Supervised Learning for Test Suit Selectionin Continuous Integration at OutSystems

Ricardo Martins
MEIC-T student
Instituto Superior Técnico - University of Lisbon
Lisbon, Portugal
ricardo.m.pires.martins@tecnico.ulisboa.pt

Rui Maranhão
IST Computer Science department
Instituto Superior Técnico - University of Lisbon
Lisbon, Portugal
rui.maranhao@tecnico.ulisboa.pt

Manuel Lopes
IST Computer Science department
Instituto Superior Técnico - University of Lisbon)
Lisbon, Portugal
manuel.lopes@tecnico.ulisboa.pt

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

***Abstract*—Continuous Integration is the process of merging code changes into a software project. This mechanism of keeping the master branch of a project always updated and unfailingly, raises problems in terms of computational costs, considering the enormous amount of code existent in large software systems that needs to be tested first. Given this situation, the work of developers also becomes harder because of the amount of time they have to wait for feedback on their commits - median of 50 mins.**

Recognizing this problem in an Outsystems context, this paper proposes a solution that aims 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 Classifier with features such as code/test files history fails, extension code files that tend to generate more errors the during testing phase and others.

The results obtained by the best Machine Learning classifier trained showed great results which could be compared to recent literature done in the same area. This model managed to reduce the median test execution time by nearly 10 minutes while mantaining 97% of recall. Additionally, the impact of innocent commits and flaky tests was taken into account and studied to understand the industrial context of OutSystems.

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

I. Introduction

A. Motivation

Normally, in a software company, the software complexity is directly proportional to its code base size. During software development there is a long and costly need for debugging. When doing so, a team of developers need to write code, test it, commit it to their repository, and possibly correcting after the execution of 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 if a software is according to the specified standards of a company, also increases. This will ultimately increase the computational costs and delay the work of developers, who will not receive feedback on their commits during the time they are focused on the problem, which makes them lose track of the work done.

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 tries to solve this by selecting a set of test cases that are more likely to generate fails given a new code submission. This set of selected tests should 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 will be based on a Test Suite Selection using a Machine Learning approach, using features related the test suite and code files changed. This features will be described in detail further ahead.

Although, this thesis is integrated in an OutSystems environment, its conclusions should help future applications of test suite selection approaches based on Machine Learning techniques.

B. Objectives

Given the OutSystems context of this thesis, it aims to assist in two aspects of the CI process at OutSystems:

- Primarily, reduce the time of the developers' feedback loop, i.e., 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 method of OutSystems' regression testing process, of re-running the entire test suite for a set of commits.

OutSystems - https://www.outsystems.com/

C. Organization of the Document

This document is organized in 7 sections which can be summarized as follows:

1) **Introduction**: provides an overview of the motivation and the objectives for this thesis.
2) **Related Work**: provides an overview of the state-of-art regression test selection techniques with an emphasis on approaches that use Supervised Learning.
3) **Solution Proposal**: describes the analysis done of the OutSystems processes' of CI and Regression Testing, as well as the architecture of the proposed solution for this problem.
4) **Implementation**: provides an overview of the state-of-art regression test selection techniques with an emphasis on approaches that use Machine Learning.
5) **Results**: describes the analysis done of the OutSystems processes' of CI and Regression Testing, as well as the architecture of the proposed solution for this problem.
6) **Conclusions**: contains the evaluation methodologies to be used in this work regarding the performance of the solution.

II. OutSystems context

A. Test Selection

At OutSystems the process of re-running all tests for a given project takes up to 1 hour. In the CI process of OutSystems (summarized in Fig.1), when developers submit code to their work repository, it needs to go through a Build process. After this, if the build is successful, it is be assigned a Test Run, which contains a suite of tests. This Test Run is run over the built project and afterwards it will return a "report" (Test Run Result) that includes which tests failed. Then the developers are notified, and proceed to rectify (if needed) the code faults, which they are responsible for.

