2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) | 978-1-6654-4456-9/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSTW52544.2021.00048

Supervised Learning for Test Suit Selection in Continuous Integration

Ricardo Martins
OutSystems
Instituto Superior Técnico
University of Lisbon, Portugal
ricardo.m.pires.martins@tecnico.ulisboa.pt

Rui Abreu
INESC-ID
University of Porto
Porto, Portugal
rui@computer.org

Manuel Lopes
INESC-ID
Instituto Superior Técnico
University of Lisbon, Portugal
manuel.lopes@tecnico.ulisboa.pt

João Nadkarni
OutSystems
Lisbon, Portugal
joao.nadkarni@outsystems.com

***Abstract*—Continuous Integration is the process of merging code changes into a software project. Keeping the master branch always updated and unfailingly is very computationally expensive due to the number of tests and code that needs to be executed. The waiting times also increase the time required for debugging. This paper proposes a solution to reduce the execution time of the testing phase, by selecting only a subset of all the tests, given some code changes. This is accomplished by training a Machine Learning (ML) Classifier with features such as code/test files history fails, extension code files that tend to generate more errors during the testing phase, and others. The results obtained by the best ML classifier showed results comparable with the recent literature done in the same area. This model managed to reduce the median test execution time by nearly 10 minutes while maintaining 97% of recall. Additionally, the impact of innocent commits and flaky tests was taken into account and studied to understand a particular industrial context.**

***Index Terms*—Continuous Integration, Test Selection, classifier model, flaky tests, innocent commits**

I. Introduction

The software complexity is directly proportional to its codebase size. During software development, there is a long and costly need for debugging. When doing so, a team of developers needs to write code, test it, commit it to their repository and possibly correct some errors after the execution of the batches of tests. The practice of merging all developers' working copies to a shared mainline is referred to as Continuous Integration (CI).

As the software complexity increases, the time it takes to test for bugs also increases. Ultimately, this increases the computational cost for testing and delays the work of developers who do not receive feedback until all their commits pass through all the tests.

This work is integrated in an OutSystems environment. OutSystems is a software company that develops a product which allows the creation of web and mobile apps using a low-code platform. Given the current situation of the regression testing at OutSystems, in which developers have to wait nearly 1 hour to receive feedback on their commits, we present a solution that tackles the problem of the excessive execution time of test suites. This approach addresses this problem by selecting a subset of test cases that are more likely to generate fails given a new code submission. This subset of selected tests could be executed in a pre-commit stage (e.g., on a developer's local machine), giving faster feedback to developers on their (possible) faulty changes.

The solution proposed in this paper is based on a test suite selection using a machine learning approach, taking advantage of features related to the test suite and code files changed.

This work aims to improve two aspects of the CI process:

- Primarily, reduce the time that developers need to wait to receive feedback on which tests failed for their newly submitted commits.
- Secondly, reduce the computational costs of the current regression testing process, that includes re-running the entire test suite for every set of commits.

This document is organized as follows: **Introduction**: provides an overview of the motivation and the objectives for this work; **Related Work**: provides an overview of the state-of-art regression test selection techniques with an emphasis on approaches that use Supervised Learning; **Solution Proposal**: describes the architecture of the proposed solution for this problem; **Implementation**: provides an overview of the state-of-art regression test selection techniques with an emphasis on approaches that use Machine Learning; **Results**: describes the analysis of the results of our approach in the context of the OutSystems processes' of CI and Regression Testing; **Conclusions**: contains the evaluation methodologies to be used in this work regarding the performance of the solution.

II. Continuous Integration Example

In this section, we present a case of continuous integration, and how flaky tests and innocent commits are identified. We consider in particular the context at OutSystems but our approach is agnostic to the particular software being developed.

A. Test Selection

We consider a CI process as summarized in Fig. 1. When developers submit code to a work repository, it goes through a build process. After this, if the build is successful, it is assigned a Test Run, which contains a suite of tests. After these tests, a report is produced (Test Run Result) that identifies which tests

This work was developed at OutSystems - https://www.outsystems.com/. This research was supported by Portuguese Fundação para a Ciência e a Tecnologia through grants UID/CEC/50021/2019 and PTDC/CCI-COM/29300/2017, and by the EU H2020 RIA project iV4xr : 856716.

978-1-6654-4456-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSTW52544.2021.00048

failed. Only then are the developers identified and notified. This process takes approximately 1 hour.

