TÉCNICO
LISBOA

Supervised Learning for Test Suit Selection in Continuous Integration

Ricardo Miguel Pires Martins

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rui Filipe Lima Maranhão de Abreu
Prof. Manuel Fernando Cabido Peres Lopes

Examination Committee

Chairperson: Prof. Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. Manuel Fernando Cabido Peres Lopes
Member of the Committee: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

January 2021

Dedicated to my family and friends

Acknowledgments

First, I would like to thank my family, especially my parents. To my mother, for being a pain in my a** during my school years and always pushing me to study more and get better grades. To my father for always being there when I needed to discuss hard Maths problems and for nurturing the geeky side of me. Both of you were always fair to me, although sometimes it was hard for me to understand it.

I also would like to thank my supervisors from IST Prof. Rui Maranhão Abreu and Prof. Manuel Lopes for sharing their insight on the subject and always giving their honest opinion on the work developed. Also, a quick thank you to Daniel Correia for sharing is thesis work.

A word of appreciation to OutSystems for allowing all of this. It was a great opportunity to work with such a great company and I will be forever thankful. The free coffee helped a lot! Thank you to the IA department for being so easygoing and welcoming me and my fellow colleagues into their working space.

I also want to give a special thanks to Eng. João Nadkarni for guiding me through this whole process. From my integration in OutSystems to correcting my thesis poor english sentences. I know this thesis would have not been delivered on time if it wasn't for his consistent wake up calls. Thank you for always answering my, sometimes, ridiculous questions and I'm sorry for bothering you on the weekends and holidays.

And last, thank you to all my friends: Gonçalo Matos whom I had the pleasure to share desk at the OutSystems' office; Catarina Coelho for always letting me borrow her notes; Ricardo Oliveira with whom I shared sleepless nights doing never ending projects. Pedro Dias and Pedro Roque for the amazing Erasmus experience. Beatriz Correia for always saying hello in my Twitch stream. Carolina Santos for making me apply to this thesis project. And finally thanks to Manuel, Diogo, Medeiros, Santos and Borges with whom I once, almost, split the price of a boat engine.

Resumo

Continuous Integration é o processo de juntar as alterações de código dentro de um projeto de *software*. Este mecanismo de manutenção da *branch* mestre de um projeto sempre atualizada e sem falhas, levanta problemas em termos de custos computacionais, considerando a enorme quantidade de código existente em grades sistemas de *software* que necessita de ser testada. Dada esta situação, o trabalho dos trabalhadores torna-se mais difícil, dada a quantidade de tempo que estes têm de esperar pelo *feedback* das suas alterações no código - média de 1 hora na OutSystems.

Reconhecendo este problema num contexto da OutSystems, esta dissertação propõe uma solução, que tenta reduzir o tempo de execução da fase de testes, selecionado apenas uma parte de todos os tests, daao uma determinada mudança no código. Isto é cumprido através do treino de um Classificador de *Machine Learning* com *features* como historico de falhas de ficheiro de código/teste, extensão de códigos de ficheiro e outros.

Os resultados obitdos pelo melhor Classificador de *Machine Learning* treinado mostra resultados muito bons, comparáveis à mais recente literatura na mesma área. Este classificador conseguiu reduzir o tempo mediano de execução de testes por 10 minutes, mantendo 97% de *recall*. Adicionalmente, o impacto de submissões de código inocentes e testes *flaky* é considerado e estudado de forma a perceber o contexto industrial da OutSystems.

Keywords: Continuous Integration, Seleção de Testes, Modelo Classificador, testes flaky, submissões inocentes

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.

Keywords: Continuous Integration, Test Selection, classifier model, flaky tests, innocent commits

Contents

Acknowledgments . . . v
Resumo . . . vii
Abstract . . . ix
List of Tables . . . xiii
List of Figures . . . xiii
Nomenclature . . . xv
Glossary . . . 1

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Topic Overview . . . 1
1.3 OutSystems context . . . 3
1.3.1 Test Selection . . . 3
1.3.2 Flaky Tests . . . 4
1.3.3 Innocent Commits . . . 4
1.4 Objectives . . . 5
1.5 Thesis Outline . . . 5

2 Background **7**
2.1 Test Selection . . . 7
2.2 Supervised Learning . . . 8
2.3 Algorithm Selection . . . 9
2.4 Hyper Parameter Tuning . . . 12
2.5 Unbalanced data sets . . . 13
2.6 Flaky Tests . . . 13

3 Related Work **15**
3.1 Test Suite Selection . . . 15
3.2 Feature Selection . . . 17
3.3 Flaky Tests . . . 18

4 Solution Proposal 19

4.1 Goals . . . 19

4.2 Solution Design . . . 20

4.2.1 Data Set and Features definition . . . 20

4.2.2 Classifier Models' Training and Tuning . . . 23

5 Implementation 25

5.1 Overview . . . 25

5.1.1 Data sets creation . . . 25

5.1.2 Features data extraction . . . 28

5.1.3 Classifier models generation . . . 31

6 Results 35

6.1 General Details . . . 35

6.1.1 Data set . . . 35

6.1.2 Code base . . . 36

6.1.3 Test Suite . . . 36

6.1.4 Software and harware . . . 36

6.2 Evaluation methodology . . . 37

6.2.1 Threshold variation experiment . . . 39

6.2.2 Time limit variation experiment . . . 39

6.3 Experiments . . . 39

6.3.1 Baseline classifier models . . . 39

6.3.2 Balanced classifier models . . . 40

6.3.3 Tuned classifier models . . . 41

7 Conclusions 49

7.1 Discussion . . . 49

7.1.1 Results comparison . . . 50

7.2 Future Work . . . 50

7.2.1 Tool with warning messages . . . 51

7.2.2 Contextual bandits approach . . . 51

7.3 Contributions . . . 52

7.3.1 OutSystems . . . 52

Bibliography 55

A Classifier models metrics values 59

List of Tables

1.1 Example of the application of the SuperSet Strategy 5

4.1 Example of the structure of the data set before adding the features 20

6.1 Example of a classifier model's results 37
6.2 Macro-recall values from the baseline classifier models 40
6.3 Macro-recall values from the balanced classifier models 40
6.4 Set of hyper parameters for each algorithm 41
6.5 Results of the tuned classifier models 42
6.6 Threshold variation results with threshold at 0.5 42
6.7 Metrics values for the time limits 2500 (CorePlatform) and 450 seconds (Developement) . 45
6.8 Time limit variation results - Core Platform - 2100 secs, Development - 188 secs 46
6.9 Time limit variation results - Core Platform - 2200 secs, Development - 167 secs 46
6.10 Time limit variation results - Core Platform - 1473 secs, Development - 51 secs 46
6.11 Time limit variation results - Core Platform - 1955 secs, Development - 36 secs 46

7.1 Results from Diogo's solution tool 50

List of Figures

1.1 From code submission to code correction (OutSystems) 3

2.1 Processes of Supervised Learning 8

5.1 Steps of the solution's implementation 26

5.2 Simplified view of the relevant database tables 26

5.3 Time-series cross validation 34

6.1 Threshold variation for the BRF (no filter data set) classifier model 43

6.2 Threshold variation for the BRF (innocent-filter data set) classifier model 43

6.3 Threshold variation for the LR-B (no filter data set) classifier model 44

6.4 Threshold variation for the BRF-OS (no filter data set) classifier model 44

7.1 Tool implementation example into the OutSystems CI pipeline 52

A.1 Threshold variation results for the Balanced Random Forest-no filter data set classifier model 59

A.2 Threshold variation results for the Balanced Random Forest-innocent filter data set classifier model 60

A.3 Threshold variation results for the Logistic Regression (balanced)-no filter data set classifier model 60

A.4 Threshold variation results for the Logistic Regression(OS)-no filter data set classifier model 61

Nomenclature

ANN Artificial Neural Network

CI Continuous Integration

KNN K-Nearest Neighbour

ML Machine Learning

RTS Regression testing selection

Chapter 1

Introduction

1.1 Motivation

Normally, in a software company, the software complexity is directly proportional to its code base size. As this 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.

1.2 Topic Overview

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). This process of CI is known to have various benefits, since it helps developers catch bugs in the code earlier and companies to release new products/functionalities twice as often as those who do not use CI [1].

As mentioned above, when a developer adds code to the repository, typically, these additions/ /changes need to go through a testing phase. The process of re-running functional and non-functional tests (tests suites) to ensure that these newly modified files do not influence older functionalities and/or under-perform, compared to older software, is described as Regression Testing.

So for a company like Google, which has two billion lines of code over nine million code files, sixteen thousand commit changes per day, when running continuously four million tests, it is crucial to establish workflows that make feasible managing and working productively [2].

To solve the problem of the increased cost in executing entire test suites, a number of different approaches have been studied to maximize the value of the accrued test suite [3]:

- Test Minimization [4] aims to identify and remove redundant tests from the test suits.
- Test Prioritization aims to order test cases to maximize certain objectives such as fault/bug detection.
- Test Selection [5] seeks to identify the test cases based on relevance given a code change.

These approaches all serve the same purpose: they provide an alternative to the execution of all test cases in a test suite, hence, reducing computational costs.

From these 3 approaches, at first sight Test Minimization can be considered the less efficient one, since it only considers the relation between test cases' coverage. On the other hand, Test Selection and Test Prioritization approaches can take into account the relation between test cases and submitted code. The first one selects a set of tests, from all of the tests. The second orders test cases to maximize, for example, the number of tests executed in a time interval.

Since code changes may introduce faults to a repository, it is crucial to look at these changes and their relation to the test case. This is why Test Selection and Test Prioritization can obtain more specific results for each change when applied to a project, thus providing a more significant support in the CI process of a company. Acknowledging the advantages of both approaches, the attention is shifted towards implementation methodologies.

In recent literature, such as [5] and [4] there are examples of Test Selection and Minimization approaches using Machine Learning techniques. The authors use features such as test files historical failure rates and number of contributors to a code file submission. The more relevant these features are, the more precise the classifier will be classifying unseen data. In the context of our solution, the selection of these datasets is very important, since it will aid the classifier model in the decision if a test suite should be executed on a given file recently changed. To identify relevant features to include in the training set, an intensive analysis of the OutSystems processes' of CI and Regression Testing was carried out.

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 that test outcomes are 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 [6]. If there is 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.

1.3 OutSystems context

In the context of Test Selection at OutSystems, this thesis follows the thesis of 2 other students: Diogo Oliveira [7] and Daniel Correia [8]. Diogo proposed solution is based on static and dynamic dependency analysis between code and test files. Daniel presented a solution based on test coverage metrics [8].

The solution proposed by Diogo revealed some concerns regarding the mapping of the tests to the code files, where external call from the tests were hard to track. Daniel's solution revealed several concerns regarding applicability to cross-language code changes (i.e., non C# files) and the scalability bottleneck from coverage data extraction.

1.3.1 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.1.1), 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.

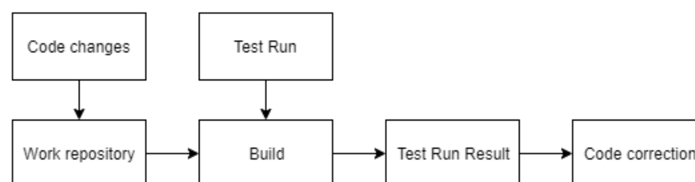


Figure 1.1: From code submission to code correction (OutSystems)

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.

1.3.2 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.

1.3.3 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 [8], 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 iteratively compare the set of failing tests from one commit to the previous one. Table 1.1 shows an application of this strategy.

When not applied any strategy to identify innocent commits, we classify every single commit as guilty. When using the SuperSet strategy, as the figure shows, Commits 3, 5 and 6 can be "set free of guilt". Lets analyze every iteration of the strategy:

[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

Commits	Commit 1	Commit 2	Commit 3	Commit 4	Commit 5	Commit 6
Failing Tests	A	A,B	A,B	A,B,C	C	C
No Strategy	-	Guilty	Guilty	Guilty	Guilty	Guilty
Superset Strat.	-	Guilty	Innocent	Guilty	Innocent	Innocent

Table 1.1: Example of the application of the SuperSet Strategy

- Commit 1 - not classified: it is out first commit, therefore we dont classify it, because we have no terms of comparison;
- Commit 2 - guilty: Test B does not fail in Commit 1;
- Commit 3 - innocent: Test A and B both fail in Commit 2;
- Commit 4 - guilty: Test C does not fail in Commit 3;
- Commit 5 - innocent: Test C fail in Commit 4;
- Commit 6 - innocent: Test C fail in Commit 5;

1.4 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.