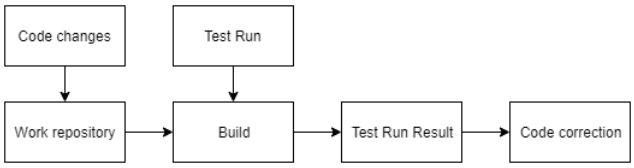

Fig. 1. From code submission to code correction (OutSystems)

Given the complex product developed at OutSystems, which is constantly being upgraded, there are several stages that represent different releases. Inside each stage, there are also different projects.

As mentioned above, when a project is successfully built, a Test Run containing test files are run over this project. For a specific stage and project there is a Test Run assigned, which will test the built project that includes the new changes. However, there is no selection process for the test cases in the Test Run given a code change, meaning that if a Test Run is selected to test a built project, all of the the test cases in it will be executed. Given the size increase of the code base at OutSystems, the number of the test cases in each Test Run tends also to increase and consequently the time to test a successful Build increases. In order to save time and resources of tests execution and help developers receive feedback in a shorter period of time, Test Selection techniques represent a good option to aid in this recurrent problem.

B. Flaky Tests

At OutSystems, flaky tests are identified in a more simple manner that does not require the re-running of tests when these fail. By not re-running these tests, OutSystems saves some computational cost. The way they do it is by analysing the behaviour of all tests during previous regression testing phases. As mentioned before, OutSystems keeps an history of the outcome of all tests. By analysing the last 25 executions of a test, OutSystems ranks tests based on the intermittency in their executions. This ranking is done based on the following metrics:

- 1 point every time a test passes with retry[1].
- 2 points every time a test has the execution pattern pass-fail-pass.

The tests that "score" the most points are considered to be more flaky than the others[2]. However, this approach does not totally guarantee that the tests marked are indeed flaky, because the executions analysis does not take into account the code which the test was covering. One test may get a high ranking only by "scoring" only on the second metric above. And there is no guarantee that the test is not failing due to faults introduced by changes in the code. The main purpose of this approach, at OutSystems, is to let developers know which tests, in the present, have been showing the most intermittence and encourage them to rectify these tests.

C. Innocent Commits

At OutSystems, developers commit changes to the same branch of code over where sequential test executions occur sequentially every time a commit arrives. Therefore, it may happen that a commit reports failing tests that were already failing due to a previous commit. In this case, this commit should be tagged as an innocent commit for the purpose of evaluating the prediction accuracy of the classifier fairly since the code change was not related with the tests that failed.

In [1], Daniel Correia applied the concept of innocent commits, by identifying and filter them in his data sets. 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 super set of the current commit's, then the current commit is innocent". Putting it in a simpler way, for a commit to be innocent its set of failing tests has to be in the set of failing tests from the previous commit. Therefore, to identify innocent commits following the Superset strategy, it is necessary to

[1]Important to mention that when a test file fails, it is immediately set for another execution (retry).

[2]The score can only be as high as 25, given that only the last 25 executions are analysed

iteratively compare the set of failing tests from one commit to the previous one.

III. Related Work

In the next section it will be presented a briefly overview of 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. in [2] provides insight on the issues in regression testing selection (RTS) techniques and presents a framework to classify these techniques which is based on four categories:

- Inclusiveness - capability of the RTS to capture modification -revealing tests, i.e., tests that have a different outcome given a new change.
- Precision - capability of the RTS not selecting tests that are not modification-revealing.
- Efficiency - measures the space and time requirements of an RTS.
- Generality - capability of the RTS to adapt to real world situations (for 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 uncovering 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 (passed or fail). Ultimately, this model will help selecting 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 (a code change is marked faulty if any of the individual tests run in response to the code change fails).
- The test selection procedure selects fewer than a third of the tests that would be selected on the basis of build dependencies.
- They also succeeded in reducing the total infrastructure cost of change-based testing by a factor of two.