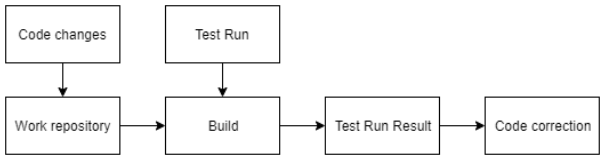

Fig. 1. From code submission to code correction (OutSystems)

In this process, there is no selection process for the test cases to be used. If a Test Run is selected to test a built project, all of the test cases in it are executed.

B. Flaky Tests

Ideally, any new test failures would indicate regressions caused by the latest changes. However, some test failures may not be due to the latest changes but due to non-determinism, which makes test outcomes unpredictable. In other words, the outcomes of an unmodified test may differ at different times, given the same code. These tests are often called flaky tests.

A practical example of a flaky test can be a test that tries to extract information from a remote server. If there are no synchronization mechanisms, where the test thread does not wait for the response of the server, and the response time varies, the test thread may succeed or not in extracting the information. Thus, making the test non-deterministically fail or pass.

C. Innocent Commits

In many situations, developers commit changes to the same branch of code, where test executions occur sequentially every time a commit arrives. Therefore, a commit may report failing tests that were already failing due to a previous commit. In this case, this commit should be tagged as an innocent commit to evaluate the performance of the classifier fairly since the code change was not related to the tests that failed.

In [1], Daniel Correia applied this concept of innocent commits, by identifying and filtering them in the data set. He presents one strategy to identify innocent commits in a set of multiple commits, which he called Superset. The ”rule” of this strategy is ”if the previous commit’s set of failing tests is a superset of the current commit’s, then the current commit is innocent”. Therefore, to identify innocent commits following the Superset strategy, it is necessary to iteratively compare the set of failing tests from one commit to the previous one.

III. Related Work

In this section, we present the state-of-art techniques related to the most important subjects of this paper: Test Suite Selection, Feature Selection, and Flaky Tests.

A. Test Suite Selection

Rothermel *et al.* [2] provides insight in regression testing selection (RTS) techniques and presents a framework to classify these techniques which are based on four categories:

- Inclusiveness - the capability of the RTS to capture modification -revealing tests, i.e., tests that have a different outcome given a new change.
- Precision - the capability of the RTS not selecting tests that are not modification-revealing.
- Efficiency - measures space and time requirements of an RTS.
- Generality - the capability of the RTS to adapt to real-world situations (e.g. handle realistic program modifications).

Wei *et al.* in [3] present a study regarding the effectiveness of a test coverage quality metric (branch coverage) on software testing. The intuition is that covering branches relates directly to uncover faults. However, the results obtained by the authors show that branch coverage is not a good indicator for the effectiveness of a test suite, where the correlation between branch coverage and the number of uncovered faults reveals to be weak.

Machalica *et al.* in [4] proposes a different predictive test selection strategy using ML techniques. The authors make use of a data set of historical test outcomes to train a machine learning classifier model. This model then tries to predict the outcome of a test execution over some changes (pass or fail). Ultimately, this model helps to select a subset of tests to exercise on a particular code change. In their results the authors report that:

- They manage to catch over 95% of individual test failures and over 99.9% of faulty code changes[1].
- The test selection procedure selects fewer than a third of the tests that would be selected based on build dependencies.
- They also succeeded in reducing the total infrastructure cost of change-based testing by a factor of two.

Philip *et al.* in [5] present Fast-Lane, a system that performs data-driven test minimization. Although the authors describe Fast-Lane as a test minimization system, their work shows similarities to test selection techniques. The authors analyze, not only, test file logs as well as commit logs to save test resources and decrease time-to-deployment. The authors based their work on three different approaches towards predicting test outcomes and therefore saving test resources:

- Commit Risk Prediction - The authors’ train classification models to predict the complexity of a commit, i.e., which commits are more ”risky” than others.
- Test Outcome-based Correlation - The authors learn association rules that find test-pairs that pass together and fail together. Thus showing test-pairs that potentially test the same functionalities.

[1]a code change is marked faulty if any of the individual tests run in response to the code change fails

- Runtime-based Outcome Prediction - The authors estimate a runtime threshold for test files, i.e., they separate passed runs from failed runs based on their runtime.

Similarly, Paterson *et al.* in [6] propose a test case prioritization strategy based on defect prediction, a technique that analyzes code features – such as the number of revisions and authors — to estimate the likelihood that any given Java class will contain a bug. In their work, the authors compared their approach, called G-clef, with other coverage and history strategies and obtained better results by reducing the number of test cases required to find a fault.