In a Test Selection point of view, the solution's aspirations is to select a subset of tests from the test suite, which can detect the same faults (or close) as the current method (re-run all). In a Test Prioritization point of view, the tool should be able to maximize the time saved in the execution of all tests and the correctly identified failing tests.

1.5 Thesis Outline

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

1. **Introduction**: provides an overview of the motivation for this work and a summary of the proposed approach.
2. **Background**: explains the theoretical concepts required to understand the identified research problem and the proposed approach.

3. **Related Work**: provides an overview of the state-of-art regression test selection techniques, feature selection and flaky tests with an emphasis on approaches that use Supervised Learning.

4. **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.

5. **Implementation**: provides an overview of the processes which lead to the construction of the proposed solution.

6. **Results**: provides the results obtained in the proposed solution.

7. **Conclusions**: contains the evaluation methodologies to be used in this work and discussion regarding the performance.

Chapter 2

Background

In this section, there will be an explanation about the main concepts of this thesis and also the context of some of these concepts inside the internal processes at OutSystems.

2.1 Test Selection

Test Suite Selection belongs to a set of approaches that try to solve the problems that derive from the Regression Testing process, by identifying the relevant test cases given a code change. Yo et al. in [3] formally defined the problem of Test Selection as follows:

Input : Program (P); Modified version of P(P'); Test Suite (T).

Objective: Find the subset of T (T'), with which to test P', which contains all the test cases that will reveal faults in P'.

This problem can be transformed into a well-known problem, that belongs to the class of NP-Complete problems: the Set-Cover problem. The Set-Cover problem is defined as follows:

Input : Universe (U={u1,u2,...,un}); Subsets (S={s1,s2,...,sk} $\subseteq$ U).

Objective: Find a set of Subsets (S') that minimizes the number of Subsets selected, such that the Subsets in S' covers the whole Universe U.

By analysing the two problem definitions and the definition for the best subset T', one should be able to reduce the Set-Cover problem to the Test Selection problem by:

- Defining the Universe U as the Test Suite, so U={t1,t2,...tn};
- Defining each Subset S as only one test case, i.e., s1=t1, s2=t2, etc.

Given this, the problem objective would be defined as finding the smallest set of S, such that this set includes every test case that will reveal all faults in P'.

Given the above problem reduction, the Test Selection problem is proved to be NP-Complete, which are not solvable in polynomial time.

In the industrial context, the common base approach to this problem is to do no selection at all and execute all tests in a test suite. This approach is often called "Retest All". However, in recent literature it

was presented some more elaborate approaches to prevent the large amount of time and computation costs that this approach requires in large software repositories.

Gligoric et al. in [9] proposes a regression testing technique that dynamically identifies the test files (subset S) that should be executed given a code change (Modified version of P - P'). By analysing dependencies of tests on files, Ekstazi collects a set of code files that are accessed during the execution of the tests. Then, it detects affected test files by checking if the code files selected changed.

Machalica et al. in [5] proposes a predictive test selection strategy using machine learning techniques. This model tries to predict the outcome of a test execution over some changes (passed or fail), based on historical test outcomes. Ultimately, this model helps selecting a subset of tests to exercise on a particular code change.

2.2 Supervised Learning

Supervised Learning is the process of learning a function that maps input variables to its classes, based on example variable-class pairs. An algorithm analyses these pairs and learns a function (model classifier), which can be used to map a future unseen data point to its unknown class. More specifically, given that OutSystems stores data regarding the outcome (test failed or passed) of old test regression processes, the approach developed during this thesis used this class-labelled data to train a Supervised Learning classifier. Kotsiantis in [10] describes the process of supervised learning which is summarized in Fig.2.1.

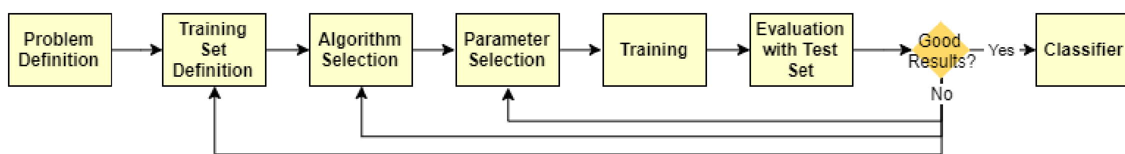


Figure 2.1: Processes of Supervised Learning

The first step in Supervised Learning is to select a data set consisting on a training set and a testing set. This data set needs to aggregate information about data points features and the class to which these data points belong.

Once the Supervised Learning algorithm is selected, the process of training is done by providing the selected training set to the algorithm, which generates a model classifier.

Once trained, the classifier is tested by using it on the different set of data points, the testing set. If the classifier was able to truly capture the relationship between variable features and class, during its training over the training data, it should be able to predict the classes of new unseen data. However, if the results show a significant error rate, we must return to a previous stage of the supervised ML process. Two examples that can lead to unsatisfactory results are the small size of the training set and the low relevance of the features used [10].

In this phase of the Supervised Learning, the procedure it is to perform multiple iterations over multiple algorithms to see which algorithms shows the best results.

However, the algorithms that are used to generate the model classifier have different parameters, each one possibly taking various values. So, to improve the accuracy of these algorithms, an important step is to perform a parameter selection for them, refereed as Hyper Parameter Tuning. The idea behind this process is, to also, iterate over different sets of algorithms' parameters by generating a model classifier and test it with the testing set. By making use of Practical Bayesian Optimizations [11] we can find the model classifier that shows the best results.

One thing important to notice, the goal for two problems can be different, i.e., in one of them the model's classifier objective may be to have a high recall values in deterioration of precision and in the other the model's classifier objective may be to balance out recall and precision. So, during the results evaluation, it is crucial to evaluate properly the results given the objectives of the problem context.

2.3 Algorithm Selection

Supervised Learning can be divided into two types: Regression and Classification. The difference between them is the type of the output. Regression algorithms learn a function that maps input variables to numerical or continuous output variables. Classification algorithms learn a function that maps input variables to discrete or categorical output variables. Given the problem context described in this paper, the following part of this section will focus on classification algorithms.

There are many cases of Classification algorithms used in various fields. Solomatine et al. in [12] compares the performance of these algorithms applied to Hydrologic forecasting. Pereira et al. in [13] describes the various stages in a Machine Learning classifier of functional Magnetic Resonance Imaging (fMRI). Kotsiantis in [10] reviews multiple classification algorithms, comparing their features, such as speed of classification, tolerance to missing values, noise and redundant attributes. Given their insight, next in this section, it will be described some of the algorithms studied: Decision Trees, Artificial Neural Networks and K-Nearest Neighbour.

Decision Tree: Decision tree builds classification models in the form of a tree structure. This tree is composed by nodes and branches. Each node represents a feature in an variable to be classified, and each branch represents a value that the node can assume.

The construction of the tree is done recursively by breaking the data set into smaller and smaller subsets, until all the features are represented by a node in the tree. In each iteration, the algorithm uses some method to find the feature that best divides the data set. There are numerous methods to do this search such as information gain [14], which is based on the decrease in entropy after a data-set is split on an attribute, gini index [15], which estimate each feature independently, and ReliefF algorithm [16] that estimates features in the context of other features.

Decision trees are prone to overfit by creating too many branches. This will most likely lead to a poor performance when faced with unseen data. There are two ways of dealing with this problem: pre-pruning the decision tree by not allowing it to grow to its full size; or post-pruning, which removes branches from the fully grown tree and assigns nodes to the most common class of the training variables.

Artificial Neural Network: An Artificial Neural Network (ANN's) is inspired by the way the biological

nervous system, such as the brain, process information [17]. It is composed of large number of highly interconnected processing units (neurons). These units are usually segregated into three classes: input units, which receive information to be processed; output units, where the results of the processing are found; and units in between known as hidden units.

ANN's can be divided into feed-forward and recurrent classes according to their connectivity. Feed-forward ANN's allow signals to travel one way only, from input to output. Recurrent ANN's allow signals to travel in both ways, by introducing loops in the network.

Considering a feed-forward ANN, with only one layer of hidden units. Each input unit has an activation value that represents some feature from a data set. The way information flows through a ANN is done by propagating this activation values to each of the hidden units to which each input unit is connected. Then, each of these hidden units calculates its own activation value and passes it to the output units.

The activation values for each receiving unit is calculated according to a simple activation function, which takes into account contributions of all sending units. The contribution of a unit is defined as the weight of the connection between the sending and receiving units multiplied by the sending unit's activation value.

The training of a ANN can be conducted by applying different algorithms, which differ from each other in the way they estimate the weight values of the connections between sending and receiving units. Back-Propagation is one of these algorithms and is described by Kotsiantis et al. in [10]. After the calculation of the output units values, given a training set, these predicted values are compared to the actual ones. Based on the difference between the actual value and the predicted value, an error value is computed, given a set cost function, and back-propagated through the network. At each layer the weights are adjusted given the error so that the cost function is minimized. This process is repeated until a defined termination criteria is achieved (number of iterations or achieving a certain loss value for e.g.).

K-Nearest Neighbour: K-Nearest Neighbor is an instance-based classifier which maps all variables in a training set to a n-dimensional space [18]. The dimension of this space depends on the number of features of the variables in the training set. KNN is based on the principle that variables with similar properties will exist close in a dimensional space.

Given a value k, when the algorithm receives an unseen variable, it analyses k number of variables that have the most in common with the new variable, maps it close to them and returns the most common class (of those k variables) as the class of the received variable.

When the algorithm maps a new variable close to similar ones, it uses a distance metric. The idea of this metric is to minimize distance between similarly classified variables, while maximizing the distance between variables of different classes.

Ensemble Learning: However, when the classifier models created by the described algorithms are used, they might not perform so well by themselves. This weak performance is derived from having high bias (underfitting) or too much variance (overfitting) which can be related to the quantity of data or dimensionality of the space. These models are often called weak learners.

To deal with this problem, what is often done is combine multiple models in order to avoid under

and overfitting. By assembling multiple models, we obtain a ”stronger” learner with better performances. This process of combining models by training them to solve the same problem, is referred as Ensemble Learning [19].

For the sake of setting up an ensemble method, two things are needed: select the base models to be aggregated; and select an algorithm to combine the base models selected.

Given the possible disparity in single base models performance, the combination between base models shows great potential. If the chosen base models have low bias but high variance, the combining algorithm should tend to reduce variance. Whereas, if the base models have high bias and low variance, the combining algorithm should tend to reduce bias. Regarding aggregation algorithms, there are 3 kinds: Bagging, Boosting and Stacking.

Bagging [20] makes use of bootstrapping approaches to create samples of the initial dataset (which are considered to be representative and independent of the distribution of the initial dataset) to train several base models in a parallel way. After the training, the final ensemble model results from averaging the results of these base models.

Since Bagging methods utilize bootstrapping, the ensemble models that these methods produce have a lower variance. This is due to the fact that the samples on which the base models are trained are approximately independent and identically distributed, and so the resulted improved models.

One variation of the bagging method is **Random Forests**, which implement deep trees as base models [21]. This variation produces even more robust models because the bootstrap samples are created by sampling over the observation in the dataset (as in normal bagging) and over features, making each tree much more unique. Thus, reducing even more the variance in the final ensemble model.

On the other hand, the idea of Boosting [22] is to train the base models sequentially. Opposite to Bagging (in terms of bootstrap and parallelisation techniques), Boosting takes into account the original dataset and, iteratively, trains the base model, aggregates it in a ensemble model and updates the training dataset. This update serves as a guideline for the training of the next base model, taking into account the strengths and weakness of the current ensemble model. This process of constant adaptation is what allows the boosting algorithms to have a lower bias than the models by which they are composed.