However, the authors do not take into account the possibility of subsets of tests having overlapping coverage and thus correlated results. Such addiction to the predictive strategy could produce even better results.

Philip et al. in [5] present Fast-Lane, a system that performs data driven test minimization. Although the authors describe Fast-Lane as test minimization system, their work shows similarities to test selection techniques. The authors analyse, not only, test file logs as well has commit logs in order save test resources and decreasing 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.
- 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.

B. Feature Selection

Memon et al. in [6] 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 hypothesis which then were examined. The authors managed to get some specific results and correlations within the context of the Google database:

- Code files at higher distances than 10 from test files (in the dependency graph) do not cause test failures on those test files.
- Code files more often changed are more likely to appear in commits that generate test failures.
- 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[3] 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 number of tests.

[3]Components with high risk of failure generation

- Cross-Features: distance (between test files and code files) in 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 among 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 evaluation of the classifier model, the accuracy of their model improves considerably, where its ability to "catch" failed tests does not decrease.

Bell et al. in [7] describe a new technique to identify flaky tests called DeFlaker. DeFlaker is able to 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 popular build and test tools, and found 87 previously unknown flaky tests in recent projects and 4,846 flaky tests in old projects.

IV. Solution Proposal

The solution presented in this thesis focus on training a ML classifier. The optimal solution for this classifier is to return the smallest subset of tests that reveal all faults given a set of commits. In this section it will be described the various steps to achieve this classifier built to solve the problem that arises from the current regression testing approach at OutSystems.

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 files changes done by developers and the tests that are run over this code changes (during the regression testing phase). So, each entry of the data set has features regarding the code files submitted in one commit, one test run over those code files and the class of the entry corresponds to the outcome of the test (pass or fail).

The work done in [8], helped us understanding the features that made sense to include in the data sets in the context of OutSystems. Of course, said work was backed by the latest literature regarding Test Selection techniques guided by ML. Nonetheless, with the analysis performed to the OutSystems' CI and Regression Testing processes, it was possible to extract some initial statistics related about test files, code files and commits. These statistics led us to some features and others were added after. The following list briefly each feature that made it into the data sets:

- **Test failure rate** - This feature relates to each test and refers to the number of times a test fails for all its executions.
- **Author failure rate** - This feature relates to every author and refers to the number of times an author is involved in failling testruns for all testruns linked to him/her.
- **File failure rate** - This feature refers to the number of times a code file generates a failed testrun compared to the total of times it is submitted to a testrun.
- **File/test failure rate** - This feature compares the number of times a test runs over a code file and fails with the total number of runs between the test and code file.
- **Author/file failure rate** - This feature compares the number of times an author submits a code file and generates a failed testrun with 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** - This feature compares the name's test with a code file's name.
- **Number of distinct files changed** - This feature simply determines the number of code files submitted by an author in the commit stage.
- **Number of distinct authors** - This feature represents the number of authors responsible for each testrun.
- **File change history** - This feature represents the frequency which a code file is submitted by authors in 3 different time intervals.
- **Test failure rate history** - This feature presents the same purpose of the test failure rate feature, however, like the previous feature we pre-define 3 time intervals.
- **File failure rate history** - This feature presents the same purpose of the code file failure rate feature, but similarly to the previous feature we pre-define 3 time intervals.

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

Flaky tests: Each day, OutSystems identifies a set of flaky tests. Therefore, for each day, the flaky tests will be removed from the data set. This removal effect will be studied similar to what was done in [4], where, basically, the prediction accuracy of the classifier is analyzed by filtering the original data set and excluding the flaky tests.

Innocent commits: Similarly, the same will be done regarding innocent commits. Using the strategy Superset, the innocent commits are excluded from the No-filter data set and the prediction accuracy of the classifier is analyzed.

B. Classifier Models' Training and Tuning

Once the data sets are created we need to select the algorithms to create the classifiers models.