B. Feature Selection

Memon *et al.* in [7] present a study done at Google, which aims to reduce test workload by avoiding the re-running of tests unlikely to fail. And second, to use test results to inform code development. Aided by a dependency graph with a file-level granularity, the authors empirically studied relationships between developers, their code, and test cases. This lead to the formulation of several hypotheses which then were examined. The authors managed to get some specific results and correlations within the context of the Google database:

- C++ files are more prone to cause test failures than Java files.
- Certain authors cause more test failures than others.
- Code files modified by multiple developers are more prone to test failures.

Philip *et al.* in [5], with FastLane, used historical data about test files and commits and used a total of 133 features to characterize commits, categorized in five types: File type and counts, change frequency, ownership, developer/reviewer history, and component risk. The authors found that the file types, code hotspots[2] and code ownership-based metrics increased the most the accuracy of the classifier model.

Machalica *et al.* in [4] in their predictive Test Selection approach train a ML classifier. Such a classifier is trained based on historical data. The classifier model is created based on three types of features:

- Change-Level: Change history for files, number of files touched in a change, number of tests triggered by a change, files extension, and number of distinct authors.
- Test-Level: Historical failure rates, associated project name (or namespace), and the number of tests.
- Cross-Features: distance (between test files and code files) in the build dependency graph and lexical distance between file paths (test and code files).

C. Flaky Tests

Machalica *et al.* in [4] filter flaky tests from a test suite by re-running a test ten times. They classify it as flaky if all the runs aren't coherent, i.e., if amongst all runs there are more than two different outcomes (pass and fail). In their results, the authors report that, by filtering these tests before training and evaluating the classifier model, the accuracy of their model improves considerably, where its ability to "catch" failed tests does not decrease.

Bell *et al.* in [8] describe a new technique to identify flaky tests called DeFlaker. DeFlaker can detect if a test failure is due to a flaky test without re-running it and with very low run time overhead. DeFlaker marks as flaky, tests in two situations: a test that changed from passed to failed and did not cover any code that changed; or a test that changed from failed to pass and was executed on unchanged code. The authors implemented DeFlaker for Java, integrating it with a popular build and test tools, and found 87 previously unknown flaky tests in recent projects and 4,846 flaky tests in old projects.

[2]Components with a high risk of failure generation

IV. Approach for Test Suit Selection

The solution presented focuses on training a ML classifier. Our goal is to use this classifier to return a small subset of tests that reveal all faults given a set of commits. In this section, we describe the various steps to achieve this classifier built to solve the problem that arises from settings of CI that include flaky tests and innocent commits.

A. Data Set and Features definition

The first step to create a classifier is to define a data set. The data set needs to aggregate information about the code changes done by developers and the tests that are run over this code (during the testing phase). So, each entry of the data set has features the CI logs regarding past test runs.

We consider some of the features already used in the past [9]. By analyzing the CI logs regarding past test runs, it is possible to extract some initial statistics related to test files, code files, and commits. These statistics led us to some features that were more correlated with the test run result outcome. The following list briefly describes each feature that was fed to the ML classifier:

- **Test failure rate** - number of testruns where a test fails, over the total number of testruns of a test.
- **Author failure rate** - number of testruns where an author is involved in failing testruns, over all testruns linked to her/him.
- **File failure rate** - number of testruns where a code file generates a failed testrun, over the total number of times it is submitted to a testrun.
- **File/test failure rate** - number of testruns where a test runs on a code file and fails, over the total number of testruns of a test on a code file.
- **Author/file failure rate** - number of testruns where an author submits a code file and generates a failed testrun, over the total number of submissions of that code file by that author.
- **Extension file type** - identification of the code file's extension.
- **Tokens shared file/test** - comparison between the name's test with a code file's name.
- **Number of distinct files changed** - number of code files submitted by an author in the commit stage.

- **Number of distinct authors** - number of authors responsible for each testrun.
- **File change history** - frequency which a code file is submitted by authors over 3 different time intervals (5, 14, 56 days).
- **Test failure rate history** - test failure rate over 3 different time intervals (5, 14, 56 days).
- **File failure rate history** - code file failure rate feature over 3 different time intervals (5, 14, 56 days).

Flaky tests and innocent commits are also taken into consideration. So in total, we have 3 data sets to train the ML classifier: unfiltered data set (No-filter data set); the data set filtered by flaky tests (Flaky-filter data set); the data set filtered by innocent commits (Innocent-filter data set).

Flaky tests: At each day, we identify a set of flaky tests and remove them from the data set. This is similar to what was done in [4], where the prediction accuracy of the classifier is analyzed by filtering the original data set and excluding the flaky tests.

Innocent commits: The same is done regarding innocent commits. Using the superset strategy proposed by Correia *et al.* [1], the innocent commits are excluded from the no-filter data set.

