Journal of Software Engineering Research and Development, 2020, 8:4, doi: 10.5753/jserd.2020.602

Reducing the Discard of MBT Test Cases

Thomaz Diniz [**Federal University of Campina Grande** | ***thomaz.morais@ccc.ufcg.edu.br***]
Everton L. G. Alves [**Federal University of Campina Grande** | ***everton@computacao.ufcg.edu.br***]
Anderson G.F. Silva [**Federal University of Campina Grande** | ***andersongfs@splab.ufcg.edu.br***]
Wilkerson L. Andrade [**Federal University of Campina Grande** | ***wilkerson@computacao.ufcg.edu.br***]

Abstract

Model-Based Testing (MBT) is used for generating test suites from system models. However, as software evolves, its models tend to be updated, which may lead to obsolete test cases that are often discarded. Test case discard can be very costly since essential data, such as execution history, are lost. In this paper, we investigate the use of distance functions and machine learning to help to reduce the discard of MBT tests. First, we assess the problem of managing MBT suites in the context of agile industrial projects. Then, we propose two strategies to cope with this problem: (i) a pure distance function-based. An empirical study using industrial data and ten different distance functions showed that distance functions could be effective for identifying low impact edits that lead to test cases that can be updated with little effort. Moreover, we showed that, by using this strategy, one could reduce the discard of test cases by 9.53%; (ii) a strategy that combines machine learning with distance values. This strategy can classify the impact of edits in use case documents with accuracy above 80%; it was able to reduce the discard of test cases by 10.4% and to identify test cases that should, in fact, be discarded.

Keywords: *MBT, Test Case Discard, Suite evolution, Agile Development*

1 Introduction

Software testing plays an important role since it helps gain confidence the software works as expected (Pressman, 2005). Moreover, testing is fundamental for reducing risks and assessing software quality (Pressman, 2005). On the other hand, testing activities are known to be complex and costly. Studies found that nearly 50% of a project's budget is related to testing (Kumar & Mishra, 2016).

In practice, a test suite can combine manually and automatically executed test cases (Itkonen et al., 2009). Although automation is always desired, manually executed test cases are still very important. Itkonen et al. (2009) state that manual testing still plays an important role in the software industry and cannot be fully replaced by automatic testing. For instance, a tester that runs manual tests tends to better exercise a GUI and find new faults. On the other hand, manual testing is often costly (Harrold, 2000).

To reduce the costs related to testing, Model-Based Testing (MBT) can be used. It is a strategy where test suites are automatically generated from specification models (e.g., use cases, UML diagrams) (Dalal et al., 1999; Utting & Legeard, 2007). By using MBT, sound tests can be extracted before any coding, and without much effort.

In agile projects, requirements are often volatile (Beck & Gamma, 2000; Sutherland & Sutherland, 2014). In this scenario, test suites are used as safety nets for avoiding feature regression. Discussions on the importance of test case reuse are not new (Von Mayrhauser et al., 1994). In software engineering, software reuse is key for reducing development costs and improving quality. This is also valid for testing (Frakes, 1994). A test case that finds faults can be a valuable investment (Myers et al., 2011). Good test cases should be stored as a reusable resource to be used in the future (Cai et al., 2009). In this context, an always updated test suite is mandatory. A recent work proposed lightweight specification artifacts for enabling the use of MBT in agile projects (N. Jorge et al., 2018), CLARET. With CLARET, one can both specify requirements using use cases and generate MBT suites from them.

However, a different problem has emerged. As the software evolves (e.g., bug fixes, change requirements, refactorings), both its models and test suite need revisions. Since MBT test suites are generated from requirement models, in practice, as requirements change, the requirement artifacts are updated, new test suites are generated, and the newer suites replace the old ones. Therefore, test cases that were impacted by the edits, instead of updated, are often considered obsolete and discarded (Oliveira Neto et al., 2016).

Although one may find it easy to generate new suites, regression testing is based on a stable test suite that evolves. Test case discarding implies important historical data that are lost (e.g., execution time, the link faults-to-tests, fault-discovering time). Test case historical data is an important tool for assessing system weaknesses and better manage it, therefore, one should not neglect it. For instance, most defect prediction models are based on historical data (He et al., 2012). Moreover, for some strategies that optimize testing resources allocation, historical data is key (Noor & Hemmati, 2015; Anderson et al., 2014). By discarding test cases, and their historical data, a project may miss important information for both improving a project and guiding its future actions. Moreover, in a scenario where previously detected faults guide development, missing tests can be a huge loss. Finally, test case discard and poor testing are known as signs of bad management and eventually lead to software development waste (Sedano et al., 2017).

