Replication of the paper "*Source code properties of defective infrastructure as code scripts*"

Ming Xiao Yuan
Department of Computer and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
ming-xiao.yuan@polymtl.ca

Reetesh Dooleea
Department of Computer and Software Engineering
Polytechnique Montreal
Montreal, QC, Canada
reetesh-chandranath.dooleea@polymtl.ca

***Abstract*—This replication effort seeks to corroborate the empirical conclusions drawn in the initial research paper entitled "Source code properties of defective infrastructure as code scripts," authored by Akhond Rahman and Laurie Williams. The study focuses on re-examining the three primary research questions proposed in the original paper using data mining and building a prediction model for defective Infrastructure as Code (IaC) scripts.**

***Index Terms*—Constructivist Grounded Theory, Extended commit message (XCM), Classification and Regression Tree (CART), K Nearest Neighbor (KNN), Logistic Regression (LR), Naive Bayes (NB), Random Forest (RF), Area Under the Receiver Operating Characteristic Curve (AUC), F-Measure**

I. Original study's statement

Infrastructure as Code (IaC) is pivotal in modern software engineering, particularly in continuous deployment environments. Previous research has identified certain source code properties that are strongly correlated with defects in IaC scripts. The objective of the original paper is to help practitioners increase the quality of IaC scripts through an empirical study that identifies source code properties of defective IaC scripts.

This study is an independent replication effort aimed at verifying the empirical findings of the research paper "*Source code properties of defective infrastructure as code scripts*" by Akhond Rahman and Laurie Williams. In this work, we try to answer the original research questions by applying data mining and building ML models as done by the authors to replicate the results. Furthermore, the GitHub repository used for the replication process can be found here.

For the first research question, we replicated sections 3.1.1 and 3.1.2 of the original work and then used the author's reported data and applied our analysis. Section 3.1.1 involves mining Github repositories from 4 organizations: Mirantis, Mozilla, OpenStack, and Wikimedia, and applying filtering criteria to obtain relevant repositories. Section 3.1.2 then processes the commit messages using Constructivist Grounded Theory to obtain a set of source code properties. We also employed the Mann-Whitney U test and Cliff's Delta to determine the validity of the results as the authors did.

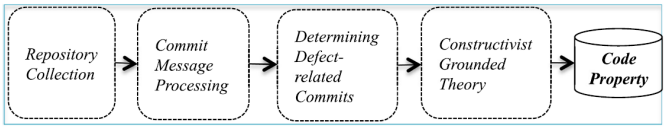

Fig. 1. Methodology for RQ1.

Finally, to answer research question 3, we have implemented a defect prediction model similar to the one used in the original paper using the identified properties obtained at the end of section 3.1.2 and five statistical learners: Classification and Regression Trees, K Nearest Neighbor classification, Logistic Regression, Naive Bayes classification, and Random Forest to predict the defective IaC scripts. Section 3.5.1 involves applying principal component analysis (PCA) while section 3.5.2 involves using statistical learners to construct defect prediction models. Section 3.5.3 involves evaluating the constructed prediction models using 10x10 cross-validation and using four performance measures: AUC, F-Measure, Precision, and Recall.

II. Research questions used in the replication

There are three research questions that the authors try to answer in the original research paper.

- **RQ-1: What source properties characterize defective infrastructure as code scripts?**
- **RQ-2: Do practitioners agree with the identified source code properties?**
- **RQ-3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?**

To answer these questions, we have replicated the same steps as the authors did. The in-depth explanation of each method is explained in the following section.

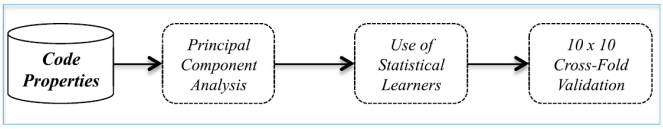

Fig. 2. Methodology for RQ3.

III. Methodologies for answering each research question

A. Methodology for RQ-1

The first research question from the original paper is "*What source code properties characterize defective infrastructure as code scripts?*" To answer it, the authors want to construct four datasets by collecting repositories from 4 organizations: Mirantis, Mozilla, OpenStack, and Wikimedia.