B. Classifier Models' Training

Once the data sets are created we need to select a particular classifier and train it.

Classifier Models' Baseline: Given the number of algorithms from which to choose to create this classifier, we made an initial study to select the ones more promising. We train each classifier in the training set of the No-filter data set. After training the classifiers, each of them is evaluated in a testing set from the no-filter data set. The classifiers which show the best results are then used in the whole process.

Hyper Parameter Tuning: Every algorithm used to create the classifiers depends on various parameters that can influence the results. For the classifiers that showed the best results in the previous phase, we perform a hyperparameter selection. In this process, the classifier is trained and evaluated with different sets of parameters. The evaluation is done using a validation set in order to maintain the testing set unseen by each model. At the end of the process, we have the best parameters for each algorithm from among the ones tested.

For each classifier, we do a hyperparameter selection in each of the 3 different data sets. This way, each process of tuning may come up with its parameters.

V. Implementation

We now present the whole process in practice including data gathering, calculation of feature values, and training of the ML classifiers. For each process mentioned, we also present the challenges faced and the decisions made.

A. Creation of the Data sets

To build our data sets with data from OutSystems, we first need to understand the structures behind this and similar processes of CI and Regression Testing. The database's tables relevant to create the data sets are the ones shown in Fig. 2.

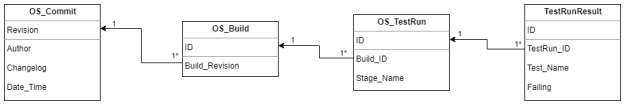

Fig. 2. Simplified view of the relevant database tables including information regarding developers commits, test suites to test newly added (or changed) code and the result of the execution of these test suites.

To build the data sets we use information from all the previous tables. Fig. 2 shows which parameters of the tables can be used. For example, we merge the tables **OS_Commit** and **OS_Build** through the parameters Revision and Build_Revision. The information relevant to build the data sets are the parameters Author, Changelog, DateTime, TestRunId, StageName, TestName, and Failing.

During this step, upon analyzing the results from the queries on the first 3 tables, we noticed that not all Revision values (from table **OS_Commit**) appear in the **OS_Build** table. The same happens with Build_ID values in the **OS_TestRun** table. The reason is that various commits may be aggregated into the same Build and the same for various Builds that are aggregated into the same TestRun. As commits and builds may be aggregated, a test's outcome of one TestRun may not depend exclusively on one commit. So, when building our data set we concatenate all "Changelogs" and "Authors" from commits that are assigned to the same "TestRun_ID".

The process of creation of the flaky-filter data set is more complex as the flaky tests are identified daily, and so may change every day.

B. Classifier models' generation

Once we have finalized the construction process for our data sets, we split the data sets into training and testing sets.

Classifier Models' Baseline: We first train all classifiers with only the no-filter data set as our baseline classifier models.

The algorithms chosen to generate the baseline models are K-Nearest Neighbour, Logistic Regression, Random Forest, Balanced Random Forest. These algorithms are part of the sklearn python package[3]. Another algorithm used was the XgBoost (used in [4]), from the xgboost python package.

Balancing data sets: The data collected is very unbalanced. As expected, most test runs have more non-failing than failing tests. There is a majority of class 0 ("not failing") over class 1 ("failing") in the no filter data set, with a ratio of 59:1. We created over and under samplings of the no-filter data set to balance the frequency of each class. The over and under sampling changed the ratio failing / not failing to 1:1.

Note that, not all algorithms are suited for over and/or under-sampling, given that some already implement it in their training process over the training. In addition to the

[3]www.scikit-learn.org

under/over sampling techniques we also explored the used of class weights to tackle the unbalanced nature of the dataset.

For each model training and testing phase, the sets used to train and test have the same filter type, i.e., if a model uses the unfiltered data set in the training phase, then it uses the unfiltered data set in the testing phase.

Hyper Parameter Tuning: The process of Hyper Parameter Tuning requires an analysis of the algorithms to see their parameter and the possible values for each parameter. To perform this process of tuning, we used the Bayesian Optimization [10].

The Hyper Parameter Tuning function allows us to maximize different metrics such as accuracy, precision, and recall. One important detail is that the set of arguments returned are the best set of parameters out of those defined in the set of parameter values to search for each algorithm. Hence, these may not be the optimal set of parameters.

An important component of the Hyper Parameter Tuning is the usage of K-Fold cross-validation, which prevents overfitting. In this tuning process, there is a validation phase for every set of hyperparameters, completed after every training phase. But, the validation set used to test each model's iteration can not correspond to the actual testing set of our data sets, to maintain the actual testing set "unseen" by the classifier model. Instead, with K Fold cross-validation, the training set is divided into k random subsets. Now, in each iteration of hyperparameter values, the model's training and testing are repeated k times, such that each time, one of the k subsets is used as the testing set and the other k-1 subsets form a training set.

