Continuous defect prediction in CI/CD pipelines: a machine learning-based framework *

Lazzarinetti Giorgio[1 [0000−0003−0326−8742]], Massarenti Nicola[1 [0000−0002−8882−4252]], Sgrò Fabio[1 [0000−0002−7684−3595]], and Salafia Andrea[1 [0000−0002−6539−5547]]

Noovle S.p.A, Milan, Italy https://www.noovle.com/en/

Abstract. Recent advances in information technology has led to an increasing number of applications to be developed and maintained daily by product teams. Ensuring that a software application works as expected and that it is absent of bugs requires a lot of time and resources. Thanks to the recent adoption of DevOps methodologies, it is often the case where code commits and application builds are centralized and standardized. Thanks to this new approach, it is now possible to retrieve log and build data to ease the development and management operations of product teams. However, even if such approaches include code control to detect unit or integration errors, they do not check for the presence of logical bugs that can raise after code builds. For such reasons in this work we propose a framework for continuous defect prediction based on machine learning algorithms trained on a publicly available dataset. The framework is composed of a machine learning model for detecting the presence of logical bugs in code on the basis of the available data generated by DevOps tools and a dashboard to monitor the software projects status. We also describe the serverless architecture we designed for hosting the aforementioned framework.

Keywords: Continuous Defect Prediction · Machine Learning · DevOps · Continuous Integration

1 Overview

In the context of the italian Fondo per la Crescita Sostenibile, Bando "Agenda Digitale", D.M. Oct. 15th, 2014, funded by "Ministero dello Sviluppo Economico", a lot of teams manage and contribute to different software projects daily. Given the high number of activities that must be taken into account, such

* Activities were partially funded by Italian "Ministero dello Sviluppo Economico", Fondo per la Crescita Sostenibile, Bando "Agenda Digitale", D.M. Oct. 15th, 2014 - Project n. F/020012/02/X27 - "Smart District 4.0".

as managing branches (development, stage, production, features), architecting software applications, coordinating the developers and interacting with project managers, it's useful to have some automatic tools that alert the developers in case bugs are detected in code. For such a reason, we propose a framework that aims at easing the management and development activities and that integrates with DevOps methodologies, with a focus on Continuous Integration (CI) and Continuous Delivery (CD) operations. With CI/CD operations, thanks to analysis tools such as Jenkins [36], it is possible to detect and avoid unit or integration errors before shipping applications to production environments. However, such tools are not able to detect logical bugs and therefore to block builds triggered from commits. For this reason we decided to develop a methodology based on machine learning techniques to detect if a commit could contain a logical bug. The final goal is that of using the proposed methodology to build a monitoring framework integrated with CI/CD operations that allows a visual exploration of the status of each software project, in order to evaluate the quality of the software produced and, in case the machine learning model detects issues, automatically raise alerts to fix the bug before it reaches production environments.

The rest of this paper is organized as follows: Chapter 2 describes the state of the art for continuous defect prediction, whereas Chapter 3 describes the dataset, the preprocessing operations, the models used and the developed dashboard. In Chapter 4 some infrastructural considerations are described and, finally, Chapter 5 draws some conclusions and some future works.

2 State of the art

DevOps is a software development methodology used in computer science that aims at enhancing communication, collaboration and integration between developers and information technology operations [1]. DevOps wants to respond to the interdependence between software development and IT operations, aiming to help an organization to develop software products and services more quickly and efficiently [2]. DevOps automated analysis systems generate huge amounts of data that can be used to detect unnecessary processes, monitor production and predict bugs. Server logs can reach hundreds of megabytes in a short time while additional monitoring tools, like Jenkins [36] or SonarQube [37] can generate gigabytes of data. Quantities force developers to set up automatic checks with the use of thresholds for identifying problems. However, the thresholds are not optimal in this context, given the scarce generalization of the parameters and the zero adaptation to the infrastructure over time [3]. Moreover, generally, the systems used in projects that adopt DevOps are many and of different nature. Each system monitors the health and performance of applications in different ways. It is therefore difficult to find relationships between different data sources. Thus, a better approach to analyze this data in real time is through the application of machine learning techniques, which allow to give a new vision of the metrics collected with the DevOps tools. Machine learning techniques applied in this context allow to monitor the progress of deliveries and the presence of

bugs using data collected by continuous integration systems. Machine learning systems can also use input data of a different nature to produce a more robust view of the applications on which they are used [4].