The first step (*section 3.1.1*) is to mine the relevant repositories from GitHub using the GitHub API and applying 3 criteria to construct the datasets:

- **Criteria-1**: The repository must be available for download.
- **Criteria-2**: At least 11% of the files belonging to the repository must be IaC scripts.
- **Criteria-3**: The repository must have at least two commits per month.

Firstly, we retrieve all the repositories via the GitHub API for each organization. It is to be noted that a recursion strategy has been applied to fetch all repositories as the API returns only 100 repositories per request. Secondly, for each repository obtained from the API, the methodology is as follows. Firstly, for the first criteria, we confirm that a repository is downloadable if it is neither a fork nor private. Secondly, for the second criteria, we retrieve all the languages used in the repository by making a GET request to the *languages* endpoint. This returns a response in the form of a dictionary. We then verify if this data structure contains a key named "Puppet" to identify IaC scripts. We then calculate the percentage of IaC scripts by dividing the IaC files by the total number of files returned from the response. If this number is above 11%, the criteria 2 is satisfied. Thirdly, we go through all the commits made in the repository and classify the date of the commit in the form of "Year-Month" in a Hashmap. Each time, a year-month pair is encountered, the value of this key in the dictionary is incremented. After this taxonomy, we iterate over the dictionary to compare the values for each key. If a key has more than 2 commits per month, this means the criteria 3 is respected.

After filtering the initial repositories with the 3 criteria, we obtained 21 repositories for Mirantis, 11 repositories for Wikimedia, and 40 repositories for OpenStack. We did not mine the repositories for Mozilla as it is not hosted on GitHub but on Mozilla's website.

The second step (*section 3.1.2*) is to process the commit messages found in the filtered repositories. The authors chose two artifacts from the VCS of the selected repositories to construct the datasets. The first one is the commits that indicate modification of IaC scripts and the second is the issue reports that are linked with the commits. The authors used commits as they contain information on how and why a file is changed and they can also include links to issue reports. After obtaining the relevant repositories, the authors go through a series of steps shown in Fig. 1, where the steps are as follows:

- First, the authors extract commits that were used to modify at least one IaC script. A commit lists the changes made on one or multiple files.
- Second, the authors extract the message of the commit identified from the previous step. A commit includes a message, commonly referred to as a commit message. The commit messages indicate why the changes were made to the corresponding files.
- Third, if the commit message included a unique identifier that maps the commit to an issue in the issue tracking system, the authors extract the identifier and use that identifier to extract the summary of the issue. The authors use regular expressions to extract the issue identifier. They then use the corresponding issue-tracking API to extract the summary of the issue.
- Fourth, the authors combine the commit message with any existing issue summary to construct the message for analysis. The combined messages are known as "extended commit messages" or XCMs. They are used to separate the defect-related commits from the non-defect-related commits.

The exact steps above have been applied during implementation to replicate the process. Once again, it is worth highlighting that a repository may have a humongous amount of commits. Hence, the GitHub API returns 100 commits at a time. To handle this issue, the pagination method has been applied via recursion to obtain the commits for several consecutive pages. However, due to GitHub's API rate limit of 5000 requests per hour, we were forced to limit the number of possible recursions to 5 HTTP calls for a maximum of 500 commits per repository as the number of HTTP calls required to retrieve all the commits for each repository was getting more and more expensive.

Upon the completion of the steps listed in Fig. 1, the author's methodology of result validation by using the Mann-Whitney U test and Cliff's Delta. The former compares the property count for defective and neutral files while the latter compares the distribution of each characteristic between defective and neutral files.

B. Methodology for RQ-3

The third research question from the original paper is "*How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?*". To answer it, the authors have followed the steps shown in Fig. 2. With the code properties found during the replication of sections 3.1.1 and 3.1.2, the authors apply first a log transformation on the extracted counts for each source code property. The application of log transformation on the values helps in defect prediction. The authors then apply the principal component analysis (PCA) followed by the use of statistical learners to construct defect prediction models. We then evaluate the constructed prediction models using 10x10 cross-validation and performance metrics such as AUC, F-Measure, Precision, and Recall.

The authors use PCA to account for multi-collinearity amongst features and it's used in the domain of defect prediction. PCQ creates an independent linear combination of the features that account for most of the co-variation of the features and it also provides a list of components and the amount of variance explained by each component. The authors use it to determine what properties should be used for building prediction models.