The most popular Boosting algorithms are Adaptive Boosting [23] and Gradient Boosting [24]. These algorithms differ on how they create and aggregate the base models during the sequential process. Adaptive Boosting updates the weights attached to each of the training dataset observations, whereas Gradient Boosting updates the value of these observations. Other example of boosting algorithms is XGBoost [5]

In the literature, ensemble models are proved to highly outperform single models [25]. When comparing Bagging and Boosting methods, Bagging slightly outperforms Boosting, because the second method often overfits and is not able to deal as well with noise [25, 26]. However, authors say that when properly applied, Boosting may be more accurate and achieve better results.

Kotstiantis et al. in [10] describes the process of applying supervised ML to a real-world problem, where they review multiple classification algorithms. The algorithms include Decision Trees, Artificial Neural Networks, Naive Bayes, K-Nearest Neighbors, Support Vector Machines and Rule-learners.

They compare the performance of these algorithms, based on empirical and theoretical studies. The algorithms which scored the best overall accuracy were Support Vector Machines and Neural Networks. Although this overall classification, the authors state that the selection of the best algorithm differs from each application problem. Also, they introduce the concept of combining classifiers as a new direction for the improvement of the performance of individual classification algorithms.

2.4 Hyper Parameter Tuning

Every algorithm used to generate ML classifiers depend on various parameters. When using any algorithm with default parameter values, the results might not be the optimal ones. Hence, finding the optimal parameter set for an algorithm, i.e., the set of parameters that maximize the results for a specific problem, is a crucial step in classifier training. The domain of a hyper parameter can be real-valued, integer valued, binary or categorical. For integer and real-valued hyper parameters, the domains are mostly bounded for practical reasons.

Moreover, the use of certain hyper parameters might be conditioned to the use of others. Consequently, when choosing a set of parameters to use in an classifier algorithm it is necessary to be aware of these nuances.

In order to accomplish the best results in a classification training process, Hyper Parameter Tuning is used to find the best possible set of hyper parameters for a given algorithm. The objective underlined by each Hyper Parameter Tuning technique is to train iteratively a classifier, by selecting the hyper parameters from a pre-defined set, constructed by the user, and returning the values for each hyper parameter with which the classifier showed the best results.

The most common Hyper Parameter Tuning techniques are Grid Search, Random Search and Bayesian Search.

The first one is the most straightforward, where every possible combination of hyper parameters in the pre-defined set are used. Using this technique with small sets may be advantageous, but as the number of values for each hyper parameter increases, so does the computational cost to run all these iterations. And so, time consumption is a huge concern regarding the usage of this technique.

To battle the time cost of the previous technique, Random Search trains the classifier by selecting random hyper parameter combinations from the searchable space (all combinations from the pre-defined set). The underlying problem with this technique is that there is no way to know if the returned set of hyper parameter values are the best combination.

The final technique Bayesian Search complements the previous tecnhique in a way that the selection of the next hyper parameter values are not chosen at random. Baeyasian Search makes use of the Bayes Principle and has 2 essential componets: a probabilistic surrogate model and an acquisition function to decide which point to evaluate next. In each iteration, the surrogate model is fitted to all observations of the target function made so far. Then the acquisition function, which uses the bays Principle , determines the utility of different candidate points [27]. In other words, Bayesian Search considers previous knowledge to make a decision, overcomming the randomness present in Random

Search. Usually, this technique sits in between the 2 previous techniques in terms of execution time.

2.5 Unbalanced data sets

When building data sets which are used for classifier training, one aspect to have in mind is that depending on the problem one is trying to solve, these data sets may be unbalanced. This means that there is a class in the data set that has a majority of examples. For example, in fraud detection problems, the data sets created are expected to be unbalanced given that in real life the non-fraudulent examples heavily dominates the fraudulent ones [28].

The problem that lies with unbalancing data sets is that classifier may become to bias towards the examples from the majority class, by not learning what makes the minority class “different” and fails to understand the underlying patterns that distinguish the classes. Then, the algorithm is prone to overfitting the majority class and it will have the tendency to just predict the majority class. These classifiers will have a high score on their loss-functions, which can lead to the Accuracy Paradox: the finding that accuracy is not a good metric for predictive classifier models.

The solution to this problem lies in balancing the data sets and there 2 major classes of techniques: Over and under sampling techniques.

Over sampling consists in increasing the number of examples from the minority class. Since it is impossible to create minority class examples, the process requires the creation of copies of the minority class so that its cardinality closely matches the majority class. SMOTE, ADASYN and Random oversampling are some examples of this technique.

Contrarily, under sampling consists in decreasing the size of the majority class sample by selecting random samples from this class until it reaches the cardinality of the minority class. One Sided Selection, Neighbourhood Cleaning Rule and Tomek Links are some examples of this technique.

2.6 Flaky Tests

In a perfect world, the outcome of a test execution should not differ given the same code change. However some tests may behave in a non-deterministic way, which can hinder the work of a developer. These tests are called flaky tests.

When a developer is notified about some code file changes that failed in regression testing phase, s/he needs to rectify the code submitted. One of the problems that arise with flaky tests, is that developers may waste time correcting a code change that had no errors, where the failed test that ”flagged” said code file was flaky.

Besides the problem mentioned above, Luo et al. in [6] mentions two more problems associated with flaky tests:

- Flaky tests may also hide real failures, in case the developer thinks one failure is the result of a flaky test, thus ignoring real bugs.

- Test failures caused by flaky tests can be hard to reproduce due to their non-determinism.

Flaky tests may be derived from various sources, such as execution of asynchronous calls, thread concurrency or test order dependencies. There are different techniques to deal with each of these causes, such as implementation of waitFor calls, addition of locks and merging tests with high level of dependencies, respectively.

When faced with flaky tests, a company can do 2 things: fix these tests or learn to live with them. The flaky tests may be fixed, thus removing the flakiness of the regression testing process. However this solution can sometimes go only to a certain point. If by removing the flakiness, developers are jeopardizing future tests by not testing certain functionalities, then the process of testing loses its purpose.

And so, companies can adapt to the existence of flaky tests by identifying these tests, prior to test execution, and mark them as flaky. This way, when developers are correcting their code, they can give priority to the tests other than those marked as flaky, and focus first in these tests.

So that, when developers are notified about some faulty changes, if the test/s that failed were marked as flaky, they can give priority to tests other focus on other tests they can assume that said faulty change is not guilty for the failing of the tests.

The most common approach to find flaky tests is to re-run tests multiple times against the same code. If in these re-run process, the tests show incoherence, i.e., they fail and pass at different times, they are marked as flaky.

Google [29] uses this method, where if a test fails on some code, the test is re-run 10 times. If it passes on any of these times, the test is considered flaky. Machalica et al. in [5] also adopted this flaky tests identification method in their testing selection approach.

Chapter 3

Related Work

In the next section it will be presented an overview of the state-of-art techniques related to the most important subjects of this paper: Test Suite Selection, Feature Selection and Flaky Tests.

3.1 Test Suite Selection

Rothermel et al. in [30] 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 [31] 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.

In order to complement this work, Daniel Correia in [8] proposes a test selection tool which pairs a test suite diagnosability metric, called DDU[1] with historical metrics of test files. In his results, given the multiple challenges presented such as multiple language code and the scalability bottleneck from coverage data extraction, Daniel's tool was still able to reduce the feedback time by making reasonable selections given the size of the code changes.

[1] DDU is an acronym for Density-Diversity-Uniqueness

Gligoric et al. in [9] proposes a RTS technique called Ekstazi. This technique tracks dynamic file dependencies of tests on files, which does not require integration with version-control systems. Ekstazi, while keeping track of the dependent code files for each test file (dependency files), performs regression testing in 3 phases: (1) In each revision, for each test file, Ekstazi checks if the checksums of the dependent code files (in the dependency files) are still the same. If so, the test is not selected for execution. (2) Ekstazi runs the test files selected in the previous phase. (3) Ekstazi monitors the execution of the tests and the code under test to collect the set of code files accessed during execution of each test, computes the checksum for these files and saves them in the corresponding dependency file. Ekstazi uses file-level granularity to detect test dependencies and code changes, which showed better results than techniques with finer granularity (class and method level).

Followed by their work, Legunsen et al. in [32] conducted an extensive study of static techniques. The authors implemented two static regression testing techniques, one class-level and one method-level, and compared several variants of these techniques. Comparing their techniques with Ekstazi, the authors found that the class-level technique showed similar performance benefits, while the method-level performed poorly.

Machalica et al. in [5] proposes a different predictive test selection strategy using machine learning 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 [4] 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.

The authors results show that their techniques can save a fifth of test-time while obtaining a test outcome accuracy of 99.99%.

Inball et al. in [33] present a novel approach, called cOmponent Sensitive Cross project software fAult pRediction model (OSCAR), for the cold-start problem in software fault prediction. This problem derives from the lack of historical data about a project that is new. The authors create a fault prediction model (belongingness classification model) for new software projects with no recorded history, by mapping a software component to the most similar project among a set of old/recurring projects. Then, for each one of the components of the new project, they predict whether it is faulty or not using the prediction model of the most similar project. In their results, the authors compared OSCAR against existing state-of-the art algorithms of cross-projects software fault prediction, where they achieved the highest accuracy amongst all algorithms. They also highlight the importance of the belongingness classification, which greatly affects the accuracy of OSCAR.

3.2 Feature Selection

Memon et al. in [34] 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 [4] present FastLane, a system that performs data driven test minimization. Particularly, in their system, which is divided in three approaches, they resort to machine learning models to learn a classifier that labels a commit as risky or safe. A commit is risky if it causes at least one test to fail. A commit is safe if all tests run on it pass. To train the classifier, they 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 [5] in their predictive Test Selection approach train a machine learning 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.
- Cross-Features: distance (between test files and code files) in build dependency graph and lexical distance between file paths (test and code files).

3.3 Flaky Tests

Machalica et al. in [5] 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 [35] 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 runtime overhead. DeFlaker operates in three stages: (1) combining syntactic change information from Git, with structural information from each program source file, DeFlaker identifies the program locations that changed; (2) DeFlaker generates a coverage report that lists each changed line/class covered by each test. (3) In the final stage, 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.

[2] Components with high risk of failure generation

Chapter 4

Solution Proposal

In this chapter, we present the solution proposal to solve the problem that arises from the current regression testing approach at OutSystems. Along side with the goals for this solution proposal, we will also present the concerns to potentially integrate it into the current OutSystems' development processes. Finally, the steps to achieve this solution will be described in the Solution Design chapter.

4.1 Goals

In order to help a developer maintain a more consistent work without having to wait this amount of time, our goal is to select a sub set of tests, out of the entire set of tests, for the developers to run in their local machines. This sub set of tests should reveal all faults given the developer's most recent changes.

Currently, a developer at OutSystems has to wait, on average, 1 hour before receiving information about which tests failed. Therefore, in order to include the proposed solution into the current OutSystems' development workflow, it needs to fulfill the following performance concerns:

- Execution time - It should execute fast enough so that it can be integrated into the existing development workflow without significant overhead.
- Feedback loop time - The time to get feedback on changes should be much lower than the current system (1 hour).

Something worth pinpointing is that the purpose of the solution is not to replace the current regression testing process at OutSystems. Therefore, the use of the tool would be in a pre-commit stage. Meaning that a developer finishes his/her work on the code base, then s/he runs the solution. It then returns the sub set of tests that fail, given the modified code, for the developer to run on their local machines. After observing the outcome of that sub set of tests (fail or pass) over the newly modified code, the developer rectifies the code, if needed, and only then s/he proceeds to make a commit to the OutSystems code repository. After the commit, the normal process of regression testing continues, where the entire set of tests are run over the newly committed code.

4.2 Solution Design

The solution presented in this thesis focus on training a Machine Learning (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.

4.2.1 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).

Table 4.1 shows an example of the data set (before the inclusion of the features) which will be used.