When it comes to software bugs, they usually appear during the software development process and are difficult to detect or identify, thus developers spend a large amount of time locating and fixing them. In order to detect them, many machine learning algorithms have been developed and tested [12]. Indeed, machine learning algorithms can be applied to analyze data from different perspectives and can benefit from the large amount of code production metrics that are also used by developers to obtain useful information. Many examples of machine learning solutions for detecting software bugs have been implemented. For example in [14] a combination approach of contexts and Artificial Neural Network (ANN) is proposed. In [15] three algorithms are compared, namely Naive Bayes (NB), Decision Tree (DT) and ANN, showing that DT has the best results over the others. In [16] Bayesian Network (BN) and Random Forest (RF) are compared, showing that BN can outperform RF. Differently, in [17] NB, RF and ANN are compared, showing that RF is better than the others. In [18] also deep learning techniques are proposed, showing good performance.

From all these studies emerge that, apart from the choice of the algorithm that varies according to the used dataset, software metrics are extremely important for fault prediction in quality assurance, hence, identification of proper metrics is essential in all software projects [9]. D'Ambros et al. [5] proposed a benchmark to compare prediction techniques on five publicly available datasets focusing on the different metrics related to code production, such as line of code, code complexity [6], number of changes [7] or previous fault [8].

In the context of DevOps CI builds, software bug detection plays an extremely important role, particularly at change-level. Change-level defect prediction, also known as *just-in-time* defect prediction, aims at predicting defective changes (i.e. commits to a version control system) and is more practical because it can not only ensure software quality in the development process, but also make the developers check and fix the defects just at the time they are introduced. There are a lot of studies about this. For example, in [13] the authors propose a deep learning based approach over six different datasets, showing good results in this kind of task. Indeed, their framework relies on a preprocessing and feature engineering step and on the definition of the deep neural network classifier. The chosen model differs from the proposal of [10], that relies on Logistic Regression (LR), because LR considers the contribution of each feature independently and performs well only when input features and output labels are linearly correlated. For such reasons, in [13] the authors propose a Deep Belief Network (DBN) which has the advantage of generating new non-linear combinations features given the initial set of features.

Scientific community also proposed several datasets related to continuous defect prediction. In [11] the authors make available 11 million data rows retrieved from CI builds that embrace 1265 software projects, 30022 distinct commit authors and several software process metrics. Another well known dataset

for continuous defect prediction is the Technical Debt Dataset [19], a curated set of project metrics data from 33 Java projects from the Apache Software Foundation. It has been produced using four tools, i.e. PyDriller [25], Ptidej [38], Refactoring Miner [26] and SonarQube. The Technical Debt Dataset includes information at commit granularity, such as the commit hash, the date, the message, on the refactoring list applied in each commit, on the code quality, such as the list of detected issues related to a commit, the style violations, the detected anti-patterns and the code smells. Other included information are the Jira [28] issues retrieved from the project's issue tracker as well as the fault-inducing and the fault-fixing commits, that are the association for each fixed fault of the commit where the fault was created and where the fault was fixed.

3 The Framework

The objective of this research is to develop a framework capable of identifying bugs from committed code in order to provide from the one hand a synoptic point of view of the status of software projects and from the other hand alerting if some inconsistencies and logical bugs are detected.

The goal is that of using such framework trained on a publicly available dataset in a real case scenario.

The proposed framework consists of three main components: a data processing pipeline, a machine learning model for classification and a monitoring dashboard. In the following sections we will described all these components in details, by focusing on the publicly available dataset used for training the model, the preprocessing operations executed to make the dataset compliant with data from the production environment (since data collected from the real case infrastructure has a different granularity with respect to those coming from the publicly available dataset because of system constraints), the machine learning models tested that follows the trend of the state of the art and the implemented dashboard with the way of using and the kinds of analysis performed over it.

3.1 The Dataset

The dataset used for model training is the Technical Debt Dataset [19]. It contains information at the granularity level of the commits organized in nine different tables:

- **Projects**: contains the links to the GitHub repository and the associated Jira issue tracker.
- **Sonar measures**: contains the SonarQube measures such as number of code lines in the commit, the code complexity and the number of functions.
- **Commits**: contains the information retrieved from the git log including the commit hash, the message, the author, the date and timezone and the list of branches.

- **Commit changes**: contains the changes contained in each commit, including the old path of the file, the new path, the type of change (added, deleted, modified or renamed), the diff, the number of lines added.
- **Jira issues**: contains Jira issues for each project with information such that the key, the creation and resolution dates and the priority.
- **Fault inducing commits**: reports the results from the execution of the SZZ [21] algorithm.
- **Refactoring miner**: contains the list of refactoring activities applied in the repository. The table contains the project, commit hash, the type of refactoring applied and the associated details.
- **Sonar issues**: contains the list of SonarQube issues such that the anti-patterns and the code smess.
- **Sonar rules**: contains the list of rules monitored by SonarQube.