Classifier Models' Baseline: Given the amount of algorithms from which to choose to create this classifier, we make a pre-selection of several algorithms. Each of these algorithms will produce a classifier by training the algorithms with the training set (No-filter data set). After the classifiers generation, each of them will be evaluated with the testing set (No-filter data set). The classifiers which show the best results will be chosen for the next steps of the solution pipeline production.

Hyper Parameter Tuning: Every algorithm used to create the classifiers depend on various parameters. Therefore, these parameters can influence the results shown by the classifiers. For this reason, for the classifiers that showed the best results in the Baseline section, we perform an hyper parameter tuning for these classifiers' algorithms. During this tuning process, the classifier is trained and evaluated iteratively with different sets of parameters. In the end of the process, we end with the best parameters for each algorithm, given the different set of parameters supplied. Important to notice that the parameters which we end up may not be the optimal ones. The following picture shows an example of the set of parameters used in the hyper parameter tuning of the Balanced Random Forest algorithm.

Given that we have 3 different data sets, for each classifier algorithm, we perform 3 hyper parameter tunings, where each of them are trained and evaluated with each of the data sets. This way, each process of tuning may come up with its own parameters, different from the others, eliminating bias from the parameters used.

V. Implementation

In this chapter we present a resume of the processes of data gathering for the data sets' assembly, calculation of features values and training of the ML classifiers. For each process mentioned, we also present the challenges faced and the decisions made.

A. Data sets creation

In order to build our data sets with data from OutSystems, we first need to understand the structures behind the OutSystems processes' of CI and Regression Testing. Recalling the OutSystems' CI process described earlier, OutSystems' database stores information regarding developers commits, test suites to test newly added (or changed) code and the result of the execution of these test suites. The database's tables relevant to create the data sets are the ones shown in Fig. 2.

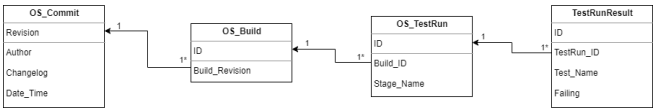

Fig. 2. Simplified view of the relevant database tables

To build our data sets we need to merge the information from all tables based on the common parameters between them. Fig. 2 shows which parameters of the tables can be used to relate these tables. 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 of the solution construction, one detail presented itself as a problem. Upon analysing the results from the queries on the first 3 tables, we noticed that not all Revision values (from table **OS_Commit**) appeared in the **OS_Build** table. The same happened with Build_ID values in the **OS_TestRun** table. After some investigation, we come to the conclusion that various commits may be aggregated into the same Build, and the same for various Builds that are aggregated into the same TestRun. This is due to the constant submission of code by developers. And, in order not to waste computational resources on every code change individually, commits and builds are aggregated. Therefore to deal with the problem of missing revisions and builds, something was done.

So, in the end we can see that, because commits and builds may be aggregated, a test's outcome of one TestRun may not depend exclusively on one commit. Instead, they may depend on the aggregate of builds, which may be composed by various commits. Hence, when building our data set we must concatenate all "Changelogs" and "Authors" from commits which are assigned to the same "TestRun_ID".

Now that we have our data set (no-filter data set), the next step is to create the two filtered data sets (Flaky-filter and Innocent-filter data sets) by filtering the original one.

The process of creation of the flaky-filter data set is a little trickier. As mentioned in a previous section, at OutSystems the flaky tests are identified everyday at midnight. Hence, to build our flaky-filter data set, first we need to retrieve the data relative to the flaky tests from the OutSystems database.

B. Classifier models' generation

Once we have finalized the construction process for our data sets, the first step before we starting the classifiers training is to split the data sets into training and testing set. The training set is the part of the data set which will be used to generate the classifiers and train them. The testing set is used to evaluate the classifiers' prediction accuracy.

Classifier Models' Baseline: The first step is to train all classifiers with only the no-filter data set as our baseline classifier models. This process is done by selecting various algorithms which will generate our classifiers. The algorithms are used without any arguments so that we can have a baseline for each of the classifiers generated by each one.