However, part of a test suite may turn obsolete due to little impacted model updates. Thus, those test cases could be easily reused with little effort and consequently reducing testing discards. Nevertheless, manual analysis is tedious, costly, and time-consuming, which often prevents its applicability in the agile context. In this sense, there is a need for an automatic way of detecting reusable and, in fact, obsolete test cases.

Distance functions map a pair of strings to a number that indicates the similarity level between the two versions (Cohen et al., 2003). In a scenario where manual test cases evolve due to requirement changes, distance functions can be an interesting tool to help us classify the impact of the changes into a test case.

In this paper, first, we assess and discuss the practical problem of model evolution in MBT suites. To cope with this problem, we propose and evaluate two strategies for automatically classifying model edits and tests aiming at avoiding unnecessary test discards. The first is based on distance functions, while the second combines machine learning and distance values.

This work is an extension over our previous one (Diniz et al., 2019) including the following contributions:

- An study using historical data from real industrial projects that investigates the impact of model evolution in MBT suites. We found that 86% of the test cases turn obsolete between two consecutive versions of a requirement file, and those tests are often discarded. Moreover, 52% of the found obsolete tests were caused by *low impact* syntactic edits and could become fully updated with the revision of 25% of the steps.
- An automatic strategy based on distance functions for reclassifying reusable test cases from the obsolete set. This strategy was able to reduce test case discard by 9.53%.
- An automatic strategy based on machine learning and distance functions for classifying test cases and model change impact. This strategy can classify the impact of edits in use case documents with accuracy above 80%, it was able to reduce the discard of test cases by 10.4%, and to identify test cases that should, in fact, be discarded.

This paper is organized as follows. In Section 2, we present a motivational example. The needed background is discussed in Section 3. Section 4 presents an empirical investigation for assessing the challenges of managing MBT suite during software evolution. Sections 5 and 6 present the strategy for classifying model edits using distance functions and the performed evaluation, respectively. Section 7 introduces the strategy that combines machine learning and distance values. Section 8 presents a discussion comparing results from both strategies. In Section 9, some threats to validity are cleared. Finally, Sections 10 and 11 present related works and the concluding remarks.

2 Motivational Example

Suppose that Ann works in a project and wants to benefit from MBT suites. Her project follows an agile methodology where requirements updates are expected to be frequent. Therefore, she decides to use CLARET (N. Jorge et al., 2018), an approach for specifying requirements and generating test suites.

The following requirement was specified using CLARET's DSL (Listing 1): "*In order to access her email inbox, the user must be registered in the system and provide a correct username and password. In case of an incorrect username or password, the system must display an error message and ask for new data.*". In CLARET, an ***ef [flow #]*** mark refers to a possible exception flow, and a ***bs [step #]*** mark indicates a returning point from an exception/alternative to the use case's basic flow.

From this specification, the following test suite can be generated: S1 = {tc1, tc2, tc3}, where tc1 = [bs:1 → bs:2 → bs:3 → bs:4], tc2 = [bs:1 → bs:2 → bs:3 → ef[1]:1 → bs:3 → bs:4], and tc3 = [bs:1 → bs:2 → bs:3 → ef[2]:1 → bs:3 → bs:4] .

Suppose that in the following development cycle, the use case (Listing 1) was revisited and updated due to both requirement changes and for improving readability. Three edits were performed: (i) the message in line 9 was updated to "displays a successful message"; (ii) system message in line 12 was updated to "alerts that username does not exist"; and (iii) both description and system message in exception 3 (line 14) were updated to "Incorrect username/password combination" and "alerts that username and/or password are incorrect", respectively.

Since steps from all execution flows were edited (basic, exception 1, and exception 2), Ann discards S1 and generates a whole new suite. However, part of S1's tests was not much impacted and could be turned to reused with little or no update. For instance, only edit (iii), in fact, changed the semantic of the use case, while (i) and (ii) are updates that do not interfere with the system's behavior. Therefore, only test cases that exercise the steps changed by (iii) should be in fact discarded (tc3). Moreover, test cases that exercise steps changed by (i) and/or (ii) could be easily reused and/or updated (tc1 and tc2).