3.2 Data Preparation

As mentioned, since the real case production environment slightly differs from the aforementioned dataset due to some constraints imposed by the adopted CI/CD tools, that imposes us to have data aggregated at push granularity instead of commit granularity, the Technical Debt Dataset has first been processed and synthetically modified to match the push granularity by aggregating subsequent commits with windows of varying lengths. Aggregation of numerical features has been executed in some cases by averaging the numerical values while in other cases selecting only the min/max values, depending on the meaning of the feature in the context of software development.

More precisely, in order to create the proper dataset to train machine learning models, we consider only some of the available tables, namely GIT_COMMITS, GIT_COMMITS_CHANGES, SONAR_ISSUES, SONAR_MEASURES and SZZ_FAULT_INDUCING_COMMITS. The choice of using only some tables and, consequently, sonly some features among those available has been done to match the features actually collected in the real case infrastructure. Tables are related as in the Entity Relation schema depicted in Figure 1. According to such schema, data are prepared as follow.

Firstly, we join GIT_COMMITS with SONAR_MEASURES on *projectID* and *commitHash* since they have the same granularity. Then, given that for each commit hash in the GIT_COMMITS table there are more commit changes in the GIT_COMMITS_CHANGES table (one for each file changed), we aggregate commit changes at commit hash level and compute sum and mean for commit changes features *linesAdded, linesRemoved, nloc, complexity* and *tokenCount* and count for *changeType* (Rename, Delete, Modify, Unknown, Add). Thus, we merge GIT_COMMITS_CHANGES with the previously prepared dataset aggregated at commit hash level and, then, merge the resulting dataset with SONAR_ISSUES by adding a label defining if the issues were created or closed within each commit hash. Here, we consider only sonar issues associated with commits that induced them and we aggregate only some relevant features, namely

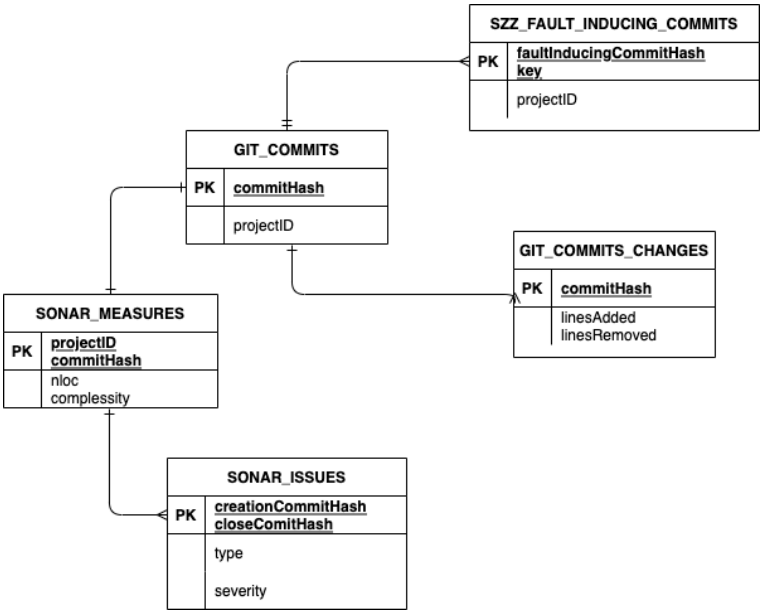

Fig. 1. List of the features with their correlation with dataset label.

severity, startLine and *effort*. Finally, we merge the resulting dataset with SONAR_ISSUES at *projectID*.

Over these feature engineered dataset, we then add the label for the final goal of detecting fault inducing commits. To this attempt, we use the SZZ_FAULT_INDUCING_COMMITS table, from which we extract all and only those commits that induced a bug. The label is assigned by the author of the dataset following their own implementation of the SZZ algorithm [19] (The OpenSZZ [21]). The SZZ [22] algorithm tries to identify the fault-inducing commits from a project's version history. The algorithm was developed in 2005 and has since been adopted in more than 200 empirical studies[23, 24]. The algorithm is based on Git's blame/annotate feature and assumes that the fault-inducing commit of a fault is known. Usually this is done by combining data from an issue tracker and from Git's log command.

3.3 Feature Engineering and Selection

Once the dataset has been reduced to the required granularity and some preliminary features have been computed as described, data are prepared for machine learning models. First of all, in order to remove missing values, numerical and categorical variable are imputed using respectively the median and the mode values. After imputation, categorical variable are converted into numerical with an ordinal encoder. Variable with zero variance are removed and finally the dataset is balanced with respect to the fault inducing class. Indeed, after data prepara-

tion, there were 517 commits associated with the fault inducing class and 78341 commits non associated with the fault inducing class. Thus, data are subsampled to match the dimensionality of the positive class.