The algorithms used to create the classifier models were:

- K-Nearest Neighbour
- Logistic Regression
- Random Forest
- Balanced Random Forest
- Xgboost

Balancing data sets: Our data sets, showed some unbalance, which is normal given that, for all testruns its expected to have more non failing failing tests than failing tests. There is a majority of class 0 ("not failing") over the class 1 ("failing"). The ratio between the classes in the no filter data set is 59:1. Hence, besides not using any arguments like in our baseline models, we also create over and under samplings of the no-filter data set to balance the frequency of each class in our data sets.

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. Also, there are some

algorithms with a ”balanced” argument, so we also used it to balance the data sets.

Therefore, we create additional model classifiers using different versions of the no-filter data set (O-Sample no-filter data set, U-Sample no-filter data set). We also use the ”balanced” argument (whenever makes sense) to generate a model classifier.

Hence, besides the classifier models generated as our baselines, we also generated others by sampling the no-filter data set using the various techniques presented above.

Hyper Parameter Tuning: The next steps are to select the classifiers which showed the best results and submit each ones algorithm an Hyper Parameter Tuning. This way, we find a better set of arguments and get better results for each model classifier.

The algorithms chosen above were using only the no-filter data set to train the classifiers. But remember that these are still not the best possible results even for this data set, given that the parameters used were all the default ones. Hence, the next step is to submit these algorithms to an Hyper Parameter Tuning process. Now, we must provide these data sets to the algorithms above and check for improvements regarding the no-filter data set and check the first results for the filtered data sets. Something interesting to assess, is if the filtering process brings any improvements to the classifiers prediction accuracy.

The process of Hyper Parameter Tuning requires an analysis over the algorithms to see their parameter and the possible values for each parameter. To perform this process of tuning, we used the Bayes Search Cross Validation function. The idea is that we collect a set of parameter values for each algorithm's parameter and iteratively run the algorithms every time with different parameter values. Summing up, we need to run 3 hyper parameter tuning processes for each algorithm above, using the 3 different data sets to iteratively train the algorithm's classifiers.

The Hyper Parameter Tuning function allows us to maximize different metrics such as accuracy, precision and recall. As explained before, we prioritize recall, thus we need to define the argument ”scoring” of the Bayes Search function as ”recall”. So, in the end of each tuning iteration, the arguments returned are the ones the maximize the recall for each pair of algorithm-data set. 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 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 testing phase for every set of hyper parameters, completed after every training phase. But, the testing set used to test each model's iteration can not correspond to the actual testing set of our data sets, in order 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 hyper parameter values, the model's training and testing is repeated k times, such that each time, one of the k subsets is used as the testing set and the other k-1 subsets are put together to form a training set.

However, there is a particularity in our data set, where there is a timeline through out our data set entries, i.e., each feature's value depend on the previous one. For example, an author will have different failure rate through our data set because this feature, such as many others, are dynamic. Therefore, in cases where there is temporal dependency between observations, we cannot choose random samples and assign them to either the test 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 previous mentioned cross validation method called Time-series cross validation. With this cross-validation method we are still dividing our training set into K folds, but in each time we only use sequential folds as our training and testing sets, without ”looking to the future”.

VI. Results

A. General Details

Before entering in the experiments with the model classifiers, first we need to present some general details regarding the data extracted from the OutSystems database, the building process of the data sets and of training of the model classifiers, and the methodology behind the evaluation of each model classifier.

Our data sets include testruns from March 1st 2020 to April 8th 2020. When performing this split on the data sets to create the training and testing set, we did it such that the testing set includes testruns from March 1st to March 31th. And the testing set includes testruns from April 1st to April 8th.

