ansible language.

Results show high prediction performance, with a median AUC-PR of 0.93 and an MCC of 0.80. Results also indicate that, at least for our data, product metrics identify defective IaC scripts more accurately than process metrics.

Contribution. The contribution of this work is the RADON FRAMEWORK FOR IAC DEFECT PREDICTION. We released a first implementation open-source on Github[3], along with (i) a dataset of 4,937 mined defect-fixing commits of Ansible scripts; (ii) a dataset of 4,434 mined Ansible scripts; (iii) 104 defect prediction models for Ansible, obtained through the framework, and their evaluation[4]. The shared material can be used as a baseline for DevOps and the research community for a better understanding of failure-prone IaC scripts and enable the comparison between competing approaches for defect prediction.

Structure of the paper. Section 2 presents background information on defect prediction and Infrastructure-as-Code focusing on Ansible. Section 3 describes the empirical framework of the study and reports details on the data collection. Section 4 describes the empirical study on Ansible code and outlines the methodology for each research question. Results are also discussed. Section 5 discusses the paper's insights, limitations, and threats to validity. Section 6 discusses the related literature. Finally, Section 7 concludes the paper and outlines future work.

2 BACKGROUND

This section provides a brief grounding and definitions on defect prediction and Infrastructure-as-Code.

2.1 DevOps and Infrastructure-as-Code

The DevOps methodology is radically changing the way software is designed and managed. DevOps entails adopting a set of organizational and technical practices, e.g., continuous integration, continuous deployment, blending development, and operation teams, to survive as an organization in the modern digital ecosystem and digital market, which demands fast and early releases, continuous software updates, constant evolution of market needs, and adoption of scalable technologies such as Cloud computing.

In this context, IaC is the DevOps practice of describing complex and (usually) Cloud-based deployments using machine-readable code. The main enabler for IaC has been the advent of Cloud computing, which has made the programmatic provisioning, configuration, and management of computational resources common practice.

Subsequently, many languages and platforms have been developed, each dealing with specific aspects of infrastructure management, from tools able to provision and orchestrate virtual machines (Cloudify, Terraform), to those doing a similar job for container technologies (Docker Swarm, Kubernetes), to machine image management tools (Packer), to configuration management tools (Chef, Ansible, Puppet). Ansible is gaining traction in the last years as a simple and agent-less (i.e., no master node) alternative to other more complex IaC technologies such as Chef and Puppet.

Ansible is an automation engine based on the YAML language that automates cloud provisioning, configuration management, and application deployment, among others. It works by connecting to nodes and pushing out scripts called *Ansible modules*, which describe or change the system state. Then, Ansible executes these modules when needed.

However, while modules allow for the proper functioning of Ansible scripts, *playbooks* make possible the orchestration of multiple slices of the infrastructure topology, with very detailed control on the scalability of the architecture (e.g., how many machines to tackle at a time). Playbooks are essential for configuration management and multi-machine deployment in Ansible. They can declare configurations and orchestrate steps of any manual ordered process by

2. Stemming from https://github.com/search using as search terms 'ansible', 'puppet' and 'chef'

3. https://github.com/radon-h2020/radon-defect-prediction-api

4. The dataset and material used to evaluate the framework is publicly available on Kaggle: https://www.kaggle.com/stefadp/ansibledefectsprediction

launching tasks within one or more *plays*. A play maps hosts to some well-defined roles, represented by Ansible *tasks* which in sum are calls to *Ansible modules*.

As an example, Figure 1 shows an Ansible code snippet representing a playbook that provisions and deploys a website.[5] To this aim, it configures various aspects such as the ports to open on the host container, the name of the user account, and the desired database to deploy. It first targets the web servers to ensure that the Apache server is at the latest version, and then the database servers to ensure that PostgreSQL is at the latest version and started. It achieves this by mapping the hosts (lines 2 and 13) to their respective tasks (lines 8-11, 17-20, 22-25). There, `yum` and `service` are modules to manage packages with the yum package manager and to control services on remote hosts, respectively; `name` (i.e., the name of the package and the database) and `state` (i.e., whether present, absent or otherwise) are parameters of these modules. In short, by composing a playbook of multiple plays, it is possible to orchestrate multi-machine deployments and run specific commands on the machines in the `webservers` and `databases` groups.

```
1  ---
2  - hosts: webservers
3    vars:
4      http_port: 80
5    remote_user: root
6
7    tasks:
8    - name: ensure apache is at the latest version
9      yum:
10       name: httpd
11       state: latest
12
13 - hosts: databases
14   remote_user: root
15
16   tasks:
17   - name: ensure postgresql is at the latest version
18     yum:
19       name: postgresql
20       state: latest
21
22   - name: ensure that postgresql is started
23     service:
24       name: postgresql
25       state: started
26
```

Fig. 1: An example of Ansible code.

2.2 Defect Prediction

From the defect prediction point of view, the research literature mainly focuses on standard applications (i.e., Java and C applications) and presents several approaches to identify the location of defective code using both supervised and unsupervised methods and in both within- and cross-project contexts [12]; a complete overview of state of the art is available in the systematic literature reviews conducted by Hall et al. [6] and Hosseini et al. [13].

5. Adapted from Ansible documentation: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html (Accessed April 2020)

In this work, we are interested in within-project defect prediction using supervised Machine-Learning techniques [14]. As such, the following concepts require to be introduced for the sake of understandability of the remaining of the paper:

- **Defect** - An imperfection or deficiency causing the IaC script not to meet its requirements or specifications (adapted from IEEE definition of defect [15]);
- **Defect-fixing commit** - A commit that takes an action related to a defect;
- **Defect-inducing commit** - A commit that contributed to introduce a defect;
- **Failure-prone script** - A script that presents defects and needs to be either repaired or replaced, as opposite of a *neutral* script;

Typically, the process for building supervised Machine-Learning-based defect predictors consists of generating instances from software archives such as version control systems, in the form of software components, source code files, classes, functions (or methods), and/or code changes (commits) according to the predefined prediction granularity. An instance is then characterized using several metrics (a.k.a., features) extracted from the software archives or computed afterward and is labeled as failure-prone or neutral, or with the number of defects it contains. The labeled instances are used to build a training set, namely the source of knowledge exploited by a Machine-Learning classifier to learn the features that discriminate and predict the presence (or number) of defects in a certain source code artifact. *In this study, we focus on the definition of binary classification models to classify IaC scripts as failure-prone or neutral.*

3 The RADON Framework for IaC Defect Prediction

Figure 2 provides a detailed overview of the proposed framework consisting of four individual components:

- The GitHub IaC Repositories Collector collects active IaC repositories on GitHub.
- The Repository Scorer computes repository metrics based on best engineering practices, which are used to select relevant repositories.
- The IaC Repository Miner mines failure-prone and neutral IaC scripts from a repository. Then, it gathers a broad set of metrics from the literature comprising of traditional application code metrics (e.g., lines of code), IaC-oriented metrics (e.g., number of configuration tasks), and process metrics (e.g., number of commits to a file), that are computed upon the collected IaC scripts to predict their failure-proneness.
- The IaC Defect Predictor pre-processes the datasets and trains the Machine Learning models. Given an unseen IaC script, this component classifies it as *failure-prone* or *neutral*.

The toolset and pipeline described along this section allowed us to build a meaningful dataset as our source of truth for the following steps of metrics calculation and defects prediction.

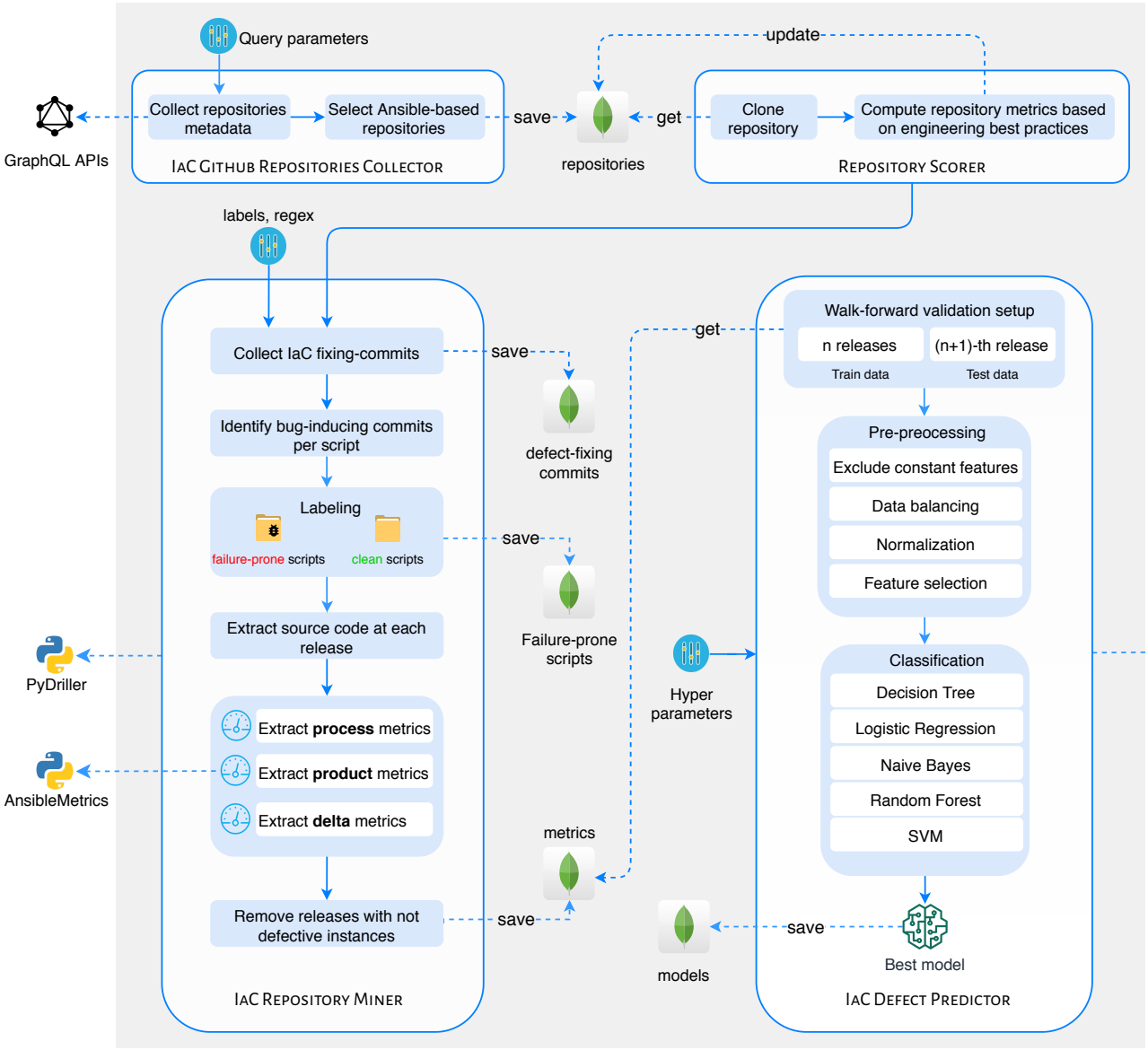


Fig. 2: The RADON framework for IaC Defect Prediction. The repositories collected by the GITHUB IAC REPOSITORIES COLLECTOR are passed as input to the REPOSITORY SCORER to pick relevant repositories. Afterward, the IAC REPOSITORY MINER mines the selected repositories, and its output, consisting of observations of *failure-prone* and *neutral* IaC scripts for the individual repositories, are used by the IAC DEFECT PREDICTOR to build and evaluate the models.