In addition, dataset's features have undergone a selection process that involves studying the correlation between each feature with respect to the dataset label, as shown in Figure 2. The process consisted of a Recursive Feature Elimination (RFE) [20] technique that allows to select the best number of features using as cutoff the F1-score drop. In particular, we train several RF classifiers increasing number of feature by following the order of feature importance as computed in Figure 2. For each classifier trained, we measure the F1-score and then we plot the values of the F1-score registered with the varying number of features. In Figure 3 we can see the result of such RFE process. It is possible to see as the best performance are reached with 18 features, with an average F1-score of 0.786. This allows us to reduce the dimensionality of data by only keeping those features valuable to the model.

Feature name	**Description**	**Aggregation**	**Source**
tokenCount	Token count of functions	sum, mean	PyDriller
complexity	Cyclomatic complexity	sum	PyDriller
nloc	Lines of code of the file	sum, mean	PyDriller
linesAdded	Number of lines added	sum, mean	PyDriller
linesRemoved	Number of lines removed	sum, mean	PyDriller
modificationType	Type of changed applied (modify, delete, add, rename)	count	PyDriller
filesChanged	Number of modified files	count	PyDriller
effort	Time needed to solve the issues	sum	SonarQube
classComplexity	Complexity of classes in commits	count	SonarQube
severity	Severity level of the issues	max	SonarQube

Table 1. Detail of selected features

From hence, the final set of features selected for training is described in Table 1. The RFE process allowed us to select only 18 features. in Table 1 we can see the feature name, the description, the aggregations and the source. Each feature is considered once per each aggregation, thus, as an example, the *tokenCount* feature is considered twice, both as sum of token count of each commit and as mean of token count of each commit. Moreover, the feature *modificationType*, given its categorical nature, is considered as a unique feature for each category it can assume (modify, delete, unknown, add, rename).

It is interesting to notice as, among all the information available coming from SonarQube and Git, the most important features are related to commit size (e.g. number of lines added/removed, lines of code in files, number of token in function, type of change applied) and to the type and the effort to restore the sonar issue. This seems to be somehow intuitively explainable if we consider