However, there is a particularity in our data set, where there is a timeline throughout our data set entries, i.e., each features' value depends on the previous one. For example, an author will have a different failure rate through our data set because this feature, such as many others, is dynamic. Therefore, in cases where there is a temporal dependency between observations, we cannot choose random samples and assign them to either the validation set or the train set. In other words, we want to avoid "looking in the future" during the training of the model.

Considering this nuance, we need to use a variation of the previously mentioned cross-validation method called Time-series cross validation [11]. With this cross-validation method, we are still dividing our training set into K folds, but each time we only use sequential folds as our training and validation sets, without "looking to the future". Fig. 3 sums up the steps of the tool's creation.

VI. Results

A. Data Analysis

Our data sets include testruns from March 1st, 2020 to April 8th, 2020. When performing a split on the data sets to create the training and testing set, we considered for the training set the testruns from March 1st to March 31th, and for the testing set the testruns from April 1st to April 8th [4].

[4]This split corresponds to a 80/20 split of the data set

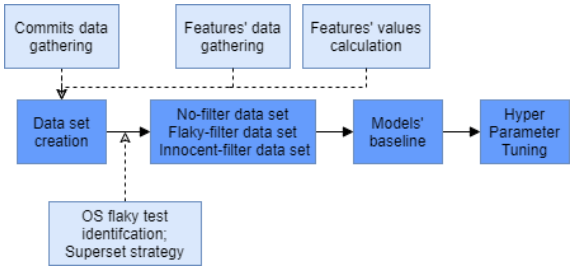

Fig. 3. Solution's pipeline

The file types were also selected. This research case study uses data from OutSystems' IDE: Service Studio. Therefore, given that testruns may aggregate various files, all of testruns in our data sets have at least one Service Studio code file with file extensions of .cs and/or .ts/.tsx (files that contain functionality).

The other component of our data sets is the test files for each testrun. In order not to overcrowd our data sets and to maintain our focus on the Service Studio component, we select specific stages of tests to be part of our data sets: Development and CorePlatform. The Development stage contains 5910 test files and the CorePlatform stages contain 6018 test files. The median execution time of the stages is 2500 and 450 seconds for CorePlatform and Development, respectively.

In terms of size, our data sets have the following number of entries/testruns: Unfiltered data set - 5,703,999 entries / 956 testruns; innocent-filter data set - 4,465,973 entries / 748 testruns; flaky-filter data set - 5,191,173 entries / 956 testruns.

B. Evaluation methodology

When evaluating the classifier model results, for the problem we have in hand, we prioritize high values of recall over precision. The recall relates to the ability of a classifier model to correctly identify the positive values (tests that fail). On the other side, precision relates to the ability to correctly distinguish the positive values from the negative values. For the problem in hand, given a large number of tests per stage (there are approximately 5000 tests), we prioritize classifiers that can correctly identify more positive values (tests that fail). In other words, we do not mind if the classifier incorrectly selects some tests that pass if he can select the maximum number of failing tests. Notice that the recall values that are analyzed are the ones highlighted in red because they are the ones related to the identification of the positive values (tests classified as "failed"). So the evaluation guideline during the training of the baseline and balanced models is to first compare recall values (1).

$$Recall = \frac{\text{\#failing tests selected}}{\text{\# failing tests}} \quad (1)$$

Although recall is one of the most important metrics to consider the reliability of each model, it can not be used to evaluate the models' results alone. Remembering one of the goals of this solution is to reduce the developers' feedback

time, we use the median execution time of the selected tests to complement the recall metric. This way, we evaluate each models' results mainly by doing a trade-off between these two metrics' values. This way, we can find the most reliable model, which reduces the test execution time of a testrun. Besides the recall, we calculate the average micro-recall, which refers to the recall in each stage individually.

The micro-recall equation is defined as:

$$Micro - Recall(n) = \frac{\text{\#failing tests selected in TR(n)}}{\text{\# failing tests in TR(n)}} \quad (2)$$

TR(n) refers to the nth testrun and Micro-Recall(n) refers to the recall of the nth testrun. The metrics calculated are differentiated by stage, i.e., we calculate the metrics values for stages Core Platform and Development separately. The reason why we choose median over the average for the execution time of the selected test suit is that the first is more resilient to outliers than the second.