Author	Changelog	TestRunId	Test	StageName	Failing
Author1	CodeFile1	100	Test1_CP	Core_Platform	0
Author1	CodeFile1	100	Test2_CP	Core_Platform	0
Author1	CodeFile1	100	Test3_CP	Core_Platform	1
Author1	CodeFile1	101	Test1_Dev	Development	0
Author1	CodeFile1	101	Test2_Dev	Development	0
Author1	CodeFile1	101	Test3_Dev	Development	0
Author2, Author3	CodeFile2, CodeFile3	110	Test1_CP	Core_Platform	0
Author2, Author3	CodeFile2, CodeFile3	110	Test2_CP	Core_Platform	0
Author2, Author3	CodeFile2, CodeFile3	110	Test3_CP	Core_Platform	1
Author2, Author3	CodeFile2, CodeFile3	111	Test1_Dev	Development	1
Author2, Author3	CodeFile2, CodeFile3	111	Test2_Dev	Development	1
Author2, Author3	CodeFile2, CodeFile3	111	Test3_Dev	Development	0

Table 4.1: Example of the structure of the data set before adding the features

The work done in [36], 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 Machine Learning. 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 describes 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. The statistics showed that tests with a large number of executions have a lower failure rate. This means that there are tests that are more prone to failing than others, thus the inclusion of this feature in the data sets.
- **Author failure rate** - For this feature, we need to introduce a conceptual idea: The author (developer) guilt for one testrun is binary: if there are failing tests in the testrun the author is given the guilt. If no tests fail, then the author is given no guilt.[1] So, 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 s/he. Statistics showed that authors who have a smaller amount of commits tend to have a higher

[1] For easier description, from now on, testruns with at least one failing test will be refered as failling testruns.

percentage of failed commits and authors with more number of commits have a lower percentage of failed ones. One plausible explanation can be related to the authors' experience. Authors with more commits are more likely to work at OutSystems for a longer time, thus having more experience, than authors who have less commits and possible being employees for a smaller time.

- **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. Much as the previous feature, where an author can get the blame for a failed testrun by only one test failing, the same happens for files. In a testrun, if a single test fails, all of the code files linked to that testrun are blamed. Hence, keep in mind that this statistic can be biased. For example, a testrun linked to 5 code files: even if only one of them is responsible for the failure of a test, all of the code files will be assigned as "guilty", then changing all failure rates.

- **File/test failure rate** - This feature is a combination of the File and Test failure rates. Test files and code files are connected by obvious reasons: test files are basically code files that test the functionality of code files. Lets take a software project as an example: we can think of the code files as functionality nodes. And for each functionality there is a test file that assures that functionality keeps working. Therefore, we can link code files to test files based on the functionality the tests evaluate. Basically, 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** - Similarly to the previous feature, 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. We can also find a relation between the authors and the code files submitted by them. In software companies, teams of developers are in charge of different parts of the company product. Thus, certain teams are more used to work with certain code files than others. This creates a pattern in these rate values, because each team will eventually have low failure rates for the files it is more used to work with.

- **Extension file type** - The statistics showed that code files with different file extensions have different failure rates. More specifically, C# files lead to more failed tests. So if the machine learning classifier model takes into account the extension of the file, it could help in the task of deciding if a test should be run over a code file with a given extension.

- **Tokens shared file/test** - As seen before tests and code files are related. This feature takes into account this relation but not from any failure rate. It compares the name's test with a code file's name. In software testing, it is instinctive to name the test after the code file or the functionality it is testing. At OutSystems, tests have the characters "." separating its name. Code files on the other hand have the character "/" delimiting the various directories from which it belongs. So, to capture possible relations between both, we split test and code files by these characters and compare the sub-strings(tokens) of both to see how much they are related. The more tokens they share, the more related they should be.

- **Number of distinct files changed** - This feature simply determines the number of code files submitted by an author in the commit stage. The statistics showed a counter intuitive pattern: as the size of the author's commit increases the failure of those commits decreases. It would be expected that the more files an author submits the more probability there is to generate a failing test. However the opposite happens, which can be explained by the fact that developers might be more meticulous when committing a large amount of code files.

- **Number of distinct authors** - One important detail, which will be explained in detail in the next chapter is that, during the regression testing process at OutSystems, multiple commits can be aggregated into the the same testrun. For this reason, when the tests run over newly submitted code, it may happen that the code which is being tested is the product of the work of multiple authors. For this reason, one feature that can be helpful in predicting the outcome of a test is the number of authors linked to one testrun.

- **File change history** - This feature represents the frequency which a code file is submitted by authors. Normally, there are 2 scenarios to when a code file is submitted multiple times in short period of times: there is new functionalities consistently being added through that code file; or there is something wrong with it and it is generating unwanted failing tests. The statistics showed that the percentage of commits which generate test failures increases with code files that are more times submitted. To turn this feature more diversified, we pre-define 3 time intervals, previous to the date of a commit and calculate 3 different values for this feature for each code file. This way, we reduce the bias towards code files with a high change history from a long time ago (for example 2 months), when in the last 10 days was never submitted.

- **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. This way, tests that have a high failure rate from a long period in the past, do not get falsely blame when show a lower failure rate in a near past. By creating the different time-intervals, we generate 3 different features depending on each one.

- **File failure rate history** - This feature presents the same purpose of the code file failure rate feature, but similarly to the previous feature we try to reduce bias towards code files with high failure rate from a long period in the past, and pre-define 3 time intervals. The same way as the previous feature, by creating the different time-intervals, we generate 3 different features depending on each one.

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 [5], 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, explained in section 1.3.3, the innocent commits are excluded from the No-filter data set and the prediction accuracy of the classifier is analyzed.

4.2.2 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.

Chapter 5

Implementation

In this chapter we present 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 ML classifier is a function which maps input variables to discrete output variables. In this context, the optimal classifier model is the one which selects the smallest set of tests which reveal all faults for a given code change.

5.1 Overview

The proposed solution was designed in the context of OutSystems' code base and CI pipeline and is organized into three main groups:

1. **Data sets creation**: process of data extraction from the OutSystems database regarding the developers commits. As mentioned in the previous chapter, there are 2 additional data sets created for this solution (innocent/flaky filter data sets). The process of the creation of these 2 data sets is also explained.

2. **Features data extraction**: process of gathering historical data about commits, developers, code and test files, calculation and appending of features values to the existing data sets.

3. **Classifier models generation**: process of creation of the classifier models' baseline and hyper parameter tuning of those classifier models.

Fig. 5.1 resumes the procedures which lead to the solution conception.

5.1.1 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

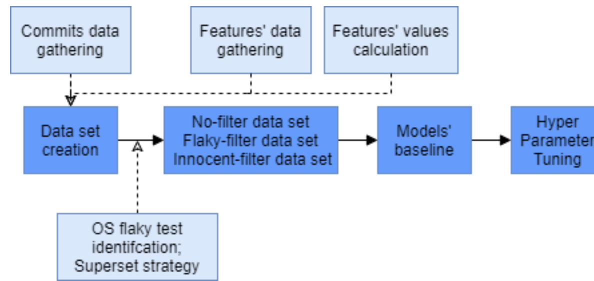


Figure 5.1: Steps of the solution's implementation

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. 5.2.

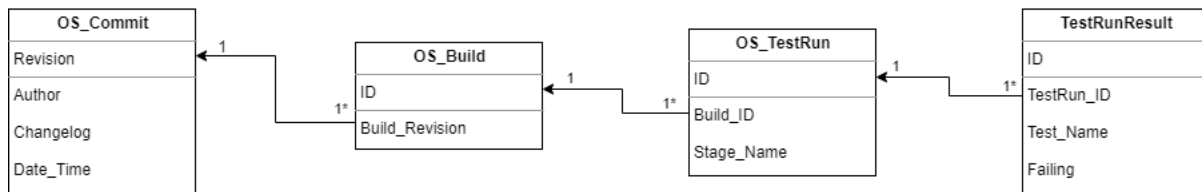


Figure 5.2: Simplified view of the relevant database tables

- **OS_Commit** - contains information about a commit done by some author. The "Changelog" parameter contains the code files submitted in each commit.
- **OS_Build** - contains information about commits that were assigned to a given build process.
- **OS_TestRun** - contains the information about a test suite executed over a given Build.
- **OS_TestRunResult** - contains, in each row, the test files which were executed in a given TestRun and their outcome (Failing - pass or failed).

To build our data sets we need to merge the information from all tables based on the common parameters between them. Fig. 5.2 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.

The first step is to retrieve the list of commits done during the desired period. The following SQL query was used to retrieve commits data from the **OS_Commit** table, where we use the parameter "Date_Time" to filter the query's result, so it matches the timeline defined.

```
SELECT Author, Changelog, Revision, Date_Time
FROM OS_Commit
WHERE Date_Time < date_lower_limit
    and Date_Time > date_upper_limit
ORDER BY Date_Time;
```

The second step is to retrieve data from the **OS_Build** table. To do this, we need to analyze the results from the query performed on the first table and check the lower and upper bounds for the ”Revision” values. Given that this second table does not store any information about the date of the Build, we need to filter the result based on those upper and lower bound values from the previous query results. The SQL query used to retrieve builds data from the **OS_Build** table is presented next.

```
SELECT ID, Build_Revision
FROM OS_Build
WHERE Build_Revision < revision_lower_limit
    and Build_Revision > revision_upper_limit
```

The third and fourth step consist on retrieving data from the tables **OS_TestRun** and **OS_TestRun Result**. The same way we used ”Revision_ID” to filter the results from the query on the second table to obtain the data we need from **OS_Build**, we will use ”Build_ID” and ”TestRun_ID” to filter the results of the queries on the **OS_TestRun** and **OS_TestRunResult** tables, respectively. The SQL queries used to retrieve data from those tables are presented next, respectively.

```
SELECT ID, Build_ID, Stage_Name
FROM OS_TestRun
WHERE Build_ID < build_lower_limit
    and Build_ID > build_upper_limit

SELECT ID, TestRun_ID, Test_Name, Failing
FROM OS_TestRunResult
WHERE TestRun_ID < testrun_lower_limit
    and TestRun_ID > testrun_upper_limit
```

After collecting the data from this tables, the next step is to merge the information from all tables. Since all tables have parameters in common, we can merge them by these parameters.

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.

In the previous chapter, we introduced the strategy SuperSet to find the innocent testruns[1] in our original data set. During this process, one thing to remind is that each testrun corresponds to one stage of tests. As mentioned before, we are only including tests from stages Development and CorePlatform in our data sets and these stages are run sequentially for every new set of changelogs. Therefore, when calculating the innocent testruns we can only compare testruns which belong to the same stage, because the tests from each stage are unique from that stage.

Given the explanation above, the process of selecting the innocent testruns is performed separated for each stage and will be explained for only one stage, once the processes are homologous. First we need to aggregate the tests that fail in each testrun (lets say in a Python list, where each element contains the failing tests for each testrun). Then, we iterate over this list and compare each element with the previous one and set a condition: ”if the previous set of failing tests is a super set of the current's one”, then we save the current testrun. After this cycle, we end up set of saved testruns (the innocent testruns). The final step is to filter the original data set, where we iterate over the same and maintain only the testruns that do not show in the set of innocent testruns saved in the previous step (from both stages). Therefore we end up with a data set which only contains the guilty testruns, thus the innocent-filter data set.

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. Given the time period of which we built our original data set, we only need to retrieve the flaky tests from March 1st 2020 to April 8th 2020.

The result from the query to the OutSystems' database (lets call it flaky table) is a table with the tests marked as flaky for every day in the time interval referred above. The next step is to iterate over the original data set and for each element check if the test name and the day of the datetime match any entry in the flaky table. If so, we remove that entry from the original data set. In the end of this iteration, we end up with a data set free of flaky tests, thus the flaky-filter data set.

5.1.2 Features data extraction

The features that our data sets will include were described in 4.2.1. In this section the process of the calculation of all these features will be explained. Also, this process is preceded by some data extraction, which will also be explained.

The majority of the features selected to be part of the data sets are dynamic, i.e., they can be constantly updated such as Test and Author Failure Rates, given that over time they change. Unlike features such as the number of distinct authors which is static, where this simply requires counting the size of the Author column in our data sets table.