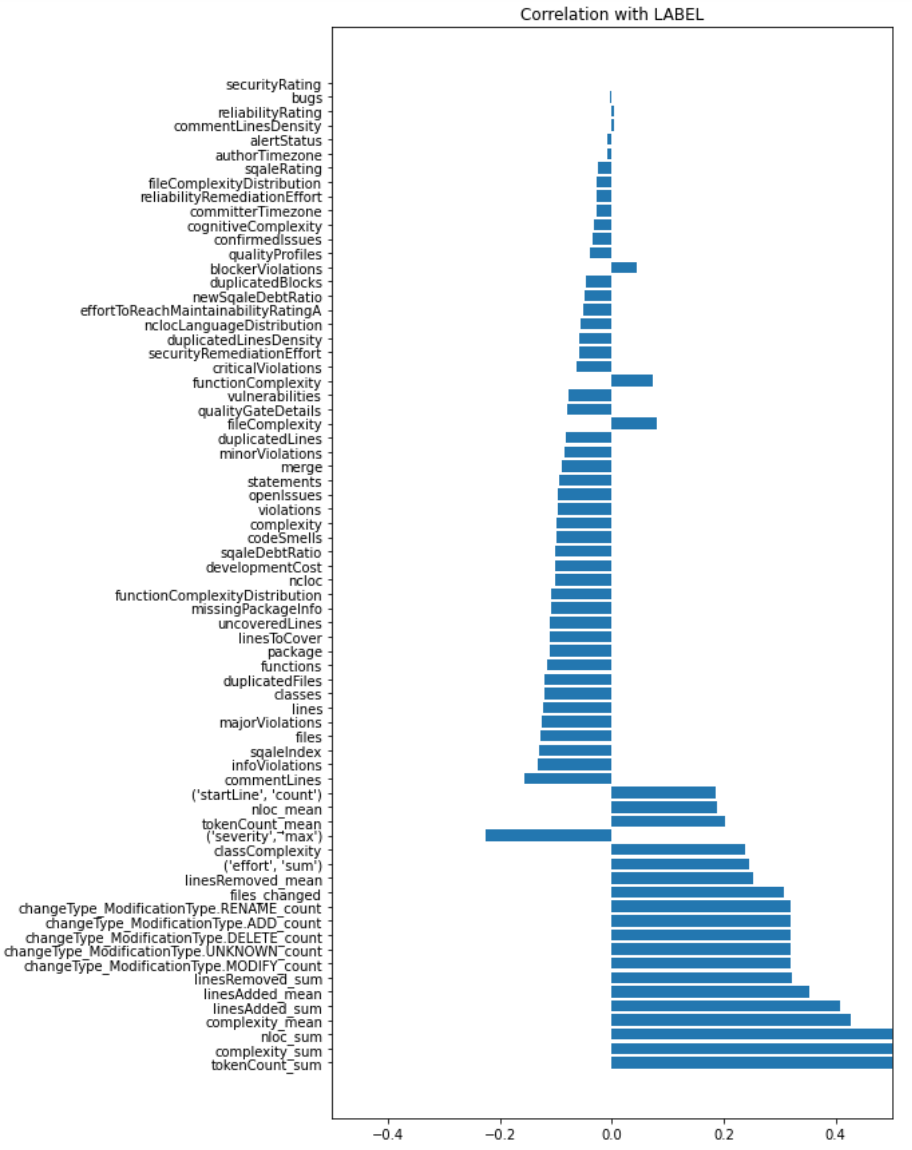

Fig. 2. List of the features with their correlation with dataset label.

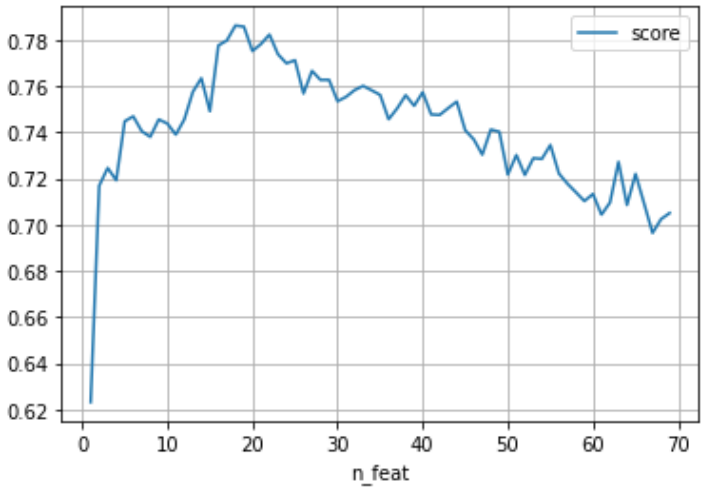

Fig. 3. F1-score values per number of features used during training.

that it is more probable to insert logical bug in code when a lot of changes are performed with respect to when very small changes are applied.

Finally, once features as been prepared and selected, as mentioned, since in the real case scenario data are aggregated at push level, instead of commit level, we aggregate subsequent commits to replicate the push granularity. To aggregate the features we create a set of windows with lengths varying from 1 to 4 in a random way. We choose to create the windows sizes randomly since, by analyzing the real case scenario data, we discovered that data does not follow a particular distribution. In Figure 4 we can see the distribution of the windows sizes used to aggregate commits. After aggregation the number of element in the positive class were 257, against 299 elements in the negative class.

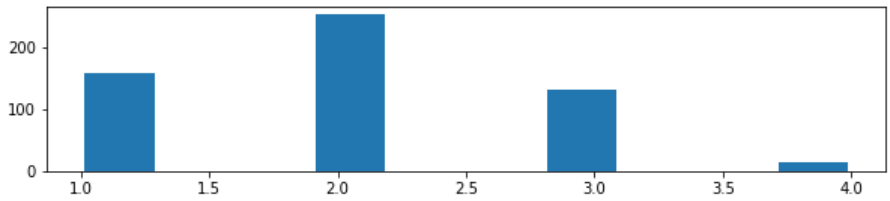

Fig. 4. Windows size distribution

3.4 The Models

The models chosen for continuous defect prediction are the ones mostly used in scientific research, such as in [14, 17]. In particular, we tested RF and ANN. To test the performance of the two models we split the dataset into train and test sets, with the 80/20 ratio rule.

As far as RF is concerned, its hyperparameters have been identified by means of grid search optimization over a 5-fold cross validation. The grid search has been performed considering the combination of the following values:

- **Criterion**: entropy, gini
- **Bootstrap**: True, False
- **Number of Estimators**: 100, 250
- **Maximum tree depth**: 10, None
- **Minimum number of samples per splitting**: 2, 3

After the grid search optimization the following parameters has been selected:

- **Criterion**: gini
- **Bootstrap**: yes
- **Number of Estimators**: 250
- **Maximum tree depth**: None
- **Minimum number of samples for splitting**: 3

The overall performance on the test set are shown in Table 2.

Class	Precision	Recall	F1-score	support
0	0.92	0.91	0.91	76
1	0.89	0.90	0.90	63

Table 2. Performance of Random Forest

Where the *True Negatives (TN)* are 57 and the *True Positives (TP)* are 69 whereas the *False Positives (FP)* are 7 and the *False Negatives (FN)* are 6.