C. Experiments

After the training of the baseline and balanced models, the models which stood out were:

- Model 1 - Logistic Regression (Oversampled) trained and tested with the no-filter data set - 92% recall
- Model 2 - Logistic Regression (Balanced) trained and tested with the no-filter data set - 92% recall
- Model 3 - Balanced Random Forest trained and tested with the no-filter data set - 92% recall
- Model 4 - Balanced Random Forest trained and tested with the innocent-filter data set - 93% recall

After these results, we calculated the other metrics for these 4 classifier models. Table I shows these metrics' values.

TABLE I
MODELS RESULTS

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)	
		CP stage	Dev. stage	CP stage	Dev. stage
LR (OS) / no filter	91.84%	88%	91%	1042	35
LR ("balanced") / no filter	92.00%	88%	93%	1098	50
BRF / no filter	92.45%	90%	95%	1189	165
BRF / innoc-filter	93.36%	80%	95%	1041	64

During the models' testing phase there is a threshold that is defined, which allows the model to determine the class of each test (failing vs. not failing). This value ranges from 0 to 1. The results from Table I correspond to the threshold defined as 0.5. Therefore, we decided to test the variation of the threshold value on the metric values. It is expected that, by increasing/decreasing the threshold value, the number of tests selected decreases /increases, respectively. By doing this, the metrics mentioned above change and we can obtain better results.

Also, instead of just showing a table with values, we decided to plot the variation of thresholds (as x) with the average micro-recall values and the execution time of the selected tests (as y1 and y2, respectively). For each classifier model, we plotted 2 graphs, one for each test stage.

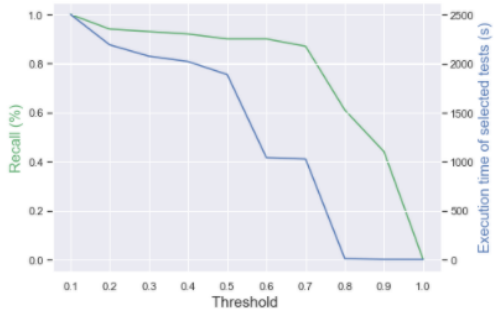

Fig. 4. Variation for the BRF (no filter data set) classifier model - CorePlatform stage

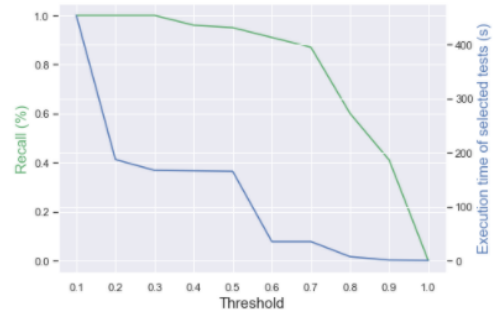

Fig. 5. Variation for the BRF (no filter data set) classifier model - Development stage

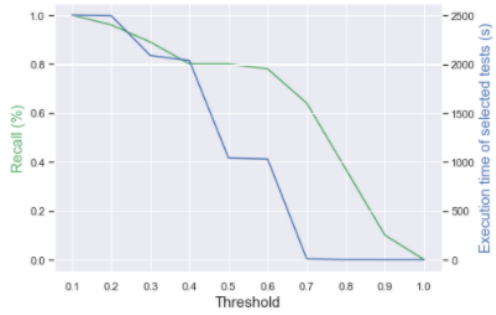

Fig. 6. Variation for the BRF (innocent filter data set) classifier model - CorePlatform stage

Regarding each stage individually, the results were similar in all classifier models. Regarding the CorePlatform stage, the one which gets better results is model 1, where the execution time of the stage reaches 2194 seconds while maintaining 94% micro-recall. Behind, this one comes the Logistic Regression classifiers with 93% micro-recall values but with 2000 seconds

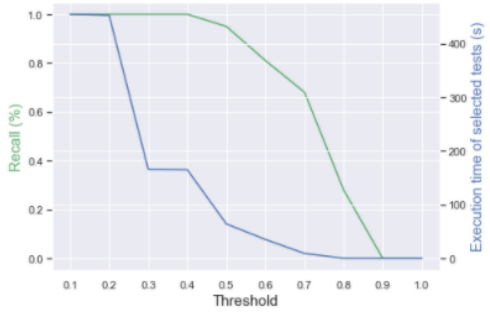

Fig. 7. Variation for the BRF (innocent filter data set) classifier model - Development stage