Another detail, regarding the testruns included in the experimental data sets, which we needed to look out for was the types of files in each testrun. The thesis' research application focuses on the OutSystems' main software component: 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 are the test files for each testrun. In order not to overcrowd our data sets and to maintain our focus on the Service Studio application, 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 contains 5910 test files. The median execution time of the stages are 2500 and 450 seconds for CorePlatform and Development, respectively.

B. Evaluation methodology

The training of the baseline and balanced models were performed to assess the most promising model classifiers.

When evaluating the classifier models results, for the problem we have in hands, we prioritize high values of recall over precision. Recall relates to the ability of a classifier model to correctly identify the positive values (tests that fail). Where

precision relates to the ability of correctly distinguish the positive values from the negative values. For the problem in hands, given the large amount of tests per stage there are (approximately 5000 tests), we give more value to classifiers, which show the best ability to identify all positive values, rather than to distinguish between the positives and negatives. In other words, we do not mind if the classifier selects a few negatives values (tests that do not fail) as tests probable to fail, while selecting the maximum number of failing tests. Notice that the recall values that are analyzed is the one 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).

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

And so, the evaluation of the results from the baseline and balanced models take only into account the values of recall from all the model classifiers.

Although recall is one of the most important metrics to considerate, when comparing the classifier models results, it must be complemented with other important metric: the execution time of the selected tests. When training our models we do not want one that optimizes the results based solemnly on recall, because for that we would select all tests we would achieve perfect recall values. Remembering one of the goals of this solution of reducing the developers' feedback time, we use the median execution time of the selected tests to complement the recall metric and we need to do a trade-off between this metrics.

After the hyper parameter tuning of all classifiers, the evaluation guidelines are more complete.

Once we get the returned parameters from the hyper parameter tuning process, we train the correspondent classifiers. Since we are in the final stage of evaluation, we do not want to only evaluate each classifier by its recall.

To produce more specific data and choose a model classifier to use in the final product of the solution tool, we decided to evaluate the classifiers performance for each testrun. Instead of just looking at the macro recall (recall over all data set) returned by the classifiers' classification report, we, additionally, calculate the following metrics:

- Average micro-recall (2) per testrun
- Median of selected tests per testrun
- Median of failing tests per testrun
- Median of number of times the classifier selects at least one test per testrun
- Median of execution time of the tests selected per testrun
- Median of time saved per testrun

The micro-recall equation is defined in (2).

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

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 average in the majority of the metrics (except micro-recall) is because the first is more resilient to outliers than the second.

An important thing to notice about micro-recall is that, for this metric, only testruns which have failing tests count, because in cases where no tests fail the micro-recall would be zero for that testrun. By doing this, we eliminate these outliers testruns. Using average instead of median would be more precise if we did not calculate the micro-recall as we explained, but by doing it this way the average micro-recall is equally reliable.

C. Experiments

Due to the limitation in the number of pages for this paper, we will show only the most important results, regarding only the classifier models trained after the Hyper Parameter Tuning process.

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.

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

Classifier model / data set used	Median of time saved (s)		Median # selected tests		Median % of times, at least, one test selected
	CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	1219	419	2181	93	100%
LR ("balanced") / no filter	1174	418	2245	143	100%
BRF / no filter	854	419	2590	1918	100%
BRF / innoc-filter	944	405	2176	245	100%

TABLE I
THRESHOLD VARIATION RESULTS WITH THRESHOLD AT 0.5

One experiment decided to implement in the evaluation of the classifier models is the variation of the threshold value for each one. 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). We chose recall and execution time metrics, given that these two are the ones whichare more representative of the classifier models performance. For each classifier model we plotted 2 graphs, one for each test stage.

Once again, given the limitation in pages we decided to include the graphs for each stage from one of the 4 classifier models - the Balanced Random Forest trained with the no-filter data set (Figures 3-4.