The other model tested is a Dense Neural Network (DNN), a particular case of the ANN, trained for binary classification. Its hyperparameters have been identified by means of grid search optimization over a 5-fold cross validation that check a combination of the following parameters:

- **Optimizer**: AdaDelta, Adam
- **Learning rate**: 0.01, 0.1
- **Maximum Number of epochs**: 128
- **BatchSize**: 8, 16, 32

Early stopping criterion has been applied to avoid overfitting and select the number of epochs. The optimization resulted in the following selection:

- **Optimizer**: Adam

- **Learning rate**: 0.01
- **Number of epochs**: 5
- **BatchSize**: 8

The network architecture is composed of a dense layer with *relu* activation function and an output dimension equal to 64 and 1344 parameters, a dropout layer with a 10% of dropout and a final dense layer with a *softmax* activation function and an output dimension equal to 2 and 130 parameters. The network has been trained with categorical cross entropy as loss function and accuracy as metric. Performance of the DNN model on the test set is shown in Table 3.

Class	Precision	Recall	F1-score	support
0	0.90	0.71	0.79	76
1	0.72	0.91	0.81	63

Table 3. Performance of Dense Neural Network

Where the *True Negatives (TN)* are 53 and the *True Positives (TP)* are 58 whereas the *False Positives (FP)* are 22 and the *False Negatives (FN)* are 6.

As shown above, performance of the RF is better than the one of the DNN, especially when comparing the precision and the recall metrics of the positive class. Indeed, the RF model outperforms the DNN when considering all the metrics.

3.5 The Monitoring Dashboard

According to the results of the trained classifiers, the RF model has been deployed and used in a production environment. In order to visualize the predictions of the model and keep track of all the operations performed in the different software projects, each project has been connected to the model and results have been recorded in a database to be visualized. Thus, a monitoring dashboard has been designed to easily read the results of the analysis. Given the high number of data collected daily, we include in the dashboard distributions and time series charts, in order to give a synoptic point of view of the software projects' status. In Figure 5 there is an example of the dashboard developed.

In details, the dashboard firstly includes some global metrics that resume the operations performed in the different code repositories over which the extraction and prediction algorithms have been connected. In particular the global metrics included are:

- **Number of commits**: total number of commits analyzed by the system.
- **Lines added** : number of code lines added in the analyzed commits.
- **Lines removed**: number of code lines removed in the analyzed commits.
- **Files changed**: number of files modified in the analyzed commits.
- **Total bugs**: number of bugs identified by the machine learning algorithm in the analyzed commits.

Fig. 5. Example of dashboard's plots and charts

The dashboard also includes a detailed tables that contains some references useful to reconstruct the history of each commit, such as the username, the branch name, the Jenkins job name, the commit hash, the critical violations and the bugs detected by SonarQube. All these informations are correlated with the model prediction, so that it is easy when the model detect a bug to identify the project, the branch and the authors of the interested commit.

Then, with a pie chart it is possible to easily understand the percentage of commits with a possible bug and with a time series chart it is possible to see the trend of detected bug together with some other important features, such as number of lines added, number of files changed and number of lines removed.

Finally, the dashboard gives the user the possibility to filter the content in order to reduce the number of visualized data and change the dimensions of the analysis. The available filters are based on date range, name of the repository, branch name, Jenkins job name, username and model prediction. In this way, all the charts and metrics previously described can be recomputed with specific filters, thus deeply exploring each project according to different dimensions.

The developed dashboard has not given only the possibility to detect bugs in real time, but also the chance to evaluate the quality of the software projects. Indeed, thanks to the different charts and filters, it has been possible to identify projects or branches with more bugs, which generally means that are more complex projects/branches or projects/branches for which the requirements are not well defined. Moreover, it has been possible to monitor also the resources that contribute to software projects. As an example, it has been possible to iden-

tify resources that usually do not commit RFEquently, but only in case of large changes or improvements in team's code quality.

4 Infrastructural considerations

To conclude, we present some considerations related to the infrastructure that we set-up to serve the proposed framework.

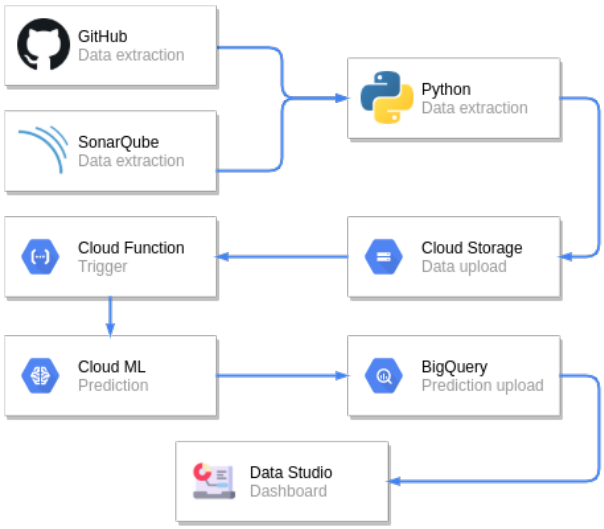

Fig. 6. Architecture of the proposed framework.