[1] As we explained earlier in this chapter, testruns may aggregate more than one commit. For this reason, from now on, we will use the term innocent/guilty testrun, instead of innocent/guilty commit

In order not to have these features' values starting from zero, we calculate baseline values for these features based on previous time intervals from the testruns included in the data sets. Similar to what was done to build the data sets, for this process we basically build an identical data set from previous time intervals (baseline data set).

After building this baseline data set, then we start to iterate over the data sets and calculating/ updating the features values given the outcome of the test in each entry. One important detail during the process of calculation and updating dynamic features values is that we need to assign to each entry values that only take into account previous testruns, excluding the test outcome of the current entry. Thus, in each iteration we calculate the value for each feature and only then we update its value given the test outcome of the test of that iteration.

The procedure for every feature calculation will be described next in more detail. Also, the procedure is exactly the same for the different data sets, whether if we want to get the features for one specific data set, we just iterate over that same data set.

Test Failure Rate: For this feature, we save 2 values during the baseline values calculation for each test: Numbers of total testruns(1) and failed testruns(2). In each iteration over our data sets, there is only one name in the "Test" column, hence we assign that test's failure rate based on the values saved previously. So basically, in each iteration we assign the feature value for that entry by dividing (2) by (1). Only after, we update the total and failed testruns values: incrementing the number of total testruns by one, and if the value of the "Failing" column is 1, then also incrementing the number of failed testruns.

Max Author Failure Rate: For this feature, we save 2 values during the baseline values calculation for each author: Numbers of total testruns(3) and failed testruns(4). In our data set, in the same testruns, the names of the authors repeat themselves (table 4.1). Therefore, during the iteration over the data set we need to assure that we only increment one time the number of total testruns for the same testrun. Similiarly, when we "find" a failing test, we also need to assure that the number of failed testruns are increment only this time and prevent the increment in the next entries for the same testrun. For this we have two flags: *flagTestrun* "stops" to keep the increment of the number of total testruns over the same testrun; and *flagFail* "stops" the increment of the number of failed testruns in case of multiple failing tests in the same testrun.

Other important detail is that the column "Author" can have more than one author, for reasons already explained. Thus, when updating the features values, this process is done to all authors that this column contains. Also, when assigning the feature value for each entry we choose the highest failure rate value between all authors in the "Author" column.

Summing up, in the first entry of a testrun, we assign the feature value for that entry by diving (4) by (3) previously saved. To notice that if the values of total and failed testruns are incremented, the authors failure rate will only change the next testruns, because the feature value that is assigned to each entry is always the same from the first entry (in the same testrun). [2]

Max File Failure Rate: This feature's process is very similar to the previous one. We save 2 values during the baseline values calculation for each code file: Numbers of total testruns(5) and failed

[2] The reason for this is that, in the regression testing process, the execution of the tests in one testrun are supposed to be "at the same time". And so, the feature values only change from one testrun to another, and not during the same.

testruns(6). For the same reasons we have the same flags to prevent wrong updates for the numbers of total testruns and failed testruns.

Similarly, the ”Changelog” column can contain more than one code file. Thus, when updating the numbers of total testruns and failed testruns, this is done to all code files present in the column. And the file failure rate chosen to assign to each entry is the highest file failure rate value out of all code files present in the column. We calculate each file’s failure rate by dividing (6) by (5) from the same code file. To notice that, like the previous feature the feature value that is assigned to each entry is always the same from the first entry (in the same testrun).

Max File/Test Failure Rate: For this feature, we save 2 values during the baseline values calculation for each pair file/test: Numbers of total testruns(7) and failed testruns(8). Trivially, given that the ”Changelog” column may contain more than one file, in each iteration, we assign the feature value to the pair with highest file/test failure rate. And similarly to other previous features we increment the number of total testruns for each pair file/tests for all files in the ”Changelog”. And in case the test fails, the number of failed testruns for all file/test pairs is incremented. Notice that for one entry the test is always the same, where we need only to variate the name of the file to find the maximum pair value. The file/test failure rates are obtain by dividing (8) by (7) for each file/test pair.

Max Author/File Failure Rate: This feature is very similar to the Max File and Max Author failure rate features. Again, we save 2 values during the baseline values calculation for each pair author/file: Numbers of total testruns(9) and failed testruns(10). We need to use those previously mentioned flags to control wrong updates given the repetition of the column ”Author” and ”File” in the same testrun.

In short, for every first entry in a testrun we assign the feature value to the highest failure rate for every pair of author/file, saved previously. These values are obtained by dividing (10) by (9) for each pair author/file. Similarly, the feature value that is assigned to each entry is always the same from the first entry (in the same testrun).

Extension file type: This feature and the next 3 are fairly simply to calculate. To identify the extensions of the files that the ”Changelog” column contains we create 3 columns that are set to 0 or 1, depending on the file extensions of those files. We take into account 3 types of file extensions: C# files(1st column), typescript files(2nd column) and other files(3rd column). For example, if we have a combination of 1, 0, 1 from the columns it means there is at least one C# file, 0 typescript files and at least one other file (other than C# and typescript).

Max Tokens shared file/test: In this feature we need to compare the strings between the files and the tests. Files’ names have ”/”’s that separate the different directories they belong to and tests’ names have ”.”’s that separate the different functions. Therefore, we split the strings by these characters and then compare the number of equal unique substrings between the file and the test. Once again, given that there can be more than one file in the ”Changelog” column we select the maximum number of tokens (substrings) shared between one file and one test. Important to remember that this feature is normalized.

Number of distinct files changed: In this feature we simply count the number of different files included in the ”Changelog” column. The values from this feature are also normalized. This normaliza-

tion [3] converts all feature values to a value between 0 and 1, which is done by dividing all values my the maximum feature value.

Number of distinct authors: In this feature we simply count the number of different authors included in the ”Author” column. The values from this feature are also normalized.

For the last 3 features we need to create a data set which corresponds to the aggregation of the baseline data set and the current data set. This data set (historic data set) will include testruns from January to March.

Test failure rate history: This feature is basically the same as Test Failure Rate. We iterate over our current data set, but in addition we define 3 dates based on the ”DateTime” column from our current data set: The first date is 3 days before the date; the second is 14 days before the date; and the last one 56 days before the date. Then, we filter the historical data set with those dates. The result are 3 portions of the historical data set where we have the entries with testruns from 3, 14 and 56 days before the date in the current iteration (the date not inclusive). After this, based on the name of the ”Test” column, we calculate the 3 different failure rates for that test in the 3 different time intervals and assign them to the features’ column of that entry.

File failure rate history: In this feature, the procedure is the same, with the only difference that we now calculate 3 failure rates for all the files in the ”Changelog” column in the 3 different time intervals. Therefore, the value assigned to the features’ column are the maximum failure rates values for each time interval for all files. So, it may happen the failure rate values may correspond to different files in each time interval. And also, given the repetition of the ”Changelog” column in the same testrun. These values are all calculated in the first entry of a testrun, and in the next they are simply assigned to the features’ columns until the the last entry of that testrun.

File change history: Similarly to the previous features, we filter the historical data set based on the same 3 time intervals. However, in this feature we calculate the number of times a file is submitted, i.e., the time it is included in the ”Changelog” column. Also, instead of choosing the file which was times submitted, the features’ values correspond to the sum of number of times all files were submitted in each time interval. Like the previous feature, these 3 values are also calculated in the first entry of a testrun, and in the next they are simply assigned to the features’ columns until the the last entry of that testrun.

5.1.3 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 is used to evaluate the classifiers’ prediction accuracy.

In the process of splitting the data sets into these 2 parts, we need to take into account that for this specific problem, there is a timeline. This means that the classifier needs to capture the relations between files, authors and tests with an order, the testruns order. Hence, the argument ”shuffle” for the

[3] Altough not necessary for the models we use, since it has a bigger impact with Neural Network models, we decided to normalize all features anyways

"train_test_split" function needs to be set to "False".

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 [4]:

- 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.

Therefore, 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

[4] These algorithms belong to the scikit-learn toolbox - `https://scikit-learn.org/stable/`

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 over-fitting. 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. Fig. 5.3 better illustrates the different folds for the iterations of training and testing during the tuning process.

After the Hyper Parameter Tuning, for each classifier models selected from the previous step, we get a set of hyper parameters. Then, the next step is to train each model with the respective set of hyper parameters, using the original testing set. The results of each classifier model are then analysed and will be showed in chapter 6.

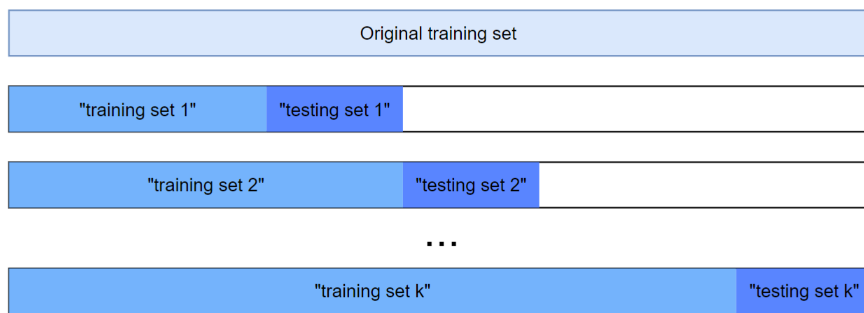


Figure 5.3: Time-series cross validation

Chapter 6

Results

This chapter mainly focus on presenting the results by the model classifiers trained during the solution tool construction pipeline. Hence, 6.3 is divided in 4 groups: Baseline Models' Results, Balanced Models' Results and Hyper Parameter Tuning Results. This chapter also includes the evaluation process and guidelines used to choose the classifiers which to move forward into future steps of the solution tool creation and the classifier chosen to "represent" the solution tool.

6.1 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.

6.1.1 Data set

Regarding our experimental data sets [1], we decided to include testruns starting from March 1st 2020 to April 8th 2020. As mentioned before in 4.2.1, we created the baseline data set to more accurately define the dynamic features' values. This baseline data set needed to contain data from previous testruns. Hence, we decided to include in this one, testruns from January 1st 2020 to February 28th 2020.

Therefore, during the data retrieval from the OutSystems database, we filtered the queries on the **OS_Commit** table based on the "Date_time" parameter.

In order to generate the classifier models, we need to split our data sets into training and testing sets. The model classifier algorithm's uses the training set to generate its classifier model. On the other hand, the testing set is used to evaluate the classifier model's prediction capabilities. When performing this split on the data sets, 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.

[1] no-filter, flaky-filter and innocent-filter data sets

6.1.2 Code base

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. Hence, in all the files submitted by developers in commits to the code repository, there are some not related with the Service Studio project. And since there is no way to identify a priori which testruns are or not related to this project, we can only identify them by checking the ”Changelog” parameter. The files related to Service Studio application start with the ”ServiceStudio” substring.

However, another nuance related to the files is their file extension. Service Studio is a visual tool, and so there are various non-product functionality files (for example .png files) inside the project. Again, we are focused only on functionality of the product, thus only files with file extensions of .cs and .ts/.tsx have to be taken into account.

Therefore, given that testruns may aggregate various files, all of testruns in our data sets have at least one Service Studio code file with one of the file extensions above.

6.1.3 Test Suite

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. Upon, talking with the OutSystems' developers with the knowledge about which stages of tests target the Service Studio application, we included tests from the stages Development and CorePlatform. The Development stage contains 5910 test files and the CorePlatform stages contains 6018 test files. The median execution time of the stages are 2500 and 450 seconds for CorePlatform and Development, respectively.

Also, given that we wish to compare results with Daniel Correia's work [8], the choice to include only these stages of tests like Daniel did, the comparison between both works results is fairer.

6.1.4 Software and harware

The work performed in this thesis from the data extraction to the building of the data sets was made using the Python programming language and the most important python libraries used were pandas and sklearn.

However, the process of training and hyper parameter tuning of all the model classifiers showed to be a very computational heavy process. Therefore, all of the processes related to the training of model classifiers were conducted on a single machine with 16GB of RAM and a 4-core Intel i7 CPU running at 2.60GHz.