3.1 GITHUB IAC REPOSITORIES COLLECTOR

Prior work on defect prediction for traditional application code uses datasets extracted from public software data archives [16], [17]. However, only a handful of IaC datasets are publicly available [7], and they are limited to Chef and Puppet scripts. Therefore, we implemented the GITHUB IAC REPOSITORIES COLLECTOR to search for *public* candidate repositories containing Ansible code through the novel GraphQL-based GitHub APIs[6].

The tool is open-source and available on Github[7] and on the Python Package Index (PyPI)[8]. It enables tuning the GraphQL search query to collect metadata, such as name, description, URL, and root directories, from repositories that match user-defined selection criteria, such as the minimum number of releases, issues, stars, and watchers. Archived, mirrored, and forked repositories are excluded. The metadata are used to select Ansible repositories by checking the word `ansible` against their *name* (e.g., `ansible/ansible-examples`), *description* (e.g., *A few starter examples of ansible playbooks*[9]), or *directory layout* (i.e., the presence of at least two of the following directories: `playbooks`, `meta`, `tasks`, `handlers`, `roles`.)

The repositories' metadata are then saved on a MongoDB instance and analyzed to mine only relevant projects.

6. https://developer.github.com/v4/
7. https://github.com/radon-h2020/radon-repositories-collector
8. https://pypi.org/project/repositories-collector/
9. From https://github.com/ansible/ansible-examples

TABLE 1: Criteria to select repositories that contain evidence of engineered software projects.

#	Name	Description	Rationale
1	**Push Events**	The repository must have at least one *push event* to its default branch in the last six months	Evidence of recent development activity
2	**Releases**	The repository must have at least 2 releases	The proposed defect predictor analyzes files at each release and between successive releases
3	**Ratio of IaC Scripts**	At least 10% of the files must be IaC scripts	Repositories must have a sufficient number of IaC scripts
4	**Core Contributors**	The project must have at least 2 contributors whose total number of commits accounts for 80% or more of the total contributions	Evidence of collaboration
5	**Continuous Integration**	The repository must use a CI service, determined by the presence of a configuration file required by that service (e.g., a *.travis.yml* for TravisCI)	Evidence of quality
6	**Comment Ratio**	The *comment ratio* must be at least 0.1%	Evidence of maintainability
7	**Commit Frequency**	The average number of commits per month must be at least 2.0	Evidence of sustained evolution
8	**Issue Frequency**	The average number of issue events transpired per month must be at least 0.01	Evidence of project management
9	**License Availability**	The repository must have evidence of a *license*	Evidence of accountability evolution
10	**Lines of Code**	The repository must have at least 100 *lines of code*	Co-assess and control the criteria 4-9

3.2 Repository Scorer

A large portion of repositories on Github is not for software development, i.e., they are mainly used for experimentation, storage, and academic projects [18]. To mine only relevant projects, the RADON Framework for IaC Defect Prediction verifies several constraints reported in Table 1 along with their description and rationale.

Criterion 1 allows the crawler to discard inactive projects, while criterion 2 is needed because the target models are trained at the release-level. These criteria are evaluated by the Github IaC Repositories Collector, while the remaining criteria are evaluated by the Repository Scorer, a Python package that we implemented and made available open-source on Github[10] and PyPI[11].

More specifically, the *ratio of IaC Scripts* represents a cut-off to analyze repositories containing IaC scripts that have been determined by the previous works [7], [8]. Criteria 4-10 are considered as good indicators of well-engineered software projects, i.e., *"software projects that leverage sound software engineering practices in one or more of its dimensions such as documentation, testing, and project management"* [19].

The tool takes as input the local path (or remote URL) to a Git repository, calculate metrics related to the last eight criteria reported in Table 1, and saves them in the MongoDB instance along with the repositories' metadata.

Once the repository is deemed relevant for the analysis, based on the computed metrics, its history is analyzed using the IaC Repository Miner.

3.3 IaC Repository Miner

IaC Repository Miner relies on the PyDriller framework [20] to analyze the history of the projects and extract the *failure-prone* and *neutral* IaC scripts needed for the analysis. To this end, first it applies the Algorithm 1 to identify *defect-fixing* commits:

1) **Lines 4-7.** It extracts the commits linked to issues closed and related to bugs (i.e., with labels `bug`, `bugfix`, etc.). GitHub provides an issue tracker that links commits and corresponding issue reports, along with labels that are used to organize issues. IaC Repository Miner comes with 22 bug-related labels that were manually selected from those belonging to the Ansible repositories in Section 4. Examples include `critical-bug`, `ansible_bug`, `bug/bugfix`. The complete list is available on the online Appendix[12]
2) **Lines 9-12.** It analyses the commits whose messages indicate defective scripts. Specifically, as previously done by Zhang et al. [21], when analyzing the commits messages, it first removes all words ending with `bug` or `fix` (apart of *bugfix*), since those terms can be affixes of other words as "debug" and "prefix". A commit message is tagged as fixing defect if it matches the following regular expression that can be tuned by the user:

(*bug* | *fix* | *error* | *crash* | *problem* | *fail* | *defect* | *patch*)

Algorithm 1 Procedure to identify defect-fixing commits.

```
1:  procedure GETFIXINGCOMMITS(labels, regex)
2:      fixingCommits = []
3:
4:      for label in labels
5:          for issue in Repo.closedIssues(label)
6:              commit = commitClosingIssue(issue)
7:              fixingCommits.append(commit.sha)
8:
9:      for commit in Repo.commits
10:         commit.msg.remove(words ending with 'bug' or 'fix')
11:         if regex matches commit.msg
12:             fixingCommits.append(commit.sha)
13:
14:     discard commits that do not modify IaC scripts
15:     return fixingCommits
```

IaC Repository Miner keeps only the commits that modify at least one IaC script (lines 14). Afterward, it determines their failure-proneness as follows. First, it applies Algorithm 2 to identify IaC files touched by a defect-fixing commit and their *defect-inducing* commits (or, *bug-inducing* commit). We refer to those files as *fixed-files*. It analyzes

10. https://github.com/radon-h2020/radon-repository-scorer
11. https://pypi.org/project/repository-scorer/
12. https://github.com/stefanodallapalma/TSE-2020-05-0217/blob/master/LABELS.md

Algorithm 2 Procedure to identify IaC files modified in fixing-commits and their bug-inducing commits.

```
1:  procedure GETFIXEDFILES(fixingCommits:List[str])
2:      fixedFiles = []
3:
4:      for commit in fixingCommits[NEWEST : OLDEST]
5:          for file in commit.modifiedFiles
6:              if file.type != 'IaC' or file.changeType != 'Change'
7:                  continue
8:
9:              bics = SZZ(commit, file)        ▷ Bug-Inducing Commits
10:
11:             currentFix = FixedFile(file.filepath,
12:                                    fic=commit.sha,
13:                                    bic=bics[OLDEST])
14:
15:             if currentFix not in fixedFiles
16:                 fixedFiles.append(currentFix)
17:             else
18:                 existingFix = fixedFiles.get(currentFix)
19:                 if currentFix.fic is older than existingFix.bic
20:                     fixedFiles.append(currentFix)
21:                 else if currentFix.bic is older than existingFix.bic
22:                     existingFix.bic = currentFix.bic
23:
24:     return fixedFiles
```

the commits backward from the most recent to the oldest (line 4). For each *fixed-file* (line 5), it relies on the *SZZ* algorithm [22] to automatically identify the **oldest** commit that introduced a defect in that script (line 9).[13]

Files that have not already been fixed in the previously analyzed commits are added to the list of fixed-files (line 15-16). Otherwise, the following procedure applies:

- **Lines 19-20.** If the current commit (i.e., *fic* in Algorithm 2) is older than the previously fixed-file's defect-inducing commit (i.e., *bic* in Algorithm 2), then the current fixed-file is added to the list of fixed-files as a new object. This scenario is illustrated in Figure 3a. Here, a file has been fixed by a recent commit C10. It fixes a defect introduced in C8 (bic), newer than the current defect-fixing commit C4 (fic). Consequently, a new *FixedFile(file=A, fic=C4, bic=C1)* is added to the list of *fixed-files*, in addition to the previous *FixedFile(file=A, fic=C10, bic=C8)*.
- **Lines 21-22.** If the current commit is more recent than the previously fixed-file's defect-inducing commit, and the latter is more recent than the current defect-inducing commit, then the existing fixed-file's *bic* is updated with the current one. This scenario is illustrated in Figure 3b. Here, a previously analyzed commit C8 fixes a defect introduced in C5. So far, the file is considered failure-prone from C5 to C7. Nevertheless, the current commit C6 fixes a defect in the same file introduced in C4. Therefore, the file is considered failure-prone from C4 to C7, and the existing *FixedFile(file=B, fic=C8, bic=C5)* is updated to *FixedFile(file=B, fic=C8, bic=C4)*. This window is expanded if any fix to the same file is found in commits before C5.

Please, note that for the sake of clearness, we omitted the lines handling file-renaming in Algorithm 2. Once Algorithm 2 ends and the *fixed-files* list is returned, the labeling process is straightforward: all the snapshots of a fixed-file between its *bic* (inclusive) and the *fic* are labeled *failure-prone*.