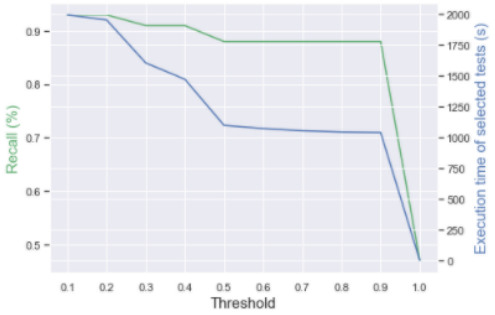

Fig. 8. Variation for the LR ("balanced" no filter data set) classifier model - CorePlatform stage

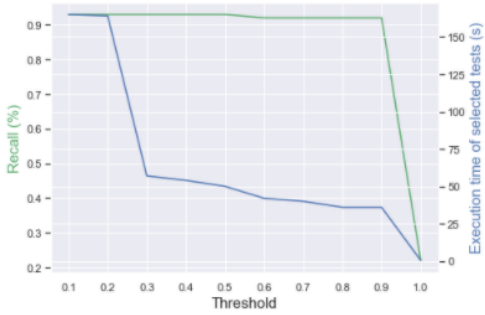

Fig. 9. Variation for the LR ("balanced" no filter data set) classifier model - Development stage

of test execution time. And in last comes the Balanced Random Forest-innocent filter data set with 89% micro-recall and 2091 seconds of test execution time.

In terms of the Development stage, we saw that the classifiers 1 and 2 models managed to achieve 100% of micro-recall for this stage while reaching execution times below 200 seconds (160 seconds). This means that the median execution time of this stage is cut down to more than half. In the Logistic Regression classifiers, we saw a bigger cut down from this stage median execution time, with test execution times below 50 seconds. However in both classifiers, to achieve these values, the micro-recall is not superior to 95%.

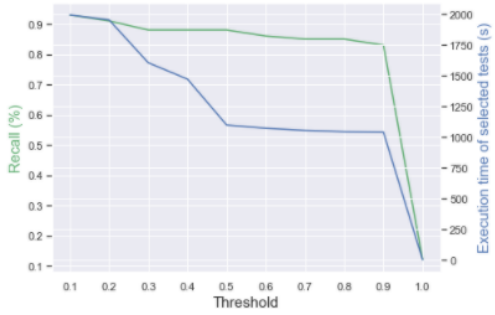

Fig. 10. Variation for the LR (oversampled no filter data set) classifier model - CorePlatform stage

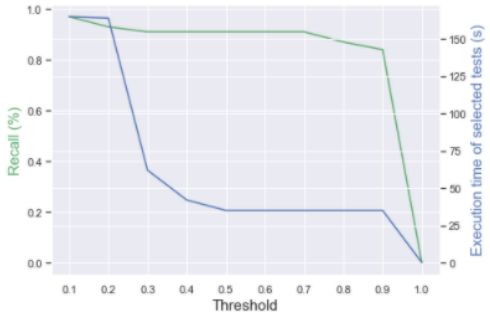

Fig. 11. Variation for the LR (oversampled no filter data set) classifier model - Development stage

Another experiment we decided to implement is one that fits more into the real world and the developers' necessities when trying to test their code - the Time limit variation. We did this by ordering the tests of each model's predictions by their predicted probability and put a limit on the execution time of that list of tests [5]. Given the results of all the models from the previous experiment, we selected four pairs of time limits for each classifier model. And then, calculated the metrics' values using all 4 pairs of time limits for all models.

Table II shows the metrics' values for one of the pair of time limits: Core Platform - 2200 secs, Development - 167 secs.

Classifier model / data set used	Recall (%)	Average micro-recall (%)	
		CP stage	Dev. stage
LR (OS) / no filter	97.70%	97%	94%
LR ("balanced") / no filter	97.40%	98%	97%
BRF / no filter	97.66%	94%	97%
BRF / innoc-filter	96.98%	93%	89%

TABLE II
TIME LIMIT VARIATION RESULTS - CORE PLATFORM - 2200 SECS, DEVELOPMENT - 167 SECS

[5]each test has an execution time value calculated from previous executions

VII. Conclusions

This work presented an approach to improve CI pipelines in the presence of flaky and innocent commits with continuous commits. Our results show that we can learn which tests are more relevant to be used to reduce the computational time of testing and the time to feedback for developers. We tested our system with real data from OutSystems IDE (Service Studio) code where the CI process takes around 1 hour.

From all the classifier models trained, there were four that stand out from the rest: the Balanced Random Forest classifiers trained with the no filter (1) and the Innocent-filter data set (2); the Logistic Regression classifier model trained with an Oversample of the no filter data set (3); and the Logistic Regression "balanced" classifier trained with the no filter data set (4). Right after the tuning process, it was clear that the classifier model (2) was the most promising since it had the highest recall values (93%), while being able to reduce the median execution time of the test suite . However, when varying the models' threshold values and calculating more specific metrics, we saw different outcomes. In the Development stage, the classifier (1) showed the best results, achieving 100% of micro-recall, while reducing the median test execution time by more than half (down to 167 seconds).