6.2 Evaluation methodology

The training of the baseline and balanced models were performed to assess the most promising model classifiers. Therefore, the evaluation of the results from these model classifiers take only into account the values of recall from all the model classifiers. Fig.6.1 shows an example of a confusion matrix and a classification report returned after the training and testing phase of a model classifier.

		Prediction	
		Negative	Positive
Real	Negative	879560	180356
	Positive	7862	83578

	prediction	recall	f1-score	support
0	0.99	0.83	0.90	1059916
1	0.32	0.91	0.47	91440
accuracy			0.84	1151356
macro avg	0.65	0.87	0.69	1151356
weight avg	0.94	0.84	0.87	1151356

Table 6.1: Example of a classifier model's results

When evaluating the values showed, 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 (6.1).

$$Recall = \frac{\text{\#failing tests selected}}{\text{\# failing tests}} \tag{6.1}$$

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 for each pair of algorithm-data set, 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 (6.2) 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 (6.2).

$$Micro - Recall(n) = \frac{\text{\#failing tests selected in TR(n)}}{\text{\# failing tests in TR(n)}} \tag{6.2}$$

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.

During the testing phase of the classifiers over the testing set, the class of each entry is predicted. More specifically, the classifiers assigns a probability to each entry and based on a pre-defined threshold[2] the class of the entries are 0 or 1: if the probability is equal or higher than the threshold, the class is predicted as 1. Otherwise, is predicted as 0.

To calculate the metric values we use the probability predicted by the classifier for each entry, instead of the actual class (0 or 1). Therefore, we need to concatenate these probabilities predictions to each entry testing set. Remembering that, for each entry in our data sets we have a testrun value, hence, when doing this concatenation, we can identify the tests' outcome in each testrun. After this process, we iterate over this new data sets to calculate the metrics mentioned above. Since each classifier may have different predictions, this process is repeated for all classifiers generated.

The reason why we use the probability prediction instead of the class value is explained in the next subsections.

[2] most common default threshold is 0.5

6.2.1 Threshold variation experiment

One experiment decided to implement in the evaluation of the classifier models is the variation of the threshold value for each one. Thus, the reason for the use of the probability value. By using the probability value, we can get the metrics values for different threshold by just vary its value, without the need to explicitly write the code for the classifier model prediction with different threshold 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.

6.2.2 Time limit variation experiment

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. To achieve this, for each testrun in the prediction data sets, the probabilities are sorted, starting by the highest probabilities. Also, for each entry in all of the prediction data sets, we concatenate the median time for the test in that entry (calculated from previous testruns). After this, we concatenate another column with the cumulative sum of these times through out the prediction data set. These last column is what allows the developers to set a time limit for the tests' execution, which is basically an iteration over the prediction data set that stop whenever the cumulative sum value reaches the time limit value defined by the developer.

For the last variation of results all of the metrics previously mentioned are calculated, with exception for the median test execution time, given that it is implicit in each time limit variation.

Also important to mentioned is that, we need to set different time limits for different stages since they have different execution times. Therefore, to set up our baseline limit for both stages, we need to calculate the median for test execution time for both stages.

Summing up, we run multiple versions of these 2 variations of evaluations over all classifiers' predictions, where we iterate over different values of thresholds and different time limit values. Then, we check which classifier has the best results with which threshold and time limit values. The best model classifier will be used in the solution tool.

6.3 Experiments

6.3.1 Baseline classifier models

The results from the baseline classifier models are shown in Fig.6.2.[3] During the training of these models, all of the functions' algorithms which were used were empty, i.e., no parameters were set. Thus, all the default values for each algorithm were used.

The figure shows a clearly advantage from Balanced Random Forest which, as the name says,

[3] All of these results are from using only the no-filter data set.

Classifier model	Macro-recall (%)	Training and testing time (s)
K-NN	0.09%	5 h
Logistic Regression	0.00%	2 min
Random Forest	0.00%	8 min
Balanced Random Forest	0.91%	3 min
XG Boost	0.00%	4 min

Table 6.2: Macro-recall values from the baseline classifier models

already implements balancing techniques. This complements the idea that our data set is indeed unbalanced.

6.3.2 Balanced classifier models

Given the unbalance in our data sets which the results obtained by the baseline models , we apply balancing techniques to our data sets. We used over and under sampling techniques. We also made use of the "class_weight" parameter from the algorithms and set it to the "balanced" value. Important to notice that regarding over and under sampling techniques, only the testing set is sampled. The testing set remains untouched.

Fig 6.3 shows the results from the classifier models fitted for balancing.

Classifier model	Macro-recall (%)	Training and testing time (s)
Logistic Regression (over sampling)	92%	7 min
Logistic Regression (under sampling)	0%	4 min
Logistic Regression ("balanced")	92%	5 min
Random Forest (over sampling)	9%	5 min
Random Forest (under sampling)	0%	10 min
Random Forest ("balanced")	0%	4 min
XG Boost (over sampling)	25%	14 min
XG Boost (under sampling)	6%	16 min
XG Boost ("balanced")	6%	5 min

Table 6.3: Macro-recall values from the balanced classifier models

The K-NN algorithm was left out, given the large amount of time it took to get the prediction results. The Balanced Random Forest is also not included for obvious reasons.

By analysing the macro-recall values of the classifier models, we can clearly see that the usage of over sampling and of the "class_weight" parameter boost up the baseline recall values. Specifically, the Random Forest with oversampling, the Logistic Regression with over sampling and with the "balanced" parameter show the most promising results with macro-recall values over the 0.90 percentage.

Given, these results these 3 classifier models are the ones chosen to mover forward into the solution tool construction where they will be subjected to an Hyper Parameter Tuning.

6.3.3 Tuned classifier models

During the Hyper Parameter Tuning process we make use of all our data set, since previously we only were using the no-filter data set. Hence, now we can see how our best classifier models deal with the two additional filters. From the results in 6.3.2 we have to perform the tuning on the Logistic Regression and Balanced Random Forest algorithms using all 3 data sets.

The tuning process of these algorithms is basically the iterative training of the classifier models with different set of hyper parameters. Therefore, we need to create a set of hyper parameters for each algorithm. This requires the study of each algorithms' hyper parameters to see which values make sense, while trying to balancing between the optimal combination of hyper parameters and the execution time for the tuning process. Fig6.4 shows the set of hyper parameters used in the tuning of the Balanced Random Forest and the Logistic Regression algorithms.

Balanced Random Forest	
Hyper Parameter	Values
n_estimators	[100, 400, 700, 1000]
criterion	['gini', 'entropy']
max_depth	[2, 4, 6, 8]
min_samples_split	[2, 4, 8, 10]
min_samples_leaf	[1, 2, 5, 10]
max_features	['auto', 'sqrt', 'log2']

Logistic Regression	
Hyper Parameter	Values
tol	[1E-4, 1E-3, 1E-2, 1E-1]
C	[0.1, 2.0, 10.0]
solver	['sag', 'saga', 'lbfgs']
l1_ratio	[1, 2, 3, 4, 5]

Table 6.4: Set of hyper parameters for each algorithm

Important to notice that to generate the Logistic Regression (balanced) classifier model, we define its hyper parameter "class_weight" as "balanced". On the other hand, the Logistic Regression (over sampling) classifier is "balanced" by over sampling the data set.

The algorithm which we use to perform the tuning process is the Bayes Search Cross Validation. This algorithm allows us to maximize the recall metric. Therefore, for each classifier model tuned, the hyper parameters returned by the function are the ones that maximize the recall values within the search space defined (table 6.4).

The results for the tuned classifier models in Fig.6.5 show the macro recall and improvement from the previous iterations. The results are also divided by the data set used to train and evaluate the classifier models. Each data set is used for both training and testing the classifier model.

Classifier model	Data set used	Macro-recall (%)	Improvement over previous iteration (%)	Training and testing time (min)
Logistic Regression (over sampling)	no filter	92%	0.00%	< 1 min
	innocent-filter	77%	-16.30%	< 1 min
	flaky filter	55%	-40.22%	13 min
Logistic Regression ("balanced")	no filter	92%	0.00%	68 min
	innocent-filter	84%	-8.07%	10 min
	flaky filter	23%	-75.00%	60 min
Balanced Random Forest	no filter	92%	0.00%	30 min
	innocent-filter	93%	+2.20%	19 min
	flaky filter	64%	-30.00%	10 min

Table 6.5: Results of the tuned classifier models

By analysing the table, we can clearly see that the classifier models, when trained and tested with the flaky-filter data set, do not show great results, with a huge decrease in the macro-recall values. Regarding the innocent-filter data set, the Balanced Random Forest model shows good results by equalizing the macro-recall values for the previous iteration. And last, with the no filter data set all classifier models maintain the macro-recall values. Summing up we have 4 classifier models which stand out from the rest.

Therefore, in order not to overcrowd the thesis with results' tables, for the two evaluation variations we use only the classifier models highlighted with red in Fig.6.5, which are the ones showing the best macro-recall values.

The next evaluation on the highlighted model classifiers is regarding the other metrics, besides the recall, mentioned in 6.2, by varying the threshold values and limiting the test execution time.

Threshold variation experiment

For the threshold variation evaluation, the first set of results correspond to the default threshold value (0.5). Table 6.6 show the values of all metrics for the threshold variation evaluation with threshold at that value.

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)		Median of time saved (s)		Median # selected tests		Median % of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	91.84%	88%	91%	1042	35	1219	419	2181	93	100%
LR ("balanced") / no filter	92.00%	88%	93%	1098	50	1174	418	2245	143	100%
BRF / no filter	92.45%	90%	95%	1189	165	854	419	2590	1918	100%
BRF / innoc-filter	93.36%	80%	95%	1041	64	944	405	2176	245	100%

Table 6.6: Threshold variation results with threshold at 0.5

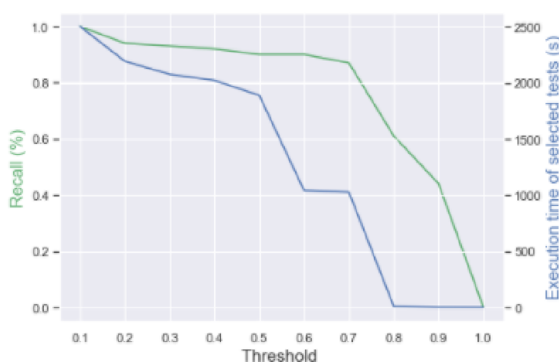
Some interesting facts worth pointing out are that the Balanced Random Forest classifier trained and tested with the innocent-filter data sets have the highest recall value of all the classifier models.

Regarding the micro-recall average, the Balanced Random Forest (with no filter and innocent-filter) classifiers have the highest average micro-recall for the Development test stage. The Balanced Random Forest (BRF) classifier with the no filter data set has the highest average micro-recall for the CorePlatform test stage with 90%.

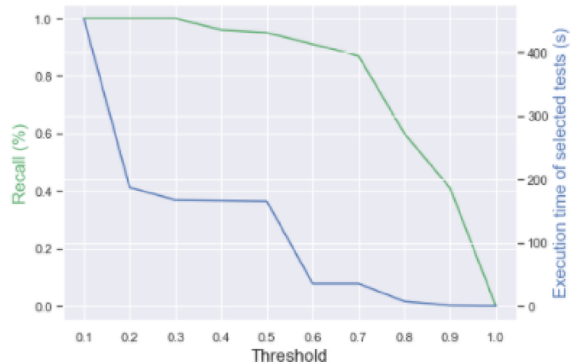
In terms of time saved on the execution of tests, all classifiers show similar saved time in the Development stage. However, for the CorePlatform stage, the Logistic Regression classifiers show best results in this topic, with over 1100 seconds (18 minutes and 20 seconds) saved in tests' execution time.

The BRF classifier (no filter data set) shows the highest values of selected tests in terms of both stages. Also, we see that all model classifiers show great capacity to select relevant tests, given the median percentage of times the models select at least one failing test.

The figures 6.1- 6.4 do not include all of this information. Since we do not want to overcrowd this chapter with tables like the previous one for all thresholds, instead we plotted 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 which are more representative of the classifier models performance. For each classifier model we plotted 2 graphs, one for each test stage.