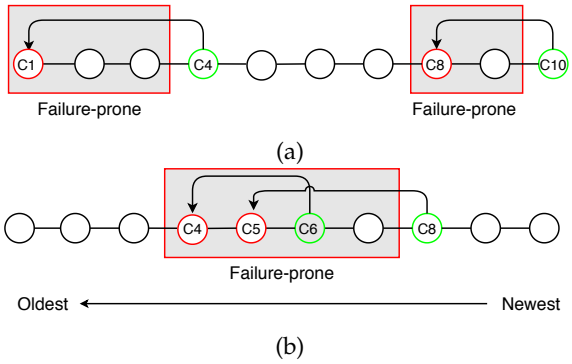


Fig. 3: Two scenarios of the labeling process.

To measure the soundness of the IaC Repository Miner in identifying *defect-fixing commits* of Ansible files, we uniformly selected and manually assessed a statistically-relevant sub-sample of the commits identified by Algorithm 1 on the data collected in Section 4. Therefore, we applied a sampling technique to determine the appropriate sample size of fixing-commits that meet our confidence level and confidence interval [23]. The population of Ansible defect-fixing commits was set to the number of data points in the dataset obtained in Section 4 (i.e., 4,937). For this research, an acceptable confidence level was set to the generally accepted value of 0.05. As a confidence interval (*a.k.a.*, the acceptable margin of error) the generally accepted 5% was chosen. By applying Cochran's formula [24], we obtained a sample size of 357 commits. Please note that Cochran's formula requires a parameter representing the population's estimated proportion with the attribute in question. Because we do not know the actual distribution of defect-fixing commits, we used the worst-case scenario percentage of 50%, which yields the largest sample size. We manually assessed the miner precision on the resulting 357 commits. The first author carried out the validation, while the second author validated 15% of the samples. Cohen's Kappa [25] was measured to compute the degree of agreement between the assessors. In disagreements, the two assessors met and discussed the disagreed commits to convey a solution, i.e., label it either as a true positive or a false positive. If no agreement was reached, the source was sent to the third author for further evaluation, without stating the two previous assessors' decisions, thus avoiding convenience bias. The resulting third-arbiter decision emerges as the final one, following a simple majority vote.

Following the aforementioned procedure, we obtained a precision of 74%. We reached a complete agreement in the resolution phase from an initial Cohen's Kappa of 0.34 (79% agreement). We applied the same procedure to evaluate the precision of Algorithm 2 in identifying bug-inducing commits. Also in that case, the population of bug-inducing commits was set to the number of data points in the dataset obtained in Section 4 (i.e., 4,434). Thus, we validated 354 bug-inducing commits and obtained a *precision* of 84%. We reached a complete agreement in the resolution phase from an initial Cohen's Kappa of 0.24 (79% agreement).

13. The IaC Repository Miner relies on the *SZZ* implemented in PyDriller 1.15.

TABLE 2: Configurations

Step	Options	Algorithm
Feature selection	Constant variables removal	`sklearn.feature_selection.VarianceThreshold`
	Recursive Feature Elimination	`sklearn.feature_selection.RFECV`
Data balancing	None	
	Random under sampling	`imblearn.under_sampling.RandomUnderSampler`
	Random over sampling	`imblearn.over_sampling.RandomOverSampler`
Data normalization	None	
	Min-max normalization	`sklearn.preprocessing.MinMaxScaler`
	Standardization	`sklearn.preprocessing.StandardScaler`
Classification	Decision Tree	`sklearn.tree.DecisionTreeClassifier`
	Logistic Regression	`sklearn.linear_model.LogisticRegression`
	Naive Bayes	`sklearn.naive_bayes.GaussianNB`
	Random Forest	`sklearn.ensemble.RandomForestClassifier`
	Support Vector Machine	`sklearn.svm.SVC`

Along with the failure-proneness of the IaC scripts, IAC REPOSITORY MINER gathers a comprehensive set of 108 features to train the defect prediction model whose description is reported in the online Appendix[14]. Such features have been extracted from previous work in defect prediction [26], [8], [27] and can be classified into three categories:

- **IaC-oriented (ICO) metrics** are structural properties derived from the analysis of the IaC source code. Hereinafter, we will use the terms ICO metrics and *product* metrics interchangeably. IAC REPOSITORY MINER *uses* the RADON ANSIBLEMETRICS tool [28] to collect the 46 metrics belonging to a recently proposed catalog of IaC quality metrics [27] that measure and potentially predict the maintainability of IaC scripts. More particularly, 8 of them are "traditional" code metrics that can be adapted for IaC scripts, e.g., the lines of code: these metrics have been long adopted in the context of traditional defect prediction and, therefore, we selected them to assess their role for predicting IaC defects. Other 14 metrics in this set have been already studied by Rahman and Williams [8] for predicting defects in Puppet code: as such, we considered them to verify their generalizability when employed for predicting Ansible defects. Finally, the last 24 refer to best and bad practices and data management of Ansible code: in our previous work [27], we conjectured that these metrics could negatively affect the maintainability of IaC code and increase its failure-proneness. As an example, let consider the number of distinct modules: IaC scripts consisting of many distinct modules are naturally less self-contained and potentially affect the complexity and maintainability of the system and, therefore, we considered it among the set of metrics for IaC defect prediction.
- **Delta metrics** capture the amount of change in a file between two successive releases. We collected 46 of such metrics, one for each IaC-oriented metric. Delta metrics have been associated with the failure-proneness of traditional source code by Arisholm et al. [29]. Again, we selected them to assess their usefulness when employed in the context of IaC scripts.
- **Process metrics** consider aspects concerning the development process rather than the code itself. IAC REPOSITORY MINER *extends* PyDriller[15] to collect 16 measures that include the number of developers that changed a file, the total number of added and removed lines, the number of files committed together. Also in this case, the selected metrics have been previously associated with the failure-proneness of traditional code [30], [26] and was our willingness to experiment with them in a different context.

These metrics are extracted from every IaC script at each release and saved on the MongoDB instance for their analysis or use by the IAC DEFECT PREDICTOR.

3.4 IAC DEFECT PREDICTOR

The IAC DEFECT PREDICTOR relies on the Python frameworks *scikit-learn* [31] and *imblearn* [32] to build the pipeline that balances and pre-processes the dataset, trains and validates the Machine-Learning models, and uses it to predict unseen instances. In particular, it uses different configurations in terms of feature selection, normalization, data balancing, classifiers, and hyper-parameters[16], as described below and in Table 2.

1) *Feature selection*. The data have been originally intended for defect prediction in our study. Nevertheless, not all the dataset features may help the task because they are constant or do not provide useful information exploitable by a learning method for a particular dataset. The RADON framework uses feature selection to reduce the dataset's size and speed-up the training, and select the optimal number of features that maximize a given performance criterion.
2) *Data balancing*. Class imbalance is a major obstacle for proper classification by supervised learning algorithms [33]. This is particularly true in defect prediction, where the *neutral* class outnumbers the *failure-prone* class. Balancing the training dataset is a well-known practice for supervised learning problems to overcome

14. For the sake of page limitation, we reported the complete list of metrics at https://github.com/stefanodallapalma/TSE-2020-05-0217/blob/master/METRICS.md

15. We used the implementation available online at release 1.13. The process metrics are documented at https://pydriller.readthedocs.io/en/latest/processmetrics.html

16. Given the number of classifiers and hyper-parameters, we preferred reporting the latter in the online appendix.

this obstacle. Therefore, once feature selection is finished, the training data are balanced such that the number of failure-prone instances equals the number of neutral instances. The RADON framework uses three configurations for balancing, namely (i) no balancing; (ii) random under-sampling of the majority class; and (iii) random over-sampling of the minority class. Both techniques (ii) and (iii) are implemented by the *imblearn* package in Python[17]. An attempt was made to balance data with other techniques (e.g., SMOTE, ADASYN, TomekLinks, etc.), but in most cases, their impact on the prediction was small compared to the increase in computational time. We, therefore, resorted to the faster random-sampling techniques.

3) *Data normalization*. In this step, the training data are normalized, scaling numeric attributes. The RADON framework uses three configurations for data normalization, namely (i) no normalization; (ii) *min-max* transformation to scale each feature individually in the range [0, 1]; (iii) *standardization* of the features by removing the mean and scaling to unit variance.
4) *Classification*. The normalized data and the learning algorithm are used to build the learner. Before the learner is tested, the original test data are normalized in the same way, and the dimensionality is reduced to the same subset of attributes from step 1. After comparing the predicted value and the actual value of the test data, the performance of one *pass* of validation is obtained. Note that, in our framework, the classification step can be applied with any machine learning algorithm, i.e., the learner selection is left to the user.

The final output consists of a *csv* file that reports the performance of the models for each validation step, and a *joblib* file for each trained model to persist them for future use without having to retrain.

4 A Large Empirical Study on Ansible code

This section reports the experimentation we conducted to investigate the role of IaC-Oriented, delta, and process metrics to predict defective IaC scripts. The *goal* is to evaluate the RADON Framework for IaC Defect Prediction in a within-project setup, with the *purpose* of early detecting defects in IaC scripts, and learning features that characterize them, from the individual projects considered.

The *quality focus* is on evaluating which classification techniques and features the framework should rely on to achieve the highest detection accuracy. The *perspective* is of researchers, who want to evaluate *in-vitro* the effectiveness of defect prediction applied to Infrastructure-as-Code. We share the dataset, metrics, and models as a baseline for DevOps and the research community to better understand (failure-prone) IaC scripts and compare competing approaches for defect prediction.[18]

Context. The *context* of the study is composed of configuration management systems and failure-prone IaC scripts.

17. https://imbalanced-learn.readthedocs.io/en/stable/api.html

18. Links to supplemental material are made available on the online Appendix at https://github.com/stefanodallapalma/TSE-2020-05-0217

TABLE 3: Number of projects discarded for each criterion.

IaC ratio	Core contributors	Comments ratio	Commit frequency	Issue frequency	LOC	CI	License
62 (6%)	406 (39%)	30 (3%)	421 (40%)	607 (58%)	21 (2%)	267 (25%)	136 (13%)

TABLE 4: Statistics on the number of defective instances for the 139 repositories containing defect-fixing commits, after applying the criteria in Table 1.