We believe that an effective and automatic analyzer would help Ann to decide when to reuse or discard test cases, and therefore reduce the burden of losing important testing data.

```
1  systemName "Email"
2  usecase "Log in User" {
3      actor emailUser "Email User"
4      preCondition "There is an active network connection"
5      basic {
6          step 1 emailUser "launches the login screen"
7          step 2 system "presents a form with username and
         password fields and a submit button"
8          step 3 emailUser "fills out the fields and click on
           the submit button"
9          step 4 system "displays a message" ef[1,2]
10     }
11     exception 1 "User does not exist in database" {
12         step 1 system "alerts that user does not exist"  bs
           3
13     }
14     exception 2 "Incorrect password" {
15         step 1 system "alerts that the password is
         incorrect" bs 3
16     }
17     postCondition "User successfully logged"
18 }
```

Listing 1: Use Case specification using CLARET.

3 Background

This section presents the MBT process, the CLARET notation, and the basic idea behind the two strategies used for reducing test case discard, distance functions, and machine learning.

3.1 Model-Based Testing

MBT aims to automatically generate and manage test suites from software specification models. MBT may use different model formats to perform its goals (e.g., Labeled Transition System (LTS) (Tretmans, 2008), UML diagrams (Bouquet et al., 2007)). As MBT test suites are derived from specification artifacts, their test cases tend to reflect the system behavior (Utting et al., 2012). Utting & Legeard (2007) discuss a series of benefits of using MBT, such as sound test cases, high fault detection rates, and test cost reduction. On the other hand, regarding MBT limitations, we can list the need for well-built models, huge test suites, and a great number of obsolete test cases during software evolution.

Figure 1 presents an overview of the MBT process. The system models are specified through a DSL (e.g., UML) and a test generation tool is used to create the test suite. However, as the system evolves, edits must be performed on its models to keep them up-to-date. If any fault is found, the flow goes back to system development. These activities are repeated until the system is mature for release. Note that previous test suites are discarded, and important historical data may be lost in this process.

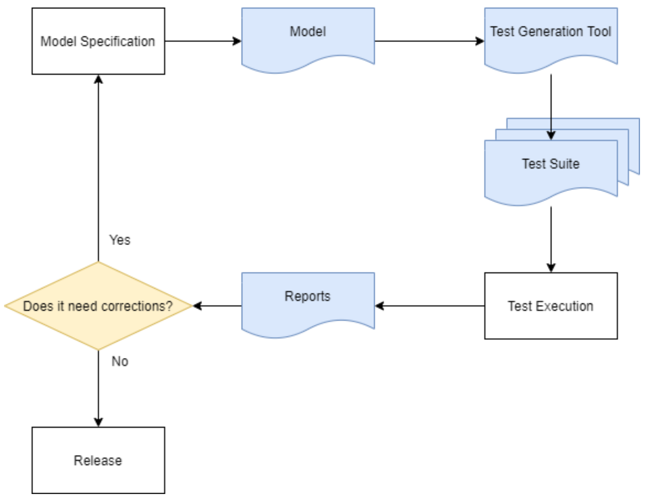

Figure 1. MBT Process

3.2 CLARET

CLARET (N. Jorge et al., 2017, 2018) is a DSL and tool that allows the creation of use case specifications using natural language. It was designed to be the central artifact for both requirement engineering and MBT practices in agile projects. Its toolset works as a syntax checker for use cases description files and provides visualization mechanisms for use case revision. Listing 1 presents a use case specification using CLARET.

From the use case description in Listing 1, CLARET generates its equivalent Annotated Labeled Transition System (ALTS) model (Tretmans, 2008) (Figure 2). Transition labels starting with *[c]* indicate pre or post conditions, while the ones starting with *[s]* and *[e]* are regular and exception execution steps, respectively.

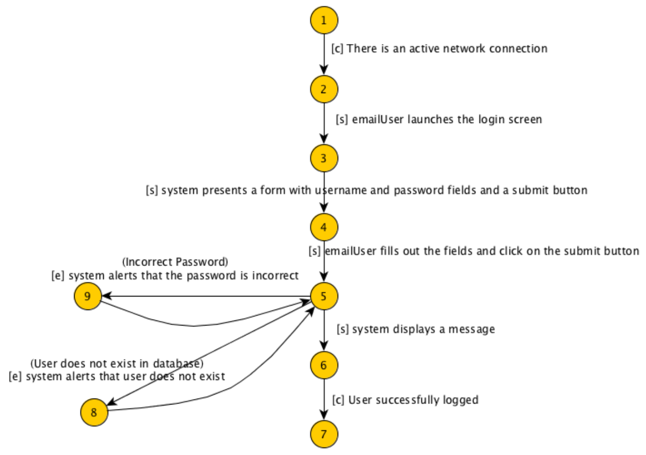
1
[c] There is an active network connection
2
[s] emailUser launches the login screen
3
[s] system presents a form with username and password fields and a submit button
4
[s] emailUser fills out the fields and click on the submit button
(Incorrect Password)
[e] system alerts that the password is incorrect
9
5
[s] system displays a message
6
(User does not exist in database)
[e] system alerts that user does not exist
8
[c] User successfully logged
7

Figure 2. ALTS model of the use case from Listing 1.