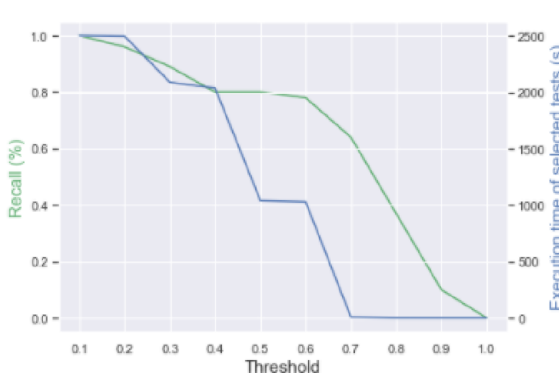


((a)) Recall and test execution time - CorePlatform stage

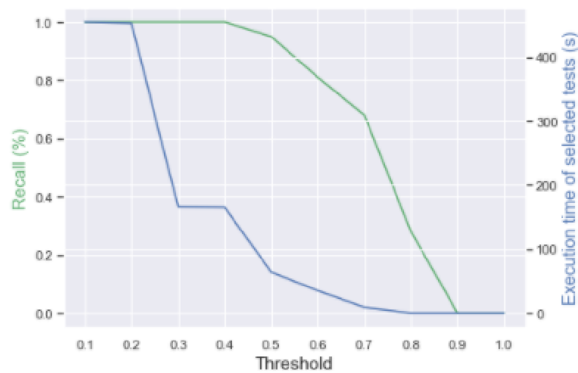


((b)) Recall and test execution time - Development stage

Figure 6.1: Threshold variation for the BRF (no filter data set) classifier model

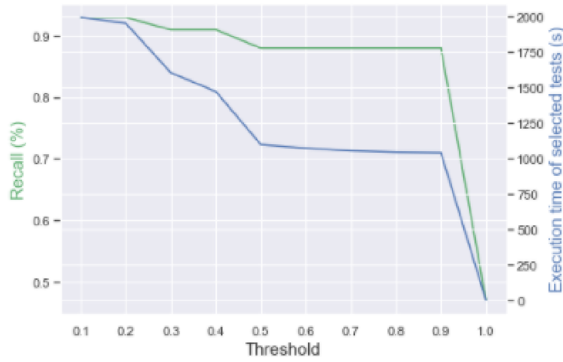


((a)) Recall and test execution time - CorePlatform stage

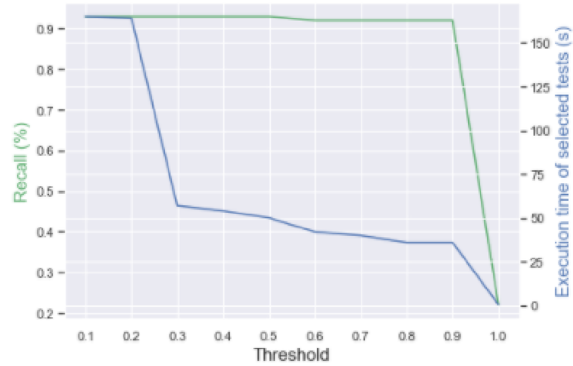


((b)) Recall and test execution time - Development stage

Figure 6.2: Threshold variation for the BRF (innocent-filter data set) classifier model

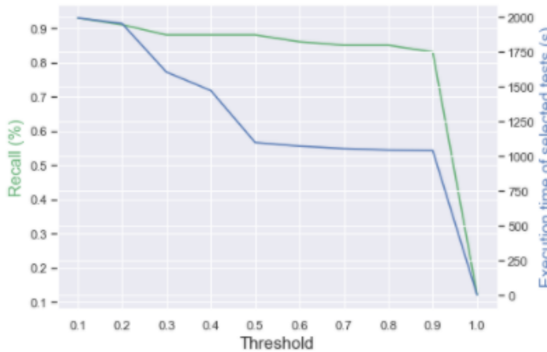


((a)) Recall and test execution time - CorePlatform stage

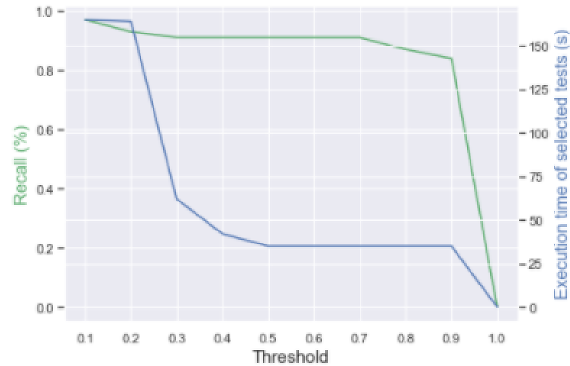


((b)) Recall and test execution time - Development stage

Figure 6.3: Threshold variation for the LR-B (no filter data set) classifier model



((a)) Recall and test execution time - CorePlatform stage



((b)) Recall and test execution time - Development stage

Figure 6.4: Threshold variation for the BRF-OS (no filter data set) classifier model

Looking at the results we can see a clearly distinguish two behaviours from the algorithms used - Balanced Random Forest and Logistic Regression. In the first one, we see a more subtle variation of the micro-recalls values. If we look closely to the micro-recall values from threshold values smaller than 0.5 in the models from this algorithm, we almost see an increase in these values. Whereas, in the Logistic Regression models, we can notice not a great variation of micro-recall values before reaching high threshold values.

Also the overall micro-recall values from the Balanced Random Forest classifiers are higher than the ones from the Logistic Regression classifiers. Now, regarding each stage individually we can see that the first two classifier models manage 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 can see 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%.

In terms of the CorePlatform stage, the results are more similar trough out all classifier models. Still, the one which gets better results is the BRF-no filter data set model, 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.

Time Limit variation experiment

Regarding the time limit variation evaluation, given the median execution time of the stages of 2500 for CorePlatform and 450 seconds for Development, these are the first time limit values for the first results in this evaluation variation. Table 6.7 shows the metrics values for this pair of time limits.

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median # of selected tests'		Median % of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	99.90%	100%	100%	5986	5652	100%
LR ("balanced") / no filter	99.95%	100%	100%	6010	5903	100%
BRF / no filter	99.88%	100%	100%	5980	5822	100%
BRF / innoc-filter	99.90%	100%	100%	5999	5680	100%

Table 6.7: Metrics values for the time limits 2500 (CorePlatform) and 450 seconds (Developement)

We can see that all classifier models have perfect micro-recall and almost perfect recall values, for those time limits, which is expected.

After the results from the threshold variation, we decided to select, for each classifier model, different pairs of time limits for the 2 stages, by taking into account specific time limits which show high micro-recall values. Therefore, to better categorize the results by time limits we decided to choose the best pair of time limits for each classifier and test stage. Then, we test each others' classifiers best pairs.

These best pairs have to take into account the micro-recall. Specifically, for stage Developement, we prioritize time limits with micro-recalls near the 100%, given the short amount of time of the stage. In contrast, in the CorePlatform stage we prioritize time limits with +90% of micro-recall.

The time limits for each classifier model and stage were:

- BRF-no filter data set: Core Platform - 2100 secs; Development - 188 secs;
- BRF-innocent filter data set: Core Platform - 2200 secs; Development - 167 secs;
- BRF-no filter data set: Core Platform - 1473 secs; Development - 51 secs;
- BRF-no filter data set: Core Platform - 1955 secs; Development - 36 secs;

These values were selected based on the threshold variation results for all classifiers model. These results will be in the Appendix chapter.

Tables 6.8- 6.11 show the metrics results from classifiers models with the time limits mentioned above.

In table 6.8, with exception to the BRF - innocent filter data set model, all classifiers show equal recall and micro-recall values. The LR(OS) - no filter data set classifier has the highest recall. However, it is the other Logistic Regression model which has the best combination of micro-recall for both stages.

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median # of selected tests'		Median % of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	97.34%	94%	97%	5034	2096	100%
LR ("balanced") / no filter	97.16%	96%	97%	5217	2101	100%
BRF / no filter	97.24%	94%	97%	5090	2150	100%
BRF / innoc-filter	96.54%	88%	89%	5156	2137	100%

Table 6.8: Time limit variation results - Core Platform - 2100 secs, Development - 188 secs

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median # of selected tests'		Median % of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	97.70%	97%	94%	5311	1932	100%
LR ("balanced") / no filter	97.40%	98%	97%	5453	1939	100%
BRF / no filter	97.66%	94%	97%	5340	1942	100%
BRF / innoc-filter	96.98%	93%	89%	5310	1943	100%

Table 6.9: Time limit variation results - Core Platform - 2200 secs, Development - 167 secs

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median # of selected tests'		Median % of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	53.68%	79%	50%	3204	197	100%
LR ("balanced") / no filter	61.16%	74%	56%	3699	254	100%
BRF / no filter	57.68%	83%	56%	3124	237	100%
BRF / innoc-filter	63.39%	62%	34%	3208	172	100%

Table 6.10: Time limit variation results - Core Platform - 1473 secs, Development - 51 secs

Classifier model / data set used	Recall (%)	Average micro-recall (%)		Median # of selected tests'		Median % of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	
LR (OS) / no filter	70.16%	93%	47%	4789	105	100%
LR ("balanced") / no filter	72.50%	95%	52%	4860	151	100%
BRF / no filter	72.48%	93%	49%	4747	128	100%
BRF / innoc-filter	85.19%	87%	19%	4793	115	100%

Table 6.11: Time limit variation results - Core Platform - 1955 secs, Development - 36 secs

In table 6.9, with the increase in the time limit for the CorePlatform stage, we see that all models increase their micro-recall values for this stage. The LR(OS) - no filter data set classifier still has the highest recall, but is again the other Logistic Regression classifier with the best combination of micro-recall values for both stages.

In table 6.10, shows poor results from all model classifiers, which indicates that the time limit used

here are too low to have high recall and micro-recall values.

In table 6.11, regarding the CorePlatform stage, we had already seen that time limit values around the 2000 secs showed good micro-recall results. However, by decreasing more the time limit for the Development stage we did not achieved any improvements in micro-recall values.

Chapter 7

Conclusions

In this chapter we discuss the work developed by comparing its results with the results obtained by Daniel Correia's thesis work with his method coverage approach. We also present future approaches on the test selection subject that can help improve the results and highlight the main contributions of this thesis by introducing a potential integration of the solution tool into the CI OutSystems' pipeline.

7.1 Discussion

The main goal of this thesis was to reduce the time developers have to wait to receive feedback on their code submissions. This thesis describes the process to build a Machine Learning classifier model which can correctly identify which tests to run for every developer commit to achieve this main goal.

In this process we faced some challenges. One of them was regarding the calculation of features values. The initial idea for some features had to be changed in order not to make our data sets larger than they already are. For example, for the "Max File Failure Rate", the initial idea was to correspond every code file individually to every test file. This made the data set 10 times larger, which made the training of the classifier model very long. Another challenge to overcome, was regarding the commits and builds aggregation into testruns. This nuance was found when analysing the data from the databases and confirming missing builds and commits.

Differently, the process of training and tuning the classifier models was fairly straight forward, using the sklearn python package.

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 (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.

7.1.1 Results comparison

Given the context of this thesis in the Test Selection at OutSystems, we decided to compare this thesis' solution with the others solutions from Diogo and Daniel's work. To obtain the fairest results possible, we used the same time interval of the testing set (April 1st 2020 to April 8th 2020).

The results from Diogo's tool solution are described in table 7.1.

Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)		Median of time saved (s)		Median # selected tests		Median % of times, at least, one test selected
	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	
8.84%	10%	3%	0	0	2499	454	0	0	0%

Table 7.1: Results from Diogo's solution tool

Given the challenges presented in his thesis with poor mapping of external calls from test files execution, Diogo's results are far inferior to the ones achieved by this thesis.

Unfortunately, regarding Daniel's solution tool, it was not possible to replicate his tool to the our time interval. For that, we cannot fairly compare this thesis work with his. However, the recall values obtained in the his tool's period of testing were far lower than the ones obtained in this thesis, given that he faced some challenges regarding the data coverage extraction and the limited applicability to cross-language code.