Regarding the CorePlatform stage, the classifiers (1), (3), and (4) had pretty similar results with micro-recalls of $93\% - 94\%$ and reducing the median test execution time to 2100-2000 seconds. However, in the mark of the 1000 seconds of execution time for this stage, the classifier (1) achieved better micro-recall values with values of 90%.

Regarding the time limits experience, the time limits which showed the best recall values were the 2200 and 167 for CorePlatform and Development stages, respectively. Given these results we can see that, by limiting the test stages with these values, we manage to achieve recall values near 98%, for classifiers (1), (3), and (4), while reducing the CorePlatform stage execution time by 300 seconds (-12%) and the Development stage execution time by 283 seconds (-63%).

Machalica *et al.* [4] ML model guarantees that over 95% of individual test failures are reported while reducing the total infrastructure cost of testing code by a factor of two, applied to a larger code base, than the one covered by this work. However, their work is backed by a dependency graph, which allows a primary filter of which tests are more likely to fail.

Regarding the OutSystems environment, this work is preceded by two other works in the Test Selection area. These works were based on static and dynamic dependency analysis between code and test files [12] and test coverage metrics [1]. Both works did not show great results as they faced some challenges due to a poor mapping of external calls from test files execution and code coverage bottleneck, respectively.

Summing up, the results presented are promising for a possible integration of this tool in CI pipelines. Given the results of all classifier models, the one chosen to be used in a future iteration of this work tool is the Balanced Random Forest-no filter data set model. More specifically in the OutSystems environment, the inclusion of the tests from all test stages in the data sets would be the first step to this tool implementation.

Future work could include the addition of warning messages for developers in a pre-commit stage, similar to what Memon *et al.* proposed [7]. Arik *et al.* in [13] propose a new canonical Deep Neural Network architecture for tabular data, TabNet, which could also be explored as future work for this work as it deals with tabular data. Also, a different approach could involve the use of Contextual Bandits [14] to support the test selection process, which have been used in recent state of the art literature in recommendation systems.

References

[1] D. Correia, R. Abreu, P. Santos, and J. Nadkarni. Applying multi-objective test selection for contin-uous integration at outsystems. Master's thesis, Instituto Superior Tecnico , 2019.

[2] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques.IEEE Trans.Software Eng., 22(8):529–551, 1996.

[3] Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In Empirical Software Engineering and Verification - International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures, pages 194–212, 2010.

[4] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice,ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019, pages 91–100, 2019.

[5] A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagappan. Fastlane: test minimization forrapidly deployed large-scale online services. InProceedings of the 41st International Conferenceon Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pages 408–418,2019.

[6] Paterson, D., Campos, J., Abreu, R., Kapfhammer, G. M., Fraser, G., McMinn, P. (2019, April). An empirical study on the use of defect prediction for test case prioritization. In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST) (pp. 346-357). IEEE.

[7] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In39th IEEE/ACM International Conference on Software Engi-neering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May20-28, 2017, pages 233–242, 2017.

[8] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: automatically detect-ing flaky tests. InProceedings of the 40th International Conference on Software Engineering, ICSE2018, Gothenburg, Sweden, May 27 - June 03, 2018, pages 433–444, 2018.

[9] Ricardo Martins, Test suite selection guided by machine Learning. Masters' thesis, Instituto Superior Tecnico, 2019.

[10] Wang, H., Taylor, S. R., Vallisneri, M. (2019). Bayesian cross validation for gravitational-wave searches in pulsar-timing array data. Monthly Notices of the Royal Astronomical Society, 487(3), 3644-3649.

[11] Bergmeir, C., Hyndman, R. J., Koo, B. (2018). A note on the validity of cross-validation for evaluating autoregressive time series prediction. Computational Statistics and Data Analysis, 120, 70-83.

[12] Pereira, P. M. (2015). Analysis of Network Attacks and Security Events using Modern Data Visualization Techniques (Doctoral dissertation).

[13] Arik, S. O., Pfister, T. (2019). Tabnet: Attentive interpretable tabular learning. arXiv preprint arXiv:1908.07442.

[14] Slivkins, A. (2011, December). Contextual bandits with similarity information. In Proceedings of the 24th annual Conference On Learning Theory (pp. 679-702). JMLR Workshop and Conference Proceedings.