CLARET's toolset includes a test generation tool, LTS-BT (Labeled Transition System-Based Testing) (Cartaxo et al., 2008). LTS-BT is an MBT tool that uses as input LTS models and generates test suites by searching for valid graph paths. The generated tests are reported in XML files that can be directly imported to a test management tool, TestLink[1]. The test cases reported in Section 2 were collected from LTS-BT.

3.3 Distance Functions

Distance functions are metrics for evaluating how similar, or different, are two strings (Coutinho et al., 2016). Distance functions have been used in different contexts (e.g., (Runkler & Bezdek, 2000; Okuda et al., 1976; Lubis et al., 2018)). For instance, Coutinho et al. (2016) use distance functions for reducing MBT suites.

There are several distance functions (e.g., (Hamming, 1950; Han et al., 2007; Huang, 2008; De Coster et al., 1; Levenshtein, 1966)). For instance, the Levenshtein function (Levenshtein, 1966; Kruskal, 1983) (equation described below) compares two strings (a and b) and calculates the number of required operations to transform a into b, and vice-versa; where $1_{ai \neq bj}$ is the indicator function equal to 0 when $a_i \neq b_j$ and equal to 1 otherwise, and $lev_{a,b}$ is the distance between the first i characters of a and the first j characters of b.

To illustrate its use, consider two strings a = "kitten" and b = "sitting". Their Levenshtein distance is three, since three operations are needed to transform a to b: (i) replacing 'k' by 's'; (ii) replacing 'e' by 'i'; and (iii) inserting 'g' at the end. A more detailed discussion about the Levenshtein and others functions, as well as an open-source implementation of them are available[2].

[1]http://testlink.org/
[2]https://github.com/luozhouyang/python-string-similarity

$$\text{lev}_{a,b}(i,j) = \begin{cases} max(i,j) & \textit{if min(i,j) = 0} \\ min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{ai \neq bj} \end{cases} & otherwise \end{cases}$$

3.4 Machine Learning

Machine Learning is a branch of Artificial Intelligence based on the idea that systems can learn from data, identify patterns, and make decisions with minimal human intervention (Michie et al., 1994). By providing ways for building data-driven models, machine learning can produce accurate results and analysis (Zhang & Tsai, 2003).

The learning process begins with observations or data (examples), it looks for data patterns, and make future decisions. By applying machine learning, one aims to allow computers to learn without human intervention, and to adjust its actions accordingly.

Machine learning algorithms are often categorized as supervised or unsupervised. Supervised machine learning algorithms (e.g., linear regression, logistic regression, neural networks) use labeled examples from the past to predict future events. Unsupervised machine learning algorithms (e.g., k-Means clustering, Gaussian mixture models) are used when the training data is neither classified nor labeled. It infers a function to describe a hidden structure from unlabeled data.

The use of machine learning in software engineering has grown in the past years. For instance, machine learning methods have been used for estimating development effort (Srinivasan & Fisher, 1995; Baskeles et al., 2007), predicting a software fault-proneness (Gondra, 2008), fault prediction (Shepperd et al., 2014), and improving code quality (Malhotra & Jain, 2012).

4 Analysing the Impact of Model Evolution in MBT Suites

To understand the impact of model evolution in MBT suites, we observed two industrial projects (SAFF and BZC) from industrial partners. Both systems were developed in the context of a cooperation between our research lab and two different companies, Ingenico do Brasil Ltda and Viceri Solution Ltda. The SAFF project is an information system that manages status reports of embedded devices; and BZC is a system for optimizing e-commences logistic activities.

The projects were run by two different teams. Both teams applied agile practices and used CLARET for use case specification and generation of MBT suites.

Both projects use manually executed system-level black-box test cases for regression purposes. In this sense, test case historical data is very important since it can help to keep track of the system evolution and to avoid functionality regression. However, the teams reported that often discard test cases when the related steps on the system use cases are updated in any form, which they refer to as a practical management problem.

Therefore, we mined the projects repositories, traced each model change (use case update), and analyzed its impact on the generated suites.

Table 1. Summary of the artifacts used in our study.