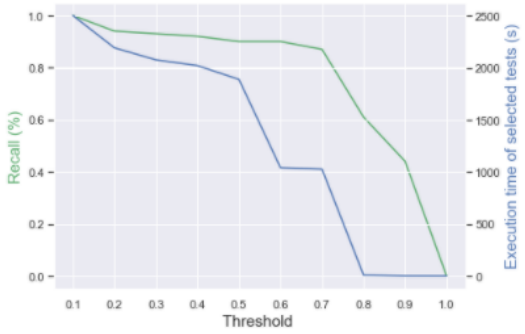

Fig. 3. Threshold variation for the BRF (no filter data set) classifier model - CorePlatform stage

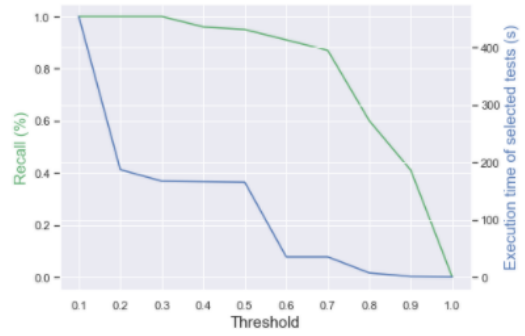

Fig. 4. Threshold variation for the BRF (no filter data set) classifier model - Development stage

Regarding each stage individually, the results were similar trough out all classifier models. Still, the one which gets better results is model 1, where the execution time of the stage reaches the 1000 secs (more than half of the median time), while maintaining 90% micro-recall. Behind, this one come the Logistic Regression classifiers with 88%-83% micro-recall values and 1000 seconds of test execution time. And in last comes the BRF-innocent filter data set with 80% micro-recall and 1000 seconds of test execution time.

In terms of the Development stage, we saw that the classifiers 1 and 2 models managed to achieve perfect micro-recall for the stage Development, while reaching execution times below the 200 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 the 50 seconds. However in both classifiers, to achieve these values, the micro-recall is no superior than 95%.

Another experiment 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 [4]. Given the results of all the models from the previous experiment, we set the best pair of time limits for each classifier model (one for each stage). And then, calculated the metrics' values using all 4 pairs 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%

Classifier model / data set used	Median # of selected tests'		Median % of times, at least, one test selected
	CP stage	Dev. stage	
LR (OS) / no filter	5311	1932	100%
LR ("balanced") / no filter	5453	1939	100%
BRF / no filter	5340	1942	100%
BRF / innoc-filter	5310	1943	100%

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

VII. CONCLUSIONS

A. Discussion

From all the classifier models trained, there were four which 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 Over-sample 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%). 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

[4]each test has a execution time value calculated from previous executions

(down to 167 seconds). Regarding the CorePlatform stage, the classifiers (1), (3) and (4) had pretty similar results with micro-recalls of 93% and reducing the median test execution time to 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 best recall and micro-recalls values were the 2200 and 167 for CorePlatform and Development stages, respectively. Given this 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%).

Summing up, the results presented his results are promising for a possible integration of this tool in the CI pipeline at OutSystems, and also that the implementation procedures could be applied in other companies' context. Given the results of all classifier models, the one chosen to be used in a future iteration of this thesis tool is the Balanced Random Forest-no filter data set model.

B. Contributions

The work developed in this thesis resulted in a contribution regarding the current CI pipeline at OutSystems. Although we only targeted the company's main component, Service Studio, this solution is exepected to work well if implemented across other OutSystems' departments which do no work directly with Service Studio.

C. Future Work

Throughout this thesis there were some approaches left aside due to limited time.

One example of an addiction to the work done would be to produce warning messages for developers in a pre-commit stage like Memon et al. do in [6]

Also, a different approach would be to use Contextual Bandits to support the test selection process.

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] 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.

[7] 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.

[8] R. Martins, R. Abreu, P. Santos, and J. Nadkarni. Test suite selection guided by machine Learning. Master's thesis, Instituto Superior Tecnico, 2019.

[9] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. InProceedingsof 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.

[10] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[11] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[12] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.