Quartile	Cum. # repos	Min	Mean	Std	Median	Max
1st	33	1	4	2	3	8
2nd	69	9	17	6	18	27
3rd	103	29	52	21	42	99
4th	139	101	314	367	175	1658

Specifically, we focus on software projects adopting Ansible for a two-folded reason: (i) Ansible is the most popular IaC language to date in industry [11], and (ii) at the best of our knowledge, there is no previous work on predicting defects in IaC scripts written in such a language.

We searched all the code repositories on GitHub containing Ansible code since 2014 with at least 2 releases and a push event to their default branch in the last six months[19]. Ansible has been developed since 2012, and we assume that two years is a reasonable amount of time for a new language to gain popularity. The total number of repositories related to our search query was 1050, and 850 were discarded from the dataset after applying the criteria depicted in Table 1, through the Repository Scorer. Table 3 shows the number of repositories discarded by each criterion. Note that the order of the criteria is not binding. Therefore, the sum of the repositories discarded by each policy is greater than the total number of repositories discarded. It is worth mentioning that the *Issue Frequency* criterion is zero when an organization uses private or external issues tracker. In that case, we manually investigated the repository to figure out whether to select it for experiments. Similarly, we applied the same process when all criteria were satisfied but the *Continuous Integration* or *License Availability*.

We then ran the IaC Repository Miner on the remaining 200 repositories. During its validation, we manually removed commits deemed false positives. We also removed commits that corrected typos in comments or messages, task names, and lint warnings such as deprecation. At this stage, 61 repositories ended up with no defect-fixing commits, and therefore they were discarded, leading to 139 repositories.

Finally, we selected the 106 repositories in the last three

19. Starting from March 2020.

TABLE 5: Statistics of the 104 analyzed repositories according to the inclusion criteria, and the third and fourth quartiles in Table 4. *License* and *Continuous integration* are not shown as boolean and *true* for all the repositories.

	Releases	IaC ratio	Core contributors	Comments ratio	Commit frequency	Issue frequency	LOC
Mean	51	67%	5	10%	26	1.50	7585
Std	77	15%	4	8%	86	2.93	14892
Min	2	19%	2	1%	2	0.01	165
Median	29	68%	4	8%	8	0.67	2570
Max	589	97%	18	45%	863	23.97	109565

quartiles of Table 4, in which defective instances range from 9 to 1658. The rationale is that repositories in the first quartile have a median of 3 defective instances. They would not practically allow the definition of a within-project model because even the best data balancing technique would have problems in generating artificial instances that are representative of the minority class [33].

Of the 106 repositories, two failed during training. Therefore, the following research questions are assessed on the remaining 104 repositories, whose statistics are depicted in Table 5.

Research Questions. In this paper, we aim at investigating *the role of Machine-Learning classifiers and a broad set of structural code and process measures for the prediction of failure-prone IaC scripts*. Specifically, the following research questions steer our research:

RQ$_1$ *To what extent does the classifier selection impact the performance of Machine-Learning models to **predict the failure-proneness of IaC scripts**?*

RQ$_2$ *How is the prediction performance affected by the choice of the **metric sets**?*

RQ$_3$ *Which metrics are **good defect predictors**? That is, what are the most selected predictors and their combinations?*

Classifier Selection. We relied on *five* classification algorithms, namely NAIVE BAYES (NB), LOGISTIC REGRESSION (LR), DECISION TREE (CART), RANDOM FOREST (RF), and SUPPORT-VECTOR MACHINE (SVM) as they have been widely used for defect prediction [13]. On the one hand, NAIVE BAYES and LOGISTIC REGRESSION are simple to understand and interpret and are fast learners, as they do not require much training data. On the other hand, DECISION TREE, RANDOM FOREST, and SUPPORT-VECTOR MACHINE are more flexible and powerful, as they are capable of fitting many functional forms and do not make assumptions about the underlying function. Although they are less efficient at training time due to many parameters, they can provide high-performance models. Furthermore, they also have a good level of interpretability. Decision Tree and Random Forest can be analyzed by observing the trees' path and the decision nodes to understand which feature affected the final decision. Support-Vector Machine provides a formula that is a weighted sum over features. The value of each feature can be analyzed to identify the extent to which it contributed to classifying the given instance. The five models complement each other in terms of pros and cons while keeping a good degree of explainability and are good candidates for our goals. All of the above techniques were implemented using scikit-learn[20], a well-known Python framework.

Model Validation. To compute the model performance, we reported the following evaluation measures, as we believe they are the most suitable for imbalanced data sets where one class is observed more frequently than the other class:

- **Area Under the Curve - Precision-Recall (AUC-PR).** This measure summarizes the precision-recall (PR) curve. On imbalanced or skewed data sets, PR curves are a useful alternative to ROC curves to highlight the performance differences that are lost in ROC curves [34], [35]. Please consider that we used AUC-PR to tune the models when applying cross-validation.
- **Matthews Correlation Coefficient (MCC).** This measure focuses on the quality of binary classifications. A coefficient of +1 represents a perfect prediction, 0 no better than a random prediction, and -1 indicates total disagreement between prediction and observation [36].

The performance is analyzed in terms of mean and standard deviation. A more comprehensive table containing all the evaluation measures for each project can be found in our online Appendix[21].

We evaluated the suitability and performance of the proposed classifiers by training them with different configurations in terms of normalization, data balancing, and hyper-parameters, as shown in Section 3.4. The model selection was guided by a randomized search on the models' parameters through a *walk-forward validation* [37]. In walk-forward, the dataset represents a time series that can be divided into orderable parts, e.g., a project's release. Such parts are chronologically ordered, and in each run, all data available before the part to predict is used as the train-set, while the part to predict is used as test-set, thus preventing the test-set from having data antecedent to the train-set. Afterward, the model performance is computed as the average among runs. The number of iterations is equal to the number of parts minus one. Specifically, we trained each model on the first n releases and tested on the (n+1)-th release, for every integer $n \in [1, |Releases|)$. This process is illustrated in Figure 4.

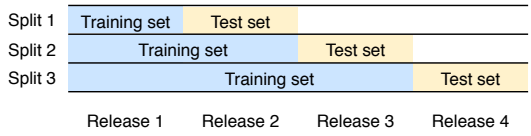


Fig. 4: Walk-forward release cross-validation.

4.1 RQ1 – Impact of Learning Techniques

The framework presented in Section 3 allowed us to gather a comprehensive and meaningful set of data to implement a learning-based method for predicting the failure-proneness of an IaC script, as discussed throughout this section, and answer the following research question:

RQ$_1$ – *To what extent does the classifier selection impact the performance of Machine-Learning models to **predict the failure-proneness of IaC scripts**?*

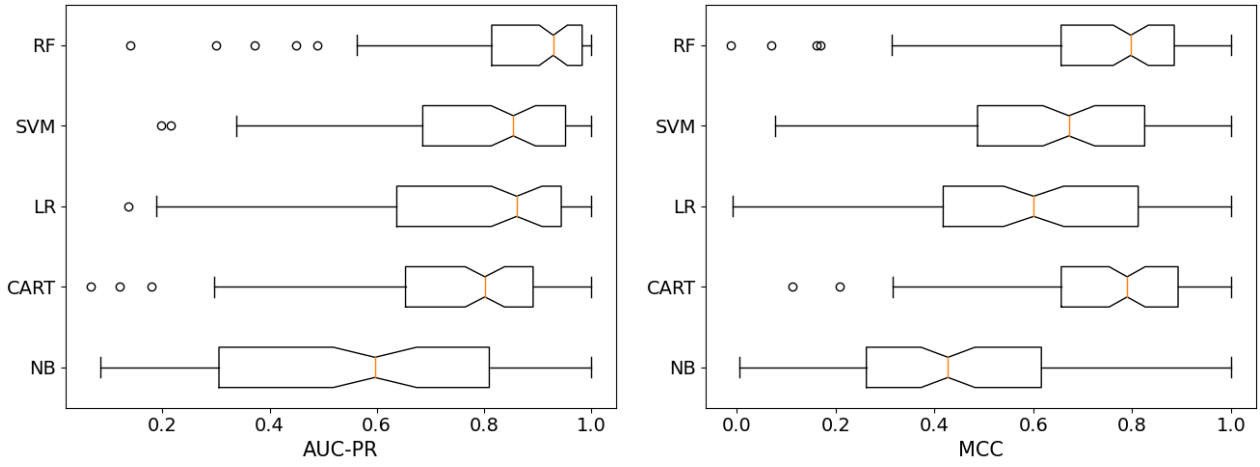
4.1.1 Methodology

We created 104 Machine-Learning models (one per project) trained and tested on different configurations, as shown in Section 3.4. At this step, we used all 108 metrics as features and decided, on purpose, not to use any feature selection algorithm as we were interested in analyzing the capabilities of the prediction models using the whole set of metrics, and considered only releases having at least one defective script.

First, we analyzed the number of times each evaluated classifier achieved the best performance (i.e., it was the best

20. https://scikit-learn.org/stable/supervised_learning.html

21. Available online at https://github.com/stefanodallapalma/TSE-2020-05-0217/blob/master/METRICS.md



(a) (b)

Fig. 5: (a) Area under the precision-recall curve and (b) Matthews correlation coefficient of each learning technique. The models trained using RANDOM FOREST perform statistically better than those relying on the remaining classifiers, both in AUC-PR and MCC. Follows, in order, SUPPORT VECTOR MACHINE, LOGISTIC REGRESSION, and DECISION TREE, albeit their performance difference is negligible. The only exception is that the DECISION TREE's MCC is very close to RANDOM FOREST, with no significant difference.

model in terms of AUC-PR) for a given project. Then, as it was difficult to make any assumptions about the underlying distribution with many evaluation measures, we applied a non-parametric test to assess the differences' significance. Specifically, for both AUC-PR and MCC, we reported *p-values* from a matched pair Wilcoxon's rank test [38] for all pairs of techniques, along with the *effect size* using Cohen's *d* [39]. Wilcoxon's rank test determines whether two or more sets of pairs are different from one another in a statistically significant manner; while Cohen's *d* measures the effect size for the comparison between two means. A Cohen's *d* below 0.2 is considered negligible, between 0.2 and 0.5 it is small, between 0.5 and 0.8 it is medium, and it is large above 0.8 [39]. Finally, given the number of comparisons being performed, we set the level of significance to $\alpha = 0.01$ and performed a posthoc Bonferroni's correction [40]. Specifically, we adjusted the significance level according to the number of comparisons (i.e., ten), therefore $\alpha = 0.001$. Finally, we reported several statistics (i.e., mean, median, minimum, maximum, standard deviation) for each evaluation measure for the classifier achieving the best performance.