	#Use Cases	#Versions	#Edits
SAFF	13	42	415
BZC	15	37	103
Total	28	79	518

Our goal in this study was to better understand the impact of model updates in the test suites and to measure how much of a test suite is discarded. To guide this investigation, we defined the following research questions:

- **RQ1**: How much of a test suite is discarded due to use case editions?
- **RQ2**: What is the impact of *low* (syntactic) and *high* (semantic) model edits on a test suite?
- **RQ3**: How much of an obsolete test case needs revision to be reused?

4.1 Study Procedure

For each CLARET file (use case model), we collected the history of its evolution in a time frame. In the context of our study, we consider a use case evolution any edit found between two consecutive versions of a CLARET file. Our study worked with 28 use cases, a total of 79 versions, and an average of 5 step edits per CLARET file. Table 1 presents the summary of the collected data. After that, we collected the test suites generated for each version of the CLARET files.

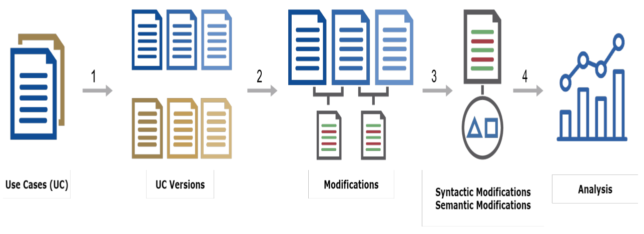

Figure 3. Procedure.

We extracted a change set for each pair of adjacent versions of a CLARET file (*uc*, *uc'*). In our analysis, we considered two kinds of edits/changes: i) *step update*. Any step in the base version (*uc*) that had its description edited in the delta version; and ii) *step removal*. Any step that existed in *uc* but not in *uc'*. We did not consider step additions. Since our goal was to investigate reuse in a regression testing scenario, we considered suites generated using only the base version (*uc*). Consequently, no step addition could be part of the generated tests.

After that, we connected the changeset to the test suites. For that, we ran a script that matched each edited step to the test cases it impacted. We say a test case is impacted by a modification if it includes at least one modified step from the changeset. Thus, our script clustered the tests based on Oliveira Neto et al. (2016)'s classification: *Obsolete*. Test cases that include updated or removed steps. These tests have different actions or system responses, when compared to its previous version; and *Reusable*. Test cases that exercise only unmodified parts of the model specification. All their actions and responses remained the same when compared to its previous version. Figure 3 summarizes our study procedure.

To observe the general impact of the edits, we measured how much of a test suite was discarded due to use case edits. Being s_total the number of test cases generated from a use case (*uc*); s_obs, the number of found obsolete test cases; and N the number of pairs of use cases; we define the *Average number of Obsolete Test Cases* (AOTC) by Equation 1.

$$AOTC = (\sum \frac{s\_obs}{s\_total}) * \frac{1}{N} \quad (1)$$

Then, we manually analyzed each element from the changeset and classified them into *low impact* (syntactic edit), *high impact* (semantic edit), or a combination of both. For this analysis, we defined three versions of the AOTC metric: AOTC_syn, the average number of obsolete test cases due to low impact edits; AOTC_sem, the average number of obsolete test cases due to high impact edits; and AOTC_both, that considers tests with low and highly impacted steps.

Finally, to investigated how much of a test case needs revision, for each test, we measured how many steps were modified. For this analysis, we defined the AMS (Average Modified Steps) metric (Equation 2), which measures the proportion of steps that need revision due to model edits. Being tc_total the number of steps in a given test case; tc_c, number of steps that need revision; and N the number of test cases:

$$AMS = (\sum \frac{tc\_c}{tc\_total}) * \frac{1}{N} \quad (2)$$

4.2 Results and Discussion

Our results evidence that MBT test suites can be very sensitive to any model evolution. A great number of test cases were discarded. On average, 86% (AOTC) of a suite's tests turned obsolete between two consecutive versions of a use case file (Figure 4). This result exposes one of the main difficulties of using MBT in agile projects, requirement files are very volatile. Thus, since test cases are derived from these models, any model edit leads to a great impact on the generated tests.

Although a small number of test cases were reused, the teams in our study found the MBT suites useful. They mentioned that a series of unnoticed faults were detected, and the burden of creating tests was reduced. Thus, we can say that MBT suites are effective, but there is still a need for solutions for reducing test case discard due to model updates.

RQ1: How much of a test suite is discarded due to use case editions? On average, 86% of the test cases became obsolete between two versions of a use case model.

We manually analyzed each obsolete test case. Figure 5 summarizes this analysis. As we can see, 52% of the cases became obsolete due to *