We deployed the services using Google Cloud Platform [29] infrastructure. As depicted in Figure 6, the proposed framework requires to retrieve data both from GitHub and SonarQube. GitHub and SonarQube information are analyzed by means of a Python [30] framework called PyDriller that aims at mining software repositories by easily extracting information from any Git repository, such as commits, developers, modification, diffs and source codes, and quickly export CSV file. On a scheduled basis, we retrieved data from Git repositories by means of such framework and we uploaded them on Google Cloud Storage [31], which is a blob-based storage service. This action triggers a Google Cloud Function [32], which is a scalable and serverless functions as a service that, in turn, invokes Google Vertex AI Prediction [33] service for model predictions. Google Vertex AI is Google's unified platform for building, deploy and scale machine learning models over which we deployed the RF models trained and fine tuned. The results are then saved to Google BigQuery [34], a serverless, highly scalable multicloud data warehouse, and made available for visualization by means of a Data Studio [35] dashboard.

5 Conclusions

The goal of this research was that of defining a framework to detect bugs in software projects' commits in a change-level defect prediction scenario. In order to define such framework, we firstly analyzed the state of the art for machine learning algorithms applied to support CI/CD operations, with a focus on continuous defect prediction. The analysis of the state of the art allowed us to define the main machine learning approach to defect prediction, the main publicly available datasets and some related works. Thus, we selected an approach and a public dataset that fit our needs to create the aforementioned framework. However, differently from the related works that focus on processing the logs of CI tools, given the constraints set by CI/CD tools used in our production environments, we needed to preprocess the dataset to consider an agglomerations of record based on subsequent commits. Thus, we preprocessed the dataset in order to match real case scenario granularity. Once data has been prepared, they have undergone a feature engineering step. From the state of the art, indeed, emerged that properly collecting and selecting features is extremely important within this context. Thus, we used a RFE process to give importance to features and select them. Then, we designed and tested two models: a RF and a DNN both with grid search over 5-fold cross validation for hyperparameters optimization and early stopping for DNN. Experimental results showed that, on the preprocessed dataset, RF outperformed DNN, with an average F1-score over the positive class of 0.91 against 0.81. Thus, we define a Google Cloud based architecture to host our framework for real time monitoring, that allowed to link different software projects to the RF model and register all the logs produced in order to visualize the results. We also design a monitoring dashboard, that allowed us to derive important insights and evaluate software quality.

The developed framework is extremely useful, especially thanks to the synoptic point of view that provides, however some enhancements can be performed. As an example, some future developments could involve the augmentation of the features with user-specific information to make the model learning user-patterns. Other future developments are related to a posterior analysis of the machine learning models. Indeed, we couldn't train a model on a real dataset and a manual analysis of the results showed that this is the reason why the model produces some false positive. Thus, validating the results of the models and retrain the model with a real dataset could enhance the model's performance and allow for a better usage of the framework.

References

1. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. IEEE Software, **33**(3), 94–100 (2016)
2. Virmani, M.: Understanding DevOps & bridging the gap from continuous integration to continuous delivery. In: Fifth international conference on the innovative computing technology (INTECH 2015), pp.78–82. IEEE, Galcia (2015)