4.1.2 Results

TABLE 6: Number of times a model appears among the best-performing models.

Learning technique	Occurrences
Random Forest	98
Support Vector Machine	26
Logistic Regression	17
Decision Tree	16
Naive Bayes	1

Table 6 shows each evaluated model's occurrences as the best model for any given project in terms of AUC-PR. As can be observed, RANDOM FOREST is the classifier that occurs most (98/104), followed by SUPPORT VECTOR MACHINE (26/104), LOGISTIC REGRESSION (17/104), and DECISION TREE (16/104). NAIVE BAYES has been observed among the best models only once. It is important to mention that the sum of the occurrences is not equal to 104 as, for some

TABLE 7: Statistical comparison of mean *AUC-PR* among learning techniques. Values below the diagonal are the differences between pairs of techniques (in %, significant in bold). A negative value means that the model in the row performed worse than the one in the column. Values above the diagonal are the effect size.

	RF	CART	SVM	LR	NB
					Effectsize(Wilcoxon (a=0.01))
RF	–	Medium	Small	Medium	Large
CART	**-12**	–	Small	Negligible	Medium
SVM	**-8**	5	–	Negligible	Large
LR	**-10**	**3**	-2	–	Large
NB	**-31**	**-19**	**-23**	**-21**	–

TABLE 8: Statistical comparison of mean *MCC* among learning techniques. Values below the diagonal are the differences between pairs of techniques (in %, significant in bold). A negative value means that the model in the row performed worse than the one in the column. Values above the diagonal are the effect size.

	RF	CART	SVM	LR	NB
					Effectsize(Wilcoxon (a=0.01))
RF	–	Negligible	Small	Medium	Large
CART	2	–	Medium	Large	Large
SVM	**-9**	**-12**	–	Small	Medium
LR	**-15**	**-16**	-5	–	Medium
NB	**-28**	**-30**	**-18**	**-13**	–

projects, multiple models achieved the same performance. In particular, we observed the co-occurrences (CART, LR, RF, SVM) four times, (LR, RF, SVM) three times, and the co-occurrences (CART, LR, NB, RF, SVM), (CART, RF, SVM), (CART, LR, RF), and (RF, SVM) once.

Figure 5a - Figure 5b and Table 7 - Table 8 show that the differences among the learning techniques in terms of mean AUC-PR and MCC are in most cases very high and of practical significance. The average PR area ranges from 0.56 for the worst-performing technique, namely NAIVE BAYES, to above 0.87 for the most performing techniques, namely RANDOM FOREST. RANDOM FOREST is the learning method with the lowest standard deviation and thus yields the most stable results regardless of the metrics used; the minimum is right below 0.14 while the maximum is 1.00, and the standard deviation 0.16.

Figure 5a and Figure 5b show a clear overview of the techniques' performance and differences. According to [41], although not a formal test, if two boxes' notches do not overlap, there is strong evidence (95% confidence) their medians differ. Thus, we observed that the median performance of RANDOM FOREST differs from the remaining techniques. In contrast, the notches' overlap between SUPPORT VECTOR MACHINE, LOGISTIC REGRESSION, and DECISION TREE suggests evidence that their medians performance do not differ significantly. The only exception is DECISION TREE's MCC, which is similar to RANDOM FOREST and significantly better than the remaining techniques. Significant differences between pairs of techniques are shown in bold in Table 7 - Table 8. Differences in AUC-PR between DECISION TREE and SUPPORT VECTOR MACHINE (5%) and between SUPPORT VECTOR MACHINE and LOGISTIC REGRESSION (2%) are not statistically significant ($p\text{-}value > 0.001$). The values above the diagonal indicate the effect size. The largest effect size, between RANDOM FOREST and NAIVE BAYES, is 1.40, meaning that the difference between the two means is larger than one standard deviation. The effect sizes between RANDOM FOREST and SUPPORT VECTOR MACHINE (0.44) and between SUPPORT VECTOR MACHINE and DECISION TREE (0.23) is small. The effect size between DECISION TREE and LOGISTIC REGRESSION (0.12) and between SUPPORT VECTOR MACHINE and LOGISTIC REGRESSION (0.09) are instead negligible.

Likewise, the average MCC ranges from 0.46 for the worst-performing technique, namely NAIVE BAYES, to approximately 0.75 for the best-performing technique, namely RANDOM FOREST. The latter classifier is also the learning technique which has the lowest standard deviation (~0.21) along with DECISION TREE (~0.18); the minimum is around 0, while the maximum is 1.00. In terms of MCC, the differences between RANDOM FOREST and DECISION TREE (1.8) and between SUPPORT VECTOR MACHINE and LOGISTIC REGRESSION (0.05) are not statistically significant. Similarly, the respective effect sizes (i.e., 0.09 and 0.21) are negligible and small, respectively.

The performances are generally very high, though we observed 20 projects where all the models perform badly, with a median AUC-PR of 0.45 (against the 0.90 of the remaining). Looking closer at the project's characteristics and their inclusion criteria in Table 1, we observed that the models are trained on a relatively lower number of releases (a median of 13 against the 32 for the projects with good models' performance). Because the models were trained and validated through a walk-forward validation across releases, a higher number of releases allows for more runs to train and validate the model to select the one that maximizes the AUC-PR. Furthermore, the median ratio of defective instances to the total instances differs by 13% between the two groups of projects: 7% and 20%, respectively. The lack of defective instances in the first group might have impacted the models' capabilities to discriminate between failure-prone and neutral instances, thus leading to bad performance.

Looking at the metrics, we observed a significant difference for the metric CHANGESETMAX (a median of 16 in the first group compared to 4 in the second), meaning that the 20 projects with poor models' performance have a median maximum number of files committed together of 16. Although our analysis considers only Ansible scripts, it is possible that in those projects, the presence of many Ansible files committed together led to an imprecise identification of failure-prone scripts, for the sole reason that a fixing-commit modified them along with the actual failure-prone file(s).

Concerning to the differences among the models, although the difference between SUPPORT VECTOR MACHINE and LOGISTIC REGRESSION are tiny, we observed 11 projects with a high difference in AUC-PR (i.e., a median difference of ~25%). In those projects, the former model outperformed the second 8 times. In those cases, the project's characteristics were similar. However, we observed that the median number of commits was higher (14) than the opposite (8). Similarly, the median number of instances used for training was almost three times higher in projects where SUPPORT VECTOR MACHINE performed better than LOGISTIC REGRESSION, while the percentage of defective instances to the total number of instances was approximately the same (~7%). This might suggest that LOGISTIC REGRESSION exploited better the projects with a small amount of data, but that might have been negatively affected by noise in the data present in larger projects. However, it is not clear how these attributes affected the models' performance. Also, the median CHANGESETMAX was notably lower in the first case (10) than in the second case (25). Similarly, the median CHANGESETAVG (i.e., the average number of files committed together) was 3 in the first case and 4 in the latter, suggesting that in those projects, SVM is less deteriorated by the noise in the dataset compared to LR. It is worth mentioning that, regardless of its importance for the final classification of IaC scripts' failure-proneness, the CHANGESET metric suggests that the dataset might contain several false positives, in the presence of which some models might perform worse than others.

Finally, RANDOM FOREST outperforms the other models, regardless of the metrics used or the project's characteristics. Therefore, we analyzed in detail the performance achieved by RANDOM FOREST, i.e., the best performing classifier according to our findings. Table 9 reports the average performance for all projects in terms of PR area, precision, recall, F1, and MCC. Although the minimum AUC-PR is 0.14, the mean and median are high: 0.87 and 0.93 respectively, with a standard deviation of 0.16, meaning that, on average, models' AUC-PR range between 0.71 and

1. Although the standard deviation might seem high, the coefficient of variation (CV) is only 0.18. As a rule of thumb, a $CV \geq 1$ indicates a relatively high variation, while a $CV < 1$ is considered low. It is worth noting that the high average MCC (0.74) indicates a strong agreement between the predictions and the observed values.

TABLE 9: Performance statistics of Random Forest across the 104 repositories.

	AUC-PR	MCC	Precision	Recall	F1
Mean	0.87	0.74	0.77	0.84	0.77
Std	0.16	0.21	0.21	0.16	0.20
Min	0.14	-0.01	0.00	0.00	0.00
Median	0.93	0.80	0.82	0.89	0.82
Max	1.00	1.00	1.00	1.00	1.00

RQ_1 summary: *The models trained using* RANDOM FOREST *perform statistically better than those relying on the remaining classifiers. The difference is statistically different with large effect size.*

4.2 RQ2 – Effect of Metric Sets

While using more features might lead to better performance because of the increased quality of information fed to the ML techniques, we are aware that using the whole set of metrics might decrease the detection performance as more features could increase noise. Furthermore, extracting all features could not be feasible in some projects or even cause more work in the metric collection. Therefore, in RQ_2, we aim at understanding the effect of different *type of metrics* on the prediction performance.

RQ_2 – *How is the prediction performance affected by the choice of the **metric sets**?*

4.2.1 Methodology

To evaluate the relative prediction power of the metric sets presented in Section 3.3 (*ICO*, *Delta* and *Process*), the data was combined to construct seven different candidate metric sets: *ICO*, *Delta*, *Process*, *ICO + Delta*, *ICO + Process*, *Delta + Process*, *Total*.

Then, as for RQ_1, to compare the magnitude of the differences between the metric sets, we reported several statistics, and we performed statistical analysis. In particular, for each evaluation measure, we showed the results of the matched pair Wilcoxon's rank test for all pairs of metrics sets aggregated across the best learning method from $\mathbf{RQ_1}$, i.e., RANDOM FOREST, along with the *effect size* computed using Cohen's d. Given the number of comparisons we performed, we set the significance level to $\alpha = 0.01$ and applied the posthoc Bonferroni's correction.

4.2.2 Results

Figure 6a - Figure 6b and Table 10-Table 11 show that the differences among the metric sets in terms of mean AUC-PR and MCC are in most cases very high and of practical significance. The average PR area ranges from 0.32, for the *Delta* metric set, up to 0.90 for the *ICO* metric set, i.e., consisting of solely Infrastructure-as-Code features.