The authors also use five statistical learners to build prediction models that learn from historical data and make prediction decisions on unseen data. Like the authors, we have used the Scikit Learn API to construct our prediction model. The five statistical learners are:

- **Classification and Regression Tree (CART):** It generates a tree based on the impurity measure and uses it to provide decisions based on input features. The authors selected CART because it does not make any assumption on the distribution of features and it's robust to model overfitting.
- **K Nearest Neighbor (KNN):** It stores all available prediction outcomes based on training data and classifies test data based on similarity measures. The authors chose it because prior research has reported that defect prediction models that use KNN perform well.
- **Logistic Regression (LR):** It estimates the probability that a data point belongs to a certain class given the values of features. It provides good performance for classification if the features are roughly linear. The authors select LR because it performs well for classification problems such as defect precision and fault prediction.
- **Naive Bayes (NB):** It computes the posterior probability of each class to make prediction decisions. The authors chose it because prior research has reported that defect prediction models that use NB perform well.
- **Random Forest (RF):** It's an ensemble technique that creates multiple classification trees, each of which is generated by taking random subsets of the training data. The authors chose it because researchers recommended the use of RF which uses ensemble techniques to build defect prediction models.

To evaluate the prediction precision performance of the performance models, the authors used four measures: Precision, Recall, Area Under the Receiver Operating Characteristic Curve (AUC), and F-measure. The precision metric measures the proportion of IaC scripts that are actually defective while the recall measures the proportion of defective IaC scripts that are correctly predicted by the prediction model. The AUC is chosen because it is threshold-independent unlike precision and recall and F-Measure measures the harmonic mean of precision and recall.

To validate the results, the authors use 10x10-fold cross-validation to evaluate the prediction model. This method randomly partitions the dataset into 10 equal-sized subsamples or folds. The performance of the prediction models is tested by using 9 of the 10 folds as training data and the remaining fold as test data. The authors perform 10 times 10-fold cross-validation to avoid prediction errors. The results that the authors reported are the median scores of the 10 runs.

IV. Results and Discussion

A. RQ-1: What source properties characterize defective infrastructure as code scripts?

1) Repository Collection 3.1.1: To begin with, we retrieved all the repositories from the three organizations. We obtained 242, 2011, and 1324 repositories for Mirantis, Wikimedia, and OpenStack respectively. To obtain relevant repositories from the organizations, the authors have applied 3 criteria.

Table I highlights our dataset, i.e., the number of filtered repositories we obtained after applying each criteria one after another whereas Table II lists the dataset of the repository collection from the authors' findings.

TABLE I
Results of our dataset from Repository Collection

Criteria	Mirantis	Wikimedia	Openstack
1	194	1944	1324
2	21	11	40
3	21	11	40

TABLE II
Results of authors' dataset from Repository Collection

Criteria	Mirantis	Wikimedia	Openstack
1	26	1638	1253
2	20	11	61
3	20	11	61

By comparing the values from Tables I and II, we can confirm that both tables have the same tendency. The values we obtain are similar to those of the authors. However, for

the first criteria, the difference between the values could be due to some updates made to the organization's repositories. For instance, some repositories may have been deleted or added. Furthermore, as for the second and third criteria, we can induce that our implementation is correct as the results are almost the same. For Mirantis, we obtain 21 repositories after the filtering whereas the authors obtain 20 repositories. For Wikimedia, we both obtain the same values. As for OpenStack, the difference of 21 repositories could be due to the additional number of repositories we retrieved from this organization. Moreover, it could also be because one or two commits have been made to the repositories recently and this eliminates the minimum two commits per month rule. As a result, the number of repositories we obtain for OpenStack is lower than the ones from the authors.

2) Commit message processing 3.1.2: From the filtered repositories obtained in the previous section, we implemented the four steps required to process and identify defective commits. The two subsequent tables illustrate the dataset from the authors' work as well as ours based on our implementations.