7.2 Future Work

Throughout this thesis there were some approaches left aside due to limited time. In this section we present 1 example of an addiction to the work done and 1 different approach to the test selection process using Contextual bandits.

7.2.1 Tool with warning messages

In the paper [34] from Google, the authors, empirically study the correlations that exist between their code, test cases, developers, programming languages, and code change and test-execution frequencies. More specifically, in their implementation they warn developers, while they write code, of the impact of their latest changes on quality. For example, if a developer is to commit a certain code file, in case that particular code file is known to cause a high percentage of Google's breakages, an alert is issued with this information. This way, the developer is more careful about said code file and performs a more thorough code review.

Adding this kind of messages in a pre-commit stage and complementing the solution tool of this thesis would be something which would enrich the value of the work of this thesis. This would not impact directly the results from the model classifier. Instead it would encourage better coding practices, since developers would be more careful when committing a ”dangerous” file.

7.2.2 Contextual bandits approach

Contextual Bandits classifies as a Reinforcement Learning algorithm. Reinforcement Learning algorithm is described as the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment. Kaelbling et al. in [37] describes the Reinforcement Learning Model as:

- a discrete set of environment states.
- a discrete set of agent actions.
- a set of scalar reinforcement signals; typically 0 or 1, or real numbers.

The traditional Reinforcement Learning process involves the agent receiving as input, in each iteration, information about the current state of the environment. Then, the agent chooses an action, that changes the state of the environment and the agent receives a scalar reinforcement signal, product of the previous transition. The objective of the agent is to find a policy that maximizes the sum of these reinforcement signals over all iterations.

A contextual bandit algorithm can be seen as a simplification of the Reinforcement Learning algorithm. The difference lays in the fact that the agent in Reinforcement Learning can make multiple consecutive actions, and reward information is sparse, thus making it harder to train a model. Where in a contextual bandit algorithm, the agent makes one action and gets its reward. In future actions it will, not only take into account the state of the environment (context), but also the reward previously received for that action.

So, one possibility of implementation in the context of test selection, would be for each developer commit, the algorithm selects a subset of tests. The metrics we want to maximize are recall and time saved in test executions. After the full regression testing phase, the algorithm receives positive or negative reward based on the comparison of the values from the current and previous regression testing phase.

7.3 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.

7.3.1 OutSystems

In order to implement the solution tool in the OutSystems context, the use of the tool itself, is preceded by the extraction of data from the company's database to build a similar data set to the one showed in previous chapter. The only difference is that this data set had to be updated (for example every 15 days, depending on the frequency of the developers commits) in order to maintain accurate values for all features. In every update on the data set, the classifier model had to be trained in order to check the continuance of its prediction capability. After the model is trained and tested, the features values correspondent to the developers commit need to be calculated. After this, the developer runs the solution tool for his commit. Fig.7.1 shows the steps for the implementation of the tool in the OutSystems' CI pipeline.

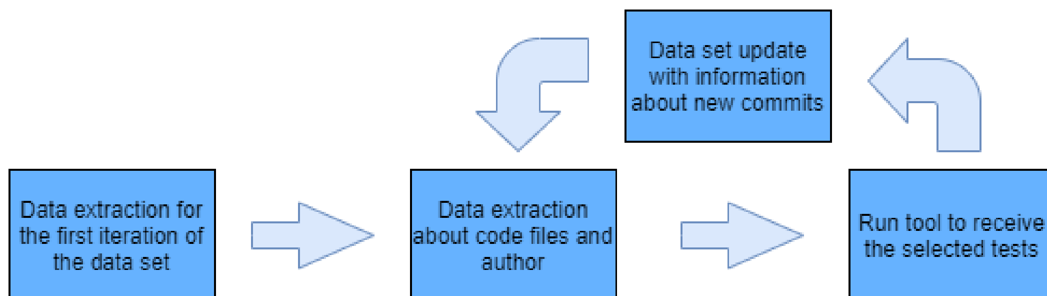


Figure 7.1: Tool implementation example into the OutSystems CI pipeline

In the end of this thesis, after all the results were obtained, we did a small questionnaire to 5 developers at OutSystems, regarding the type of output the tool should give to the developers. In this questionnaire we explain the main goal of the work developed in the thesis and presented 3 different variations of the tool's output results.

The first variation would be a tool which simply returned a list of predicted tests.

The idea of the second variation comes from the Time Limit Evaluation, where the tool would return the tests ordered by the probability value given by the classifier model. After, this, the developer would decide the time limit for the execution of this ordered test list.

The third variation would be a combination of both variations above, but more informative. The tool would return the list of tests predicted by the classifier model (to fail) with information about its execution time and the number of predicted tests. However, the tool would give the opportunity to the developer to increase or decrease the time of the tests execution time.

The feedback received from the developers was very good, specifically regarding the third variation,

where one of the developers highlighted that this addition to the tool would ”promote” small changes by developers at a time, instead of large ones. Since it would be relatively fast to receive feedback from the solution tool, developers would resort to the tool output more often.

Bibliography

[1] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 426–437, 2016.

[2] J. Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59 (7):78–87, 2016.

[3] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab.*, 22(2):67–120, 2012.

[4] A. A. Philip, R. Bhagwan, R. Kumar, C. S. Maddila, and N. Nagappan. Fastlane: test minimization for rapidly deployed large-scale online services. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 408–418, 2019.

[5] 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.

[6] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 643–653, 2014.

[7] P. M. Pereira. *Analysis of Network Attacks and Security Events using Modern Data Visualization Techniques*. PhD thesis, 2015.

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

[9] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 211–222, 2015.

[10] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007.

[11] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[12] D. P. Solomatine, M. Maskey, and D. L. Shrestha. Eager and lazy learning methods in the context of hydrologic forecasting. In *Proceedings of the International Joint Conference on Neural Networks, IJCNN 2006, part of the IEEE World Congress on Computational Intelligence, WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006*, pages 4847–4853, 2006.

[13] F. Pereira, T. M. Mitchell, and M. Botvinick. Machine learning classifiers and fmri: A tutorial overview. *NeuroImage*, 45(1):S199–S209, 2009.

[14] E. B. Hunt, J. Marin, and P. J. Stone. Experiments in induction. *Academic Press*, 1966.

[15] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[16] I. Kononenko. Estimating attributes: Analysis and extensions of RELIEF. In *Machine Learning: ECML-94, European Conference on Machine Learning, Catania, Italy, April 6-8, 1994, Proceedings*, pages 171–182, 1994.

[17] A. K. Jain, J. Mao, and K. M. Mohiuddin. Artificial neural networks: A tutorial. *IEEE Computer*, 29 (3):31–44, 1996.

[18] L. E. Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[19] T. G. Dietterich. Ensemble learning. 2002.

[20] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[21] A. Liaw and M. Wiener. Classification and regression by randomforest. 2002.

[22] R. E. Schapire. A brief introduction to boosting. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 1401–1406, 1999.

[23] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, pages 148–156, 1996.

[24] J. Friedman. Greedy function approximation: a gradient boosting machine. In *Annals of Statistics 29(5)*, page 1189–1232, 2001.

[25] R. Maclin and D. W. Opitz. An empirical evaluation of bagging and boosting. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 546–551, 1997.

[26] J. R. Quinlan. Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1*, pages 725–730, 1996.

[27] M. Feurer and F. Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019.

[28] T. M. Padmaja, N. Dhulipalla, P. R. Krishna, R. S. Bapi, and A. Laha. An unbalanced data classification model using hybrid sampling technique for fraud detection. In *International Conference on Pattern Recognition and Machine Intelligence*, pages 341–348. Springer, 2007.

[29] J. Micco. The state of continuous integration testing @google. 2017.

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

[31] 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.

[32] O. Legunsen, A. Shi, and D. Marinov. STARTS: static regression test selection. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 949–954, 2017.

[33] I. Roshanski, M. Kalech, R. Stern, and A. Elmishali. The cold start problem in software fault prediction. 2019.

[34] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming google-scale continuous testing. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 233–242, 2017.

[35] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. Deflaker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 433–444, 2018.

[36] R. Martins, R. Abreu, P. Santos, and J. Nadkarni. Test suite selection guided by machinelearning. Master's thesis, Instituto Superior Técnico, 2019.

[37] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res.*, 4:237–285, 1996.

Appendix A

Classifier models metrics values

Threshold	Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)		Median of time saved (s)		Median # selected tests		Median # of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	
10%	100%	100%	100%	2505	454	35	33	6018	5932	100%
20%	99.95%	94%	100%	2195	187	297	267	5312	2152	100%
30%	96.82%	93%	100%	2078	167	412	288	5147	1944	100%
40%	93.08%	92%	96%	2023	166	443	289	5032	1931	100%
50%	92.45%	90%	95%	1190	165	854	419	2590	1918	100%
60%	92.30%	90%	91%	1041	35	1297	420	2177	95	100%
70%	90.50%	87%	87%	1027	35	1469	420	2131	84	100%
80%	69.54%	61%	60%	9	7	2496	451	31	22	100%
90%	42.95%	44%	41%	1	0.75	2499	453	4	3	100%
100%	0.00%	0%	0%	0	0	2506	454	0	0	100%

Figure A.1: Threshold variation results for the Balanced Random Forest-no filter data set classifier model

Threshold	Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)		Median of time saved (s)		Median # selected tests		Median # of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	
10%	100%	100%	100%	2505	455	0	0	6018	5941	100%
20%	99.96%	96%	100%	2499	453	0	0	6015	5916	100%
30%	99.92%	89%	100%	2090	166	345	289	5166	1938	100%
40%	93.56%	80%	100%	2039	165	451	289	5021	1921	100%
50%	93.36%	80%	90%	1041	63	944	405	2176	245	100%
60%	90.41%	78%	81%	1030	35	1464	420	2139	89	100%
70%	79.47%	64%	68%	9	9	2476	452	31	24	100%
80%	46.11%	37%	21%	0	0	2499	454	3	0	100%
90%	6.30%	10%	0	0	0	2505	455	0	0	0%
100%	0.00%	0%	0%	0	0	2506	454	0	0	100%

Figure A.2: Threshold variation results for the Balanced Random Forest-innocent filter data set classifier model

Threshold	Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)		Median of time saved (s)		Median # selected tests		Median # of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	
10%	92.47%	93%	93%	1995	165	504	290	4916	1915	100%
20%	92.35%	93%	93%	1954	164	525	290	4826	1910	100%
30%	92.22%	91%	93%	1605	57	702	401	3730	269	100%
40%	92.07%	91%	93%	1472	54	963	416	3631	188	100%
50%	92.01%	88%	93%	1098	50	1174	419	2245	143	100%
60%	91.94%	88%	92%	1072	42	1218	419	2226	143	100%
70%	91.85%	88%	92%	1054	40	1218	419	2204	143	100%
80%	91.79%	88%	92%	1044	36	1338	419	2181	136	100%
90%	91.73%	88%	92%	1040	35	1402	419	2176	114	100%
100%	18.77%	47%	22%	0.97	0.54	2502	453	1	3	100%

Figure A.3: Threshold variation results for the Logistic Regression (balanced)-no filter data set classifier model

Threshold	Recall (%)	Average micro-recall (%)		Median of selected tests' execution time (s)		Median of time saved (s)		Median # selected tests		Median # of times, at least, one test selected
		CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	CP stage	Dev. stage	
10%	94.66%	93%	97%	2003	165	491	289	4960	1923	100%
20%	92.41%	91%	93%	1841	164	569	290	4409	1909	100%
30%	92.11%	88%	91%	1683	62	744	403	3808	259	100%
40%	92.02%	88%	91%	1065	42	1218	414	2207	156	100%
50%	91.84%	88%	91%	1042	35	1219	419	2181	93	100%
60%	91.67%	86%	91%	1040	35	1453	420	2174	90	100%
70%	91.60%	85%	91%	1040	35	1466	420	2174	89	100%
80%	89.42%	85%	87%	1035	35	1468	420	2170.0	88	100%
90%	84.25%	83%	84%	764	35	1468	420	1559	88	100%
100%	0.01%	12%	0%	0	0	2503	454	0	0	0%

Figure A.4: Threshold variation results for the Logistic Regression(OS)-no filter data set classifier model