The *ICO* metrics are also the metrics with the lowest standard deviation and thus yields the most stable results; the minimum is right above 0.23, and the maximum is 1.00, while the standard deviation 0.14. These results align with those observed in $\mathbf{RQ_1}$, suggesting that the results were due to those metrics. Indeed, there is a significant performance decrease (shown in bold in Table 10) when adding other metric sets to the *ICO* or using them alone.

Like $\mathbf{RQ_1}$, this can be caught at a glance from the notched boxplot analysis in Figure 6a and Figure 6b. On the one hand, the gap between the models' performance featuring *ICO* and those relying on *Process* or *Delta* metrics is evident. On the other hand, the notches' overlap between *ICO*, *ICO+Process*, *ICO+Delta*, and *Total* suggests that the models' medians performance using those features do not differ significantly. Consequently, adding process, delta metrics, or both to the IaC-Oriented metrics does not significantly affect the model's performance.

For example, adding the *Delta* metrics or the *Process* metrics significantly decreases the performance by 1% and 2% on average. However, the effect's magnitude is negligible. Therefore, those metrics do not contribute to improving the results. Although, at the same time, they do not worsen the performance significantly, using them would require an additional effort for metrics collection. Using the *Total* set of metrics decreases the performance similarly (i.e., 6% on average). Finally, the use of the *Process* and *Delta* metrics alone produce bad results, with a median AUC-PR of 0.31.

Likewise, the average MCC ranges from 0.07 for the worst metric set, namely *Delta*, to above 0.80 for the best metric set, namely *ICO*. The *ICO* metrics lead to a model with the third lowest standard deviation (~0.18, after the 0.13 of *Process* and *Delta* metrcs); the minimum *ICO*'s MCC is right above 0.12 while the maximum is 1.00, and its median is 0.92. Similarly to AUC-PR, the difference between the model trained on the *ICO* metric set is significantly higher than the ones trained on the other metric sets.

RQ_2 summary: *The models which feature IaC-Oriented metrics perform statistically better than those relying on the remaining metric sets. The difference is statistically significant with large effect size.*

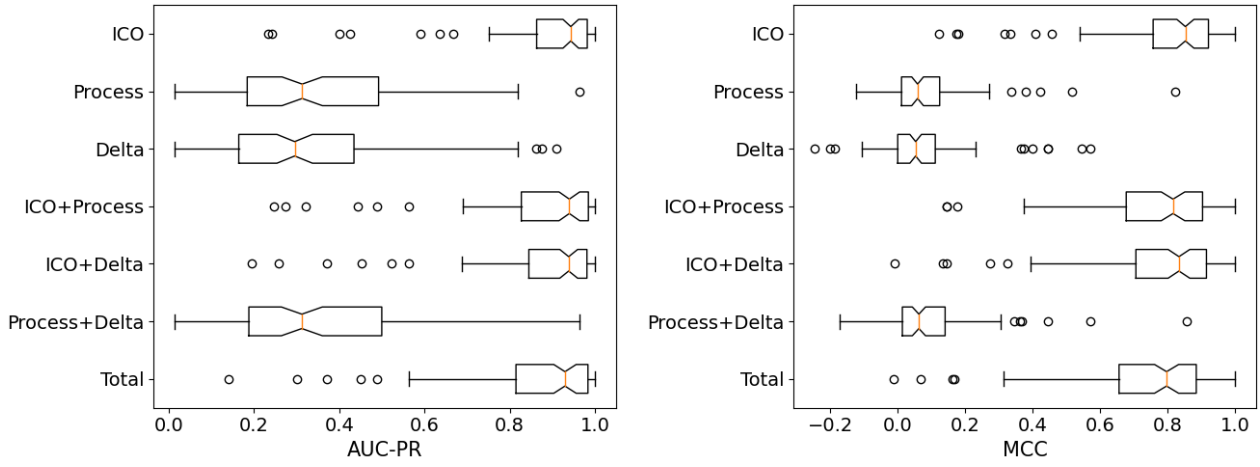
4.3 RQ3 – Best Predictors

In the previous research question, we observed that a model trained on the structural source code metrics performs better than one trained on the other metrics. While the previous question aimed to study how the prediction performance is affected by the choice of the metrics as grouped by their type, with the following research question, we aim at identifying and ranking the *individual metrics* based on their impact on the prediction performance.

RQ_3 – *Which metrics are **good defect predictors**? That is, what are the most selected predictors and their combinations?*

4.3.1 Methodology

To answer this research question, we performed a recursive feature selection to find the metrics that maximize the performance and to rank them according to their importance



(a) (b)

Fig. 6: (a) Area under the precision-recall curve and (b) Matthews correlation coefficient of each metric set. Models featuring IaC-Oriented metrics perform statistically better with a large effect size than those not relying on them, both in AUC-PR and MCC. Adding process, delta metrics, or both, does not significantly affect their performance. Models featuring process and delta metrics alone, or their combination, perform poorly with almost no difference.

TABLE 10: Statistics of mean *AUC-PR* among metric sets. Values below the diagonal are the differences between pairs of metric sets (in %, significant in bold). A negative value means that the model featuring the row's metrics performs worse than a model featuring the column's metrics. Values above the diagonal are the effect size.

	ICO	ICO + Delta	ICO + Process	Total	Process + Delta	Process	Delta
							Effectsize(Wilcoxon (a=0.01))
ICO	–	Negligible	Negligible	Negligible	Large	Large	Large
ICO + Delta	-1	–	Negligible	Negligible	Large	Large	Large
ICO + Process	**-2**	**-1**	–	Negligible	Large	Large	Large
Total	**-2**	-2	0	–	Large	Large	Large
Process + Delta	**-55**	**-54**	**-53**	**-53**	–	Negligible	Negligible
Process	**-55**	**-55**	**-54**	**-53**	0	–	Negligible
Delta	**-58**	**-58**	**-56**	**-56**	**-3**	**-3**	–

TABLE 11: Statistics of mean *MCC* among metric sets. Values below the diagonal are the differences between pairs of metric sets (in %, significant in bold). A negative value means that the model featuring row's metrics performs worse than a model featuring the column's metrics. Values above the diagonal are the effect size.

	ICO	ICO + Delta	ICO + Process	Total	Process + Delta	Process	Delta
							Effectsize(Wilcoxon (a=0.01))
ICO	–	Negligible	Negligible	Small	Large	Large	Large
ICO + Delta	**-3**	–	Negligible	Negligible	Large	Large	Large
ICO + Process	**-3**	-1	–	Negligible	Large	Large	Large
Total	**-6**	**-4**	-3	–	Large	Large	Large
Process + Delta	**-71**	**-7**	**-68**	**-65**	–	Negligible	Negligible
Process	**-72**	**-72**	**-69**	**-66**	-1	–	Negligible
Delta	**-73**	**-73**	**-69**	**-66**	-1	-1	–

for the prediction. Given an external estimator that assigns weights to features (e.g., the importance of each feature in a Random Forest model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features that optimize the performance criteria. The algorithm trains the estimator on the initial set of features and ranks the features by importance. The least important features are pruned from the current set. The procedure is recursively repeated on the pruned set until the algorithm eventually reaches the desired number of features to select.

However, RFE requires to select the number of features to keep, which is often not known in advance. To find the optimal number of features, we need to apply cross-validation to score the different feature subsets and select the best scoring collection of features. To this end, we used the *RFECV* method available in *sklearn*[22] along with the RANDOM FOREST model from $\mathbf{RQ}_1$ as estimator and *walk-forward* as cross-validation.

4.3.2 Result

The results of RFECV show a median of 11 optimal features per model, with a mean test PR area and standard deviation of 0.89 and 0.14, respectively. The average AUC-PR aligns with the results observed in $\mathbf{RQ}_1$. However, the lower

22. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html

TABLE 12: Ten most recurring features.

Rank	Predictor	Type	Occurrences
1	NUMTOKENS	ICO	84
1	TEXTENTROPY	ICO	84
1	LINESCODE	ICO	84
2	NUMKEYS	ICO	78
3	AVGTASKSIZE	ICO	64
4	LINESBLANK	ICO	58
5	NUMPARAMETERS	ICO	57
6	NUMUNIQUENAMES	ICO	55
7	NUMDISTINCTMODULES	ICO	54
8	NUMCONDITIONS	ICO	49

TABLE 13: Ten most recurring combinations of features.

Rank	Predictor	Occurrences
1	NUMTOKENS, LINESCODE	79
2	NUMTOKENS, TEXTENTROPY	76
3	LINESCODE, TEXTENTROPY	75
4	LINESCODE, NUMKEYS	72
4	NUMKEYS, TEXTENTROPY	72
4	LINESCODE, NUMTOKENS, TEXTENTROPY	72
5	NUMKEYS, NUMTOKENS	71
6	LINESCODE, NUMKEYS, NUMTOKENS	69
7	LINESCODE, NUMKEYS, TEXTENTROPY	68
8	NUMKEYS, NUMTOKENS, TEXTENTROPY	67

number of optimal features suggests that most of them are redundant and decrease the overall performance.

Table 12 and Table 13 show a ranked list of the most ten recurring features and their combinations. NUMTOKENS, TEXTENTROPY and LINESCODE are the three features that occur most among the features selected by the RFECV.

It is interesting to note that the most recurring process metrics, namely CHANGESETMAX, CHANGESETAVG and CODECHURNCOUNT only occur 42, 30 and 29 times, respectively, and are at rank 11, 17 and 18; while the most occurring delta metric is DELTATEXTENTROPY at rank 20 with 27 occurrences, and most of them are between rank 26 and 42 (i.e., the last rank).

RQ_3 summary: *IaC-oriented metrics tend to maximize the prediction performance. In particular,* NUMTOKENS, TEXTENTROPY *and* LINESCODE *are the most occurring predictors.*

5 DISCUSSION, IMPLICATIONS, AND LIMITATIONS

In $\mathbf{RQ}_1$, we found out that the collected metrics have a high prediction power for the failure-proneness of IaC. Regardless of the metrics used, RANDOM FOREST provided the best results in predicting, over-performing other learning techniques, reaching a median AUC-PR of 0.93, an MCC of 0.80, and an F1-score of 0.82.

We observed that the performance difference between SUPPORT VECTOR MACHINE and LOGISTIC REGRESSION, and between LOGISTIC REGRESSION and DECISION TREE is not statistically significant, although the former provided better results most of the time. Consequently, depending on the desired model flexibility and the available computational resources, one can choose them interchangeably without significantly negatively affecting the prediction.