3. Madeyski, L., Kawalerowicz, M.: Continuous defect prediction: the idea and a related dataset. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 515-518. IEEE, Buenos Aires (2017)
4. Nogueira, A. F., Ribeiro, J. C., Zenha-Rela, M. A., Craske, A.: Improving la redoute's ci/cd pipeline and devops processes by applying machine learning techniques. In: 2018 11th international conference on the quality of information and communications technology (QUATIC), pp. 282–286. IEEE, Coimbra (2018)
5. D'Ambros, M., Lanza, M., Robbe, R.: An Extensive Comparison of Bug Prediction Approaches. In: Proceedings of 7 th IEEE Working Conference on Mining Software Repositories, pp. 31–41. IEEE, Cape Town (2010)
6. Gyimothy, T., Ferenc, R., Siket, I.: Empirical Validation Of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Transactions on Software Engineering **31**(10), 897–910 (2005)
7. Hassan, A.: Predicting Faults Using the Complexity of Code Changes. In: Proceedings of the 31 st International Conference on Software Engineering, pp. 78–88. IEEE, Vancouver (2009)
8. Hassan, A., Holt, R.: The Top Ten List: Dynamic Fault Prediction. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 263–272. IEEE, Budapest (2005)
9. Madeyski, L., Jureczko M.: Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study. Software Quality Journal **23**(3), 393–422 (2015)
10. Kamei, Y., Shihab,E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. TSE **39**(6), 757–773 (2013)
11. Madeyski, L., Kawalerowicz, M.: Continuous defect prediction: the idea and a related dataset. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 515–518. IEEE, Buenos Aires (2017)
12. Alnor, N., Khleel, A., Nehéz, K.: Comprehensive Study on Machine Learning Techniques for Software Bug Prediction. International Journal of Advanced Computer Science and Applications **12**(8), (2021)
13. Yang, X., Lo, D., Xia, X., Zhang, Y., Sun, J.: Deep learning for just-in-time defect prediction. In: 2015 IEEE International Conference on Software Quality, Reliability and Security, pp. 17–26. IEEE, Vancouver (2015)
14. Li, Y., Wang, S., Nguyen, T. N., Nguyen, S. V.: Improving bug detection via context-based code representation learning and attentionbased neural networks. In: Proceedings of the ACM on Programming Languages, pp. 1–30. Association for Computing Machinery, New York (2019)
15. Hammouri, A., Hammad, M., Alnabhan, M., Alsarayrah, F.: Software bug prediction using machine learning approach. International Journal of Advanced Computer Science and Applications **9**(2), 78–83 (2018)
16. Pandey, S. K., Mishra, R. B., Triphathi, A. K.: Software bug prediction prototype using Bayesian network classifier: A comprehensive model. Procedia Computer Science **132**, 1412–1421 (2018)
17. Uqaili, I. U. N., Ahsan, S. N.: Machine learning based prediction of complex bugs in source code. The International Arab Journal of Information Technology **17**(1), 26–37 (2020)
18. Islam, M. J., Pan, P., Nguyen, G., Rajan, H.: A comprehensive study on deep learning bug characteristics. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations

of Software Engineering, pp. 1–11. Association for Computing Machinery, Tallin (2019)
19. Lenarduzzi, V., Saarimaki, N., Taibi, D.: The Technical Debt Dataset. In: Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 1–2. Association for Computing Machinery, New York (2019)
20. Guyon, I., Weston, J., Barnhill, S.,Vapnik, V.: Gene Selection for Cancer Classification using Support Vector Machines. Machine Learning **46**, 389–422 (2002)
21. Pellegrini, L., Lenarduzzi, V., Taibi, D.: OpenSZZ: A Free, Open-Source, Web-Accessible Implementation of the SZZ Algorithm. In: Proceedings of the 28th international conference on program comprehension, pp. 446–450. Association for Computing Machinery, New York (2020)
22. Zeller, A., Sliwerski, J., Zimmermann, T.: When Do Changes Induce Fixes?. In: proceedings of the 2005 International Workshop on Mining Software Repositories, pp. 1–5. Association for Computing Machinery, St. lOUSI, mISSURI (2005)
23. Robles, G., Rodriguez-Perez, G., Gonzalez-Barahona, J.M.: Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. Information and Software Technology **99**, 164–176 (2018).
24. da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A. E.: A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. IEEE Transactions on Software Engineering **43**(7), 641–657 (2017).
25. Spadini, D., Aniche, M., Bacchelli, A.: PyDriller: Python Framework for Mining Software Repositories. In: Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 908–911. Association for Computing Machinery, New York (2018)
26. Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., Dig, D.: Accurate and Efficient Refactoring Detection in Commit History. In: Proceedings of the 40th International Conference on Software Engineering (ICSE '18), pp. 483–494. Association of Computing Machinery, New York,(2018)
27. GitHub Homepage, https://github.com/. Last accessed 30 Sep 2021
28. Jira Homepage, https://www.atlassian.com/it/software/jira. Last accessed 30 Sep 2021
29. Google Cloud Platform Homepage, https://cloud.google.com. Last accessed 30 Sep 2021
30. Python Homepage, https://www.python.com. Last accessed 30 Sep 2021
31. Google Cloud Storage Homepage, https://cloud.google.com/storage. Last accessed 30 Sep 2021
32. Google Cloud Functions Homepage, https://cloud.google.com/functions. Last accessed 30 Sep 2021
33. Google Cloud Vertex AI Homepage, https://cloud.google.com/vertex-ai. Last accessed 30 Sep 2021
34. Google Cloud BigQuery Homepage, https://cloud.google.com/bigquery. Last accessed 30 Sep 2021
35. Data Studio Homepage, https://datastudio.google.com/. Last accessed 30 Sep 2021
36. Jenkins Homepage, https://www.jenkins.io/. Last accessed 30 Sep 2021
37. SonarQube Homepage, https://www.sonarqube.org/. Last accessed 30 Sep 2021
38. Ptidej GitHub Repository, https://github.com/ptidejteam/v5.2. Last accessed 30 Sep 2021