TABLE III
RESULTS OF OUR DATASET FROM COMMIT MESSAGE PROCESSING

Statistic	Mirantis	Wikimedia	Openstack
Puppet Scripts	173	821	2253
Commits with Puppet Scripts	1350	1497	4619

TABLE IV
RESULTS OF AUTHORS' DATASET FROM COMMIT MESSAGE PROCESSING

Statistic	Mirantis	Wikimedia	Openstack
Puppet Scripts	180	296	1383
Commits with Puppet Scripts	1021	972	7808

Once again, by comparing both tables above, we can deduce that our values for Mirantis organization match more or less with those from the authors' side. We obtain 173 Puppet scripts from the selected repositories whereas they obtain 180 repositories. The number of commits associated with the Puppet Scripts can be marked as similar as well. However, there is a huge difference between the values of Wikimedia and OpenStack organizations. This could be because the number of repositories have been modified and more commits have made to these projects recently. As a result, more Puppet scripts have been inserted in the repositories.

3) Mann-Whitney U test and Cliff's Delta: As for this part, we first downloaded the authors' reported data in order to perform the required analysis. We then read the downloaded CSV files and apply the Mann-Whitney U test as well as Cliff's Delta on the dataset. We save the output in CSV files for each organization. The table V and VI below show our results for all four organizations.

TABLE V
OUR RESULTS OF MANN-WHITNEY U TEST AND CLIFF'S DELTA - PART I

Property	Mirantis		Mozilla	
	p-value	Cliff	p-value	Cliff
Attribute	< 0.001	0.41	< 0.001	0.41
Command	< 0.001	0.24	< 0.001	0.18
Comment	< 0.001	0.37	0.47	0.03
Ensure	< 0.001	0.38	0.05	0.09
File	< 0.001	0.36	< 0.001	0.18
File_mode	< 0.001	0.41	< 0.001	0.24
Hard_coded_string	< 0.001	0.55	< 0.001	0.41
Included	< 0.001	0.33	< 0.001	0.31
Lines_of_code	< 0.001	0.5	< 0.001	0.51
Require	< 0.001	0.36	< 0.001	0.2
SSH_KEY	< 0.001	0.39	< 0.001	0.24
URL	< 0.001	0.22	0.02	0.08

TABLE VI
OUR RESULTS OF MANN-WHITNEY U TEST AND CLIFF'S DELTA - PART II

Property	Openstack		Wikimedia	
	p-value	Cliff	p-value	Cliff
Attribute	< 0.001	0.35	< 0.001	0.47
Command	< 0.001	0.07	< 0.001	0.18
Comment	0.87	0	< 0.001	0.22
Ensure	< 0.001	0.19	< 0.001	0.29
File	< 0.001	0.09	< 0.001	0.31
File_mode	< 0.001	0.07	< 0.001	0.24
Hard_coded_string	< 0.001	0.37	< 0.001	0.55
Included	< 0.001	0.22	< 0.001	0.37
Lines_of_code	< 0.001	0.33	< 0.001	0.51
Require	< 0.001	0.11	< 0.001	0.32
SSH_KEY	< 0.001	0.07	< 0.001	0.24
URL	0.97	0	< 0.001	0.17

Instantly, we observe that the authors' results match almost perfectly with ours. On one hand, this means our replication implementation is correct. On the other hand, we arrive at the same conclusions as the authors. Firstly, the "Lines of codes" is high for three of the organizations. Secondly, the Cliff's value of "Hard Coded string" is high for Mirantis and Wikimedia. Thirdly, the properties with the highest Cliff's value for Mirantis, Mozilla, Openstack and Wikimedia are "Hard Coded string", "Lines of codes", "Hard Coded string" and "Lines of codes" respectively. This analysis shows us the most two common properties having the strongest correlation with defective IaC scripts. Basically, since our tables' values are similar to the authors', it is almost certain that we both have the same results and conclusions after performing the Mann-Whitney U test and Cliff's Delta analysis.

B. RQ-3: How can we construct defect prediction models for infrastructure as code scripts using the identified source code properties?