The results achieved in the context of $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ are surprising. In $\mathbf{RQ}_2$, we found that models trained on process metrics have poor performance, as opposed to code metrics. In $\mathbf{RQ}_3$, we observed that the top 13 predictors (in terms of occurrences among the most critical features resulted from the recursive feature elimination) include IaC-Oriented metrics only. Therefore, we conclude that, for the collected Ansible-based projects, structural code metrics outperform process metrics, although the latter is often more effective when predicting the failure-proneness of source code instances in traditional Defect Prediction [30], [26]. We conjecture that this result is due to the lower number of infrastructure code changes than application code, limiting the information exploitable by the process metrics.

5.1 Implications for Research and Practice

- **Implications for researchers:** There is still room for further research in this area. Our findings put a baseline to investigate which prediction models should be used based on the characteristics of the software project to analyze (e.g., the number of core contributors, size in terms of commits, lines of code, and the ratio of IaC files). This aspect is of particular interest in the context of Cross-Project Defect Prediction, where the lack of historical data forces organizations to use pre-trained models built on similar projects. Further research is needed to understand the relationship between the failure-proneness of Infrastructure-as-Code and the collected metrics. These results can lead to a better understanding of which features to utilize to improve defect prediction of IaC.
- **Implications for practitioners:** Practitioners that still do not use prediction models for IaC can build upon our findings to implement novel models by extracting only subsets of features such as the ones that we showed in $\mathbf{RQ}_3$. This aspect will reduce the number of features to collect, reduce the number of tools required to mine them, and speed-up the training phase.
  For each project on which we trained our models, we report several statistics such as the size, number of commits, and core contributors to allow practitioners to compare their projects with those used in this study and use our pre-trained models.

5.2 Threats to Validity

This section describes the threats that can affect the validity of our study.

5.2.1 Threats to Construct validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the repositories we collected and the measurements we performed. Such a threat is the most occurring in our study, and it is related to:

- *Imprecise identification of relevant repositories*: it is possible that some repositories relevant for the analysis are missing or that some repositories are not relevant (e.g., too small for practical significance). We filtered out repositories based on ten criteria to identify active, IaC-related, and well-engineered software projects to mitigate this threat.

- *Imprecise identification of the failure-prone scripts*: it is possible that some scripts were imprecisely identified or not identified at all as "failure-prone". To mitigate this threat, we considered the developers' intent where possible (i.e., by analyzing issues' labels) and selected a state-of-the-art strategy to analyze the commit messages. Although using commit messages to mine defect information might be biased, Rahaman et al. [42] reported that increasing the sample size can leverage the possible bias in defect data. We also ignored renamed files without modification to the source code, added or removed by the defect-fixing commits. Besides, failure-prone scripts were extracted from a set of ~5,000 defect-fixing commits. Although our selection reflects a set that is statistically-relevant for our available data, we acknowledge this may not be the complete population of publicly available defect-fixing commits; we confer the validity of our study to the systematicity of the approach utilized for data collection, which warrants reasonable certainty that a large part of the available fixing-commits was considered.
- *Undocumented bugs present in the system*: we relied on the issue trackers to identify bugs fixed during the change history. Undocumented bugs (i.e., no issues related to the bug) could be present in some scripts, leading to classify failure-prone scripts as "neutral" mistakenly. To mitigate this threat, we took into account the commit messages, too.
- *Approximations due to identifying fix- and defect-inducing changes using keywords and the SZZ algorithm*: we relied on ad-hoc sets of issue labels, keywords to identify defect-fixing commits, and the SZZ implementation available in PyDriller to identify defect-inducing commits. A diverse combination of labels, keywords, and SZZ algorithm may bring different results. To the best of our knowledge, we selected the most common representative keywords used by developers to indicate fixes (while analyzing commit messages). Furthermore, we collected the labels related to bugs by manually analyzing the labels of each project.
- *Imprecise computation of process and code metrics*: poor implementation of the metrics might lead to the imprecise computation of their values, thus making the predictor wrongly believe a metric is more important than others. To mitigate this threat, we implemented the metrics following their documentation and adopting a test-driven approach. First, we documented the metrics with the intended behavior and examples. We implemented the test cases beforehand the production code to improve our confidence in the metric miners.

5.2.2 Threats to Internal Validity

Threats to *internal validity* concern internal factors we did not consider that could affect the investigated variables. In particular, the choice of the metrics (e.g., product vs. process metrics) might positively or negatively influence the classification. We mitigated this threat by considered a comprehensive set of metrics gathered from the literature comprising of

- metrics that take into account the structural properties of infrastructure code;
- metrics that track the changes between two successive releases for each of the structural metrics;
- complementary metrics that take into account the development process rather than the structural characteristics of the code.

The nature of data could hinder the accuracy of the models. In this context, *multicollinearity* is a condition where two or more independent variables are highly correlated. Although this aspect is particularly relevant for LOGISTIC REGRESSION, to guarantee a fair comparison with the other models, we did not take it into account. However, our results suggest that the multicollinearity effect could be significant. To assess this, we have conducted an additional analysis[23], where we reduce the multicollinearity by discarding the features having a Variable Inflation Factor (VIF) larger than 10 [43]. Our results show that the models where we apply VIF have a median MCC and AUC-PR 21% and 10% lower than those not including it; albeit, the effect's magnitude of these differences is negligible. The relatively small difference might indicate that our methodology did not influence too much the achieved results. Similarly, data balancing is a critical aspect of defect prediction. Class imbalance and the proposed framework's balancing techniques could affect the model's performance. Although we observed that *no balancing* is the balancing technique occurring the most (followed by Random Over-Sampling and Random Under-Sampling), we acknowledge that the impact of other state-of-the-art balancing techniques (e.g., SMOTE, ROSE, etc.) should be investigated as well. Nevertheless, we feel that the exhaustive analysis of the impact of feature selection and balancing techniques is out of scope for this study: a careful evaluation of such aspects would require dedicated studies, which we plan as future research.

5.2.3 Threats to External Validity

Threats to *external validity* concern the generalization of results. First, we analyzed 104 Ansible-based systems from different application domains and having different characteristics (number of contributors, size, number of commits, etc.). However, systems from different ecosystems (e.g., Chef, Puppet) and orchestration language (e.g., TOSCA) should be analyzed with the same framework to corroborate our findings. Our framework can be easily extended to other configuration management and orchestration languages. In particular, process metrics are language agnostic, and therefore they can be extracted from any versioning control system regardless of the language. Many of the structural metrics we collected can also be extended to other languages, as they have a general-scope and/or can be extracted by any plain text file (e.g., NUMTOKENS and TEXTENTROPY).

Second, our work revolves around within-project defect prediction, and, as such, we aim at learning features that characterize failure-prone IaC scripts from the individual projects considered. Repositories having no defects or a small number of defective instances were not used in this context: indeed, the absence of defects would not allow any machine-learner to distinguish failure-prone from failure-free scripts, while projects having a tiny amount of de-

23. Available on the online Appendix.

fective instances would not practically allow the definition of a within-project model because even the best data balancing technique would have problems in generating artificial instances that are representative of the minority class [33]. For these reasons, our framework might not generalize on projects with either zero or small amounts of defective instances. Those projects would be enforced to use a cross-projects strategy, where information is gathered from external projects so that a learner could be used in their own environments. Nevertheless, the investigation of the proposed framework performance in cross-project defect prediction is part of our future research.

Finally, another threat is related to the classifier selection. We chose five classifiers widely used in previous studies on bug prediction (e.g., [44], [45]).

5.2.4 Threats to Conclusion Validity

Threats to *conclusion validity* concern the relation between the treatment and the outcome. The metrics used to evaluate our defect prediction approach (i.e., AUC-ROC, precision, recall, F-Measure, and MCC) are widely used in the evaluation of the performances of defect prediction techniques [6], [13], [16], [17], [44]. Moreover, we used the AUC-PR, an alternative and more conservative measure than the AUC-ROC, to evaluate the models' overall performance for highly imbalanced problems.

Furthermore, since we needed to exploit change-history information to compute the metrics we collected, our study's evaluation design differs from the k-fold cross validation generally exploited while evaluating defect prediction techniques. In particular, we used the whole history of a system for the evaluation by adopting a walk-forward validation and assuring that new data (i.e., new releases) used to evaluate the model were never antecedent to those used to train it.

6 Related Work

IaC has recently received increasing attention in the research community, mainly due to the paradigm shift in software design and development. Various work relates to our study by empirically investigating the adoption, challenges, and particularly defects of IaC.

In the field of IaC defects and smells, various works have appeared in very recent years. Jiang and Adams [46] analyzed the co-evolution between infrastructure and production code, finding that the former is tightly coupled with test files, leading testers to change infrastructure specifications often when modifying tests.

Sharma et al. [47] looked for code smells in the source code of configuration management tools (e.g., Puppet, Chef). As a result, they proposed a catalog of 13 *implementation* and 11 *design* configuration smells. Then they benchmarked the catalog against 4,621 Puppet open source repositories. Interestingly, design smells showed higher average co-occurrence than implementation smells. That is, one wrong or non-optimal design decision introduces many quality issues in the future.

Rahman et al. [48] investigated the challenges in developing IaC, specifically in the context of configuration management tools. They looked for the questions that were more asked by programmers on Stack Overflow to help IaC developers. Also in this case, the focus was on Puppet-related questions. By applying qualitative analysis, they identified the three most common question categories: syntax errors, provisioning instances, and assessing the Puppet's feasibility to accomplish specific tasks. The three categories of questions that yielded the most unsatisfactory answers were installation, security, and data separation. The authors then classified IaC defects according to standard non-IaC defects categories by doing qualitative analysis of commit messages and issue report descriptions in open source projects, limited to Puppet code.

Furthermore, Rahman et al. [49] investigated the research challenges in IaC through a Systematic Literature Review. The main goal was to identify the various research areas surrounding the field of IaC. The four main topics that have been identified are (i) framework/tool for IaC; (ii) use of IaC; (iii) empirical studies related to IaC; and (iv) testing in IaC. They concluded that, while several studies exist on framework and tools, research in the context of IaC defects and security flaws is still at its early stages. Subsequently, Rahman et al. [50] proposed a catalog of seven security smells in IaC. Such smells were extracted from qualitative analysis of Puppet scripts in open source repositories. The identified smells comprise: (i) granting admin privileges by default; (ii) empty passwords; (iii) hard-coded secrets; (iv) invalid IP address binding; (v) suspicious comments (such as 'TODO' or 'FIXME'); (vi) use of HTTP without TLS; and (vii) use of weak cryptography algorithms. However, this is again limited to Puppet scripts, and not all smells are generalizable to other languages or tools. Later on, Van der Bent et al. [51] defined another measurement model to assess the quality of Puppet code.

Our work can be seen as complementary to those mentioned above, as it aims at studying the impact of structural code and process characteristics when predicting the failure-proneness of IaC scripts. For our goal, the closer articles are recently proposed by Rahman et al. [7], [8]. The former uses text mining techniques, such as bag-of-word, upon textual features extracted from IaC (Puppet) scripts and reports a median F-Measure of 73% for both techniques and three different datasets (Mozilla, OpenStack, and Wikimedia Commons). These results are not directly comparable with ours, being different datasets and languages. However, both approaches can be combined, leveraging both textual features and information from product and process metrics. The latter uses ten source code properties that correlate with defective IaC (Puppet) scripts to construct defect prediction models and reports a precision of 0.70~0.78 and a recall of 0.54~0.67. Our work is complementary to these two. First, rather than focusing on textual metrics, *we aim at deeply investigating the value of a broad set of structural and process metrics*; secondly, we focus on Ansible and not on Puppet.

7 Conclusion

We presented our approach for within-project defect prediction of Infrastructure-as-Code, based on product and process metrics. The approach is suited to work on IaC scripts, enabling us to analyze code and detect defects at the implementation level.

Although our study and implementation targets particularly Ansible, we believe the models may be easily portable to other languages (Chef, Puppet), as many of the most recurring features are general-purpose metrics.

Afterward, we compared five machine learning methods for defect prediction: Decision Tree, Logistic Regression, Naive Bayes, Random Forest, and Support Vector Machine. We crafted a dataset of publicly available Ansible-based IaC scripts extracted from relevant and active GitHub repositories to train the models. We classified scripts as failure-prone or neutral and then analyzed the predictive power of the models. Results show that RANDOM FOREST outperforms other models, with a median AUC-PR = 0.93.

Moreover, the dataset served also as a validation artifact for existing and upcoming approaches focusing on IaC.

Both DevOps and researchers can benefit from our contributions. Effective prediction of failure-prone IaC scripts enables organizations embracing the DevOps methodology to focus on such critical scripts during Quality Assurance and allocate effort and resources more efficiently [7]. Researchers gain valuable knowledge of novel IaC languages and tools, a comprehensive dataset and metrics set, and prediction models. Our work puts a clean baseline for existing and future approaches.

As future work, we plan to leverage the general-purpose metrics to extend our approach to configuration orchestration languages, starting from TOSCA [52], a YAML-based OASIS standard for defining infrastructure topologies. TOSCA was initially designed as an open standard for formatting templates, so tasks, such as cloud resource deployment and orchestration, could be translated into a generally readable form and become more portable across platforms. Overall, the standard aims to make it easier to update, extend or move cloud-based resources, thus opening up opportunities for building a universal defect-prediction framework for configuration orchestration languages. With such ultimate goal, we are also considering semi-supervised and unsupervised learning as our research agenda and their comparison with the proposed approach, and more generally to standard approaches based on supervised binary defect prediction.

ACKNOWLEDGMENT

Stefano, Dario, and Damian are supported by the European Commission grants no. 825040 (RADON H2020) and no. 825480 (SODALITE H2020). Fabio is partially supported by the Swiss National Science Foundation through the SNF Project No. PZ00P2 186090 (TED).

REFERENCES

[1] L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A Software Architect's Perspective.*, ser. SEI series in software engineering. Addison-Wesley, 2015.

[2] K. Morris, *Infrastructure as Code*. O'Reilly Media, Inc., 2016. [Online]. Available: https://www.oreilly.com/library/view/infrastructure-as-code/9781491924334/

[3] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "TOSCA Solves Big Problems in the Cloud and Beyond!" *IEEE Cloud Computing*, pp. 1–1, 2018.

[4] G. Ayyappan and S. Karpagam, "Analysis of a bulk service queue with unreliable server, multiple vacation, overloading and standby server." *Int. J. Math. Oper. Res.*, vol. 16, no. 3, pp. 291–315, 2020.

[5] S. Chaisiri, R. Kaewpuang, B. Lee, and D. Niyato, "Cost Minimization for Provisioning Virtual Servers in Amazon Elastic Compute Cloud," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011, pp. 85–95.

[6] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, 2012.

[7] A. Rahman and L. Williams, "Characterizing Defective Configuration Scripts Used for Continuous Deployment," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 34–45.

[8] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," *Information and Software Technology*, vol. 112, pp. 148 – 163, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584919300965

[9] A. Rahman, E. Farhana, and L. Williams, "The 'as code' activities: development anti-patterns for infrastructure as code," *Empirical Software Engineering*, 2020.

[10] G. Casale, M. Artač, W. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. P. A. Russo, S. Srirama *et al.*, "Rational Decomposition and Orchestration for Serverless Computing," in *The Symposium and Summer School on Service-Oriented Computing (SummerSoc)*, 2019.

[11] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[12] P. Paramshetti and D. Phalke, "Survey on Software Defect Prediction Using Machine Learning Techniques," *International Journal of Science and Research*, vol. 3, no. 12, pp. 1394–1397, 2014.

[13] S. Hosseini, B. Turhan, and D. Gunarathna, "A Systematic Literature Review and Meta-Analysis on Cross Project Defect Prediction," *IEEE Transactions on Software Engineering*, vol. 45, no. 2, pp. 111–147, 2019.

[14] J. Nam, "Survey on Software Defect Prediction," *Department of Compter Science and Engineerning, The Hong Kong University of Science and Technology, Tech. Rep*, 2014.

[15] "IEEE Standard Classification for Software Anomalies - Redline," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) - Redline*, pp. 1–25, Jan 2010.

[16] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "Automated Parameter Optimization of Classification Techniques for Defect Prediction Models," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 321–332.

[17] W. Fu, T. Menzies, and X. Shen, "Tuning for Software Analytics: Is it Really Necessary?" *Information and Software Technology*, vol. 76, pp. 135–146, 2016.

[18] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The Promises and Perils of Mining GitHub," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 92–101. [Online]. Available: https://doi.org/10.1145/2597073.2597074

[19] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[20] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python Framework for Mining Software Repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[21] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards Building a Universal Defect Prediction Model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 182–191. [Online]. Available: https://doi.org/10.1145/2597073.2597078

[22] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 81–90.

[23] J. Kotrlik, C. Higgins, and J. Bartlett, "Organizational Research: Determining Appropriate Sample Size in Survey Research," *Information technology, learning, and performance journal*, 2001.

[24] W. G. Cochran, *Sampling techniques*, 3rd ed. New York: John Wiley & Sons, 1977.

[25] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[26] F. Rahman and P. Devanbu, "How, and Why, Process Metrics Are Better," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 432–441.

[27] S. Dalla Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Toward a catalog of software quality metrics for infrastructure code," *Journal of Systems and Software*, vol. 170, p. 110726, 2020.

[28] S. Dalla Palma, D. Di Nucci, and D. A. Tamburri, "AnsibleMetrics: A Python library for measuring Infrastructure-as-Code blueprints in Ansible," *SoftwareX*, vol. 12, p. 100633, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711020303460

[29] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *J. Syst. Softw.*, vol. 83, no. 1, p. 2–17, Jan. 2010. [Online]. Available: https://doi.org/10.1016/j.jss.2009.06.055

[30] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 181–190.

[31] F. Pedregosa, 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, "Scikit-learn: Machine Learning in Python ," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[32] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[33] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, p. 20–29, Jun. 2004. [Online]. Available: https://doi.org/10.1145/1007730.1007735

[34] K. Boyd, K. H. Eng, and C. D. Page, "Area under the Precision-Recall Curve: Point Estimates and Confidence Intervals," in *Proceedings of the 2013th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*, ser. ECMLPKDD'13. Berlin, Heidelberg: Springer-Verlag, 2013, p. 451–466. [Online]. Available: https://doi.org/10.1007/978-3-642-40994-3_29

[35] M. Goadrich, L. Oliphant, and J. Shavlik, "Gleaner: Creating ensembles of first-order clauses to improve recall-precision curves," *Machine Learning*, vol. 64, no. 1-3, pp. 231–261, 2006.

[36] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442 – 451, 1975. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0005279575901099

[37] D. Falessi, J. Huang, L. Narayana, J. F. Thai, and B. Turhan, "On the need of preserving order of data when validating within-project defect classifiers," *Empirical Software Engineering*, 2020.

[38] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[39] J. Cohen, "The effect size index: d," *Statistical power analysis for the behavioral sciences*, vol. 2, pp. 284–288, 1988.

[40] E. W. Weisstein, "Bonferroni correction," *https://mathworld. wolfram. com/*, 2004.

[41] J. M. Chambers, *Graphical Methods for Data Analysis*. CRC Press, 2018.

[42] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, "Sample size vs. bias in defect prediction," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 147–157. [Online]. Available: https://doi.org/10.1145/2491411.2491418

[43] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied Linear Statistical Models*. Irwin Chicago, 1996, vol. 4.

[44] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, "A Developer Centered Bug Prediction Model," *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, p. 5–24, Jan. 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2659747

[45] D. Bowes, T. Hall, and J. Petrić, "Software Defect Prediction: Do Different Classifiers Find the Same Defects?" *Software Quality Journal*, vol. 26, no. 2, p. 525–552, Jun. 2018. [Online]. Available: https://doi.org/10.1007/s11219-016-9353-3

[46] Y. Jiang and B. Adams, "Co-evolution of Infrastructure and Source Code - An Empirical Study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 45–55.

[47] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does Your Configuration Code Smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 189–200.

[48] A. Rahman, A. Partho, P. Morrison, and L. Williams, "What Questions Do Programmers Ask About Configuration As Code?" in *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, ser. RCoSE '18. New York, NY, USA: ACM, 2018, pp. 16–22.

[49] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "Where Are The Gaps? A Systematic Mapping Study of Infrastructure as Code Research," *CoRR*, vol. abs/1807.04872, 2018.

[50] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, in Press.

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

[52] M. Rutkowski, C. Lauwers, C. Noshpitz, and C. Curescu, "TOSCA Simple Profile in YAML Version 1.3," 2019.