1) Principal component analysis 3.5.1: With the set of source code properties found using constructivist grounded, we have replicated the author's method by applying Principal component analysis (PCA). It computes the total amount of variance to determine what properties should be used for

Dataset	Property-based	Bag-of-words
Mirantis	1	50
Mozilla	1	140
Openstack	2	400
Wikimedia	2	150

Fig. 3. Table for the number of principal components from original paper

TABLE VII
NUMBER OF PRINCIPAL COMPONENTS FOUND DURING REPLICATION

Dataset	Number of principal components
Mirantis	6
Mozilla	8
Openstack	7
Wikimedia	7

building prediction models. We only select principal components that account for at least 95% of the total variance to avoid overfitting. Fig. 3 shows the number of principal components that the original paper found while Table VII shows the number of components the original paper found. We can see a large difference between the number of components found originally and during replication. There might be a few factors explaining it. The first factor is the *Analysis Methodology*. Even with the same data, different methods or parameters used for the PCA could yield different numbers of principal components. We should analyze further the algorithm settings, the initialization and the stopping criteria of the original code. Another factor can be the *Reported vs Used Components*. The original paper might not have reported all the components that were actually significant according to their criteria. Furthermore, some implementations of PCA involve random initialization, which could lead to different outcomes.

TABLE VIII
PRECISION METRIC FOR EACH PREDICTION MODEL DURING REPLICATION

	CART	KNN	LR	NB	RF
Mirantis	0.67	0.72	0.73	0.75	0.71
Mozilla	0.68	0.62	0.7	0.68	0.74
Openstack	0.7	0.74	0.66	0.74	0.73
Wikimedia	0.66	0.71	0.72	0.75	0.67

TABLE IX
RECALL METRIC FOR EACH PREDICTION MODEL DURING REPLICATION

	CART	KNN	LR	NB	RF
Mirantis	0.63	0.63	0.71	0.71	0.71
Mozilla	0.7	0.52	0.6	0.56	0.71
Openstack	0.71	0.72	0.77	0.65	0.77
Wikimedia	0.67	0.63	0.73	0.65	0.69

2) Statistical learners 3.5.2 and Evaluation methods 3.5.3: The authors used five statistical learners to build prediction models to predict IaC scripts and we have used the same learners to replicate. We have reported the results for each evaluation metric on four tables, from Tables VIII to XI. After comparison with the author's original data, we have found that we got similar results as the authors for all the results. The results are not identical because of several reasons. The first reason might be due to some computational limitations. It can introduce tiny variances in calculations, especially when dealing with floating-point numbers. Another possible reason is that algorithms like Random Forests or K-Nearest Neighbors might have elements or randomness in initialization, leading to small variations each time the model is run. A final reason why the results might have slight variations is that K-Fold validation splits the data into 10 and splitting the data differently might yield different results.

TABLE X
AUC METRIC FOR EACH PREDICTION MODEL DURING REPLICATION

	CART	KNN	LR	NB	RF
Mirantis	0.63	0.72	0.78	0.76	0.75
Mozilla	0.73	0.71	0.75	0.74	0.83
Openstack	0.64	0.74	0.68	0.73	0.76
Wikimedia	0.61	0.72	0.77	0.77	0.72

TABLE XI
F-MEASURE METRIC FOR EACH PREDICTION MODEL DURING REPLICATION

	CART	KNN	LR	NB	RF
Mirantis	0.63	0.67	0.7	0.71	0.7
Mozilla	0.69	0.56	0.64	0.61	0.73
OpenStack	0.71	0.73	0.7	0.69	0.75
Wikimedia	0.66	0.65	0.72	0.69	0.67

V. CONCLUSION

In conclusion, the goal of the original paper is to help practitioners increase the quality of IaC scripts by providing empirical data that identifies code properties from defective IaC scripts. Our replication project is to validate their results by replicating their steps.

For the first part, We mined repositories from Mirantis, OpenStack, and Wikimedia and then applied filtering criteria to obtain a list of relevant repositories containing defective IaC scripts. Then, we followed the author's commit message processing where we identified commit messages mentioning at least one IaC script that might have issue reports. For the second part, with the data provided by the authors, we replicated their methodology to build a prediction model able to predict defective IaC scripts. The methodology involves applying principal component analysis (PCA) to determine what properties should be used for building prediction models. We then apply statistical learners to build prediction models that learn from historical data and make prediction decisions on unseen data. Finally, we validate our prediction models by using 10x10-fold cross-validation.

REFERENCES

[1] Rahman, A., Williams, L., 2019. Source code properties of defective infrastructure as code scripts. Inf. Softw. Technol. 112, 148–163.

[2] Our GitHub repository for the Replication project. https://github.com/keshavDooleea/LOG6307E_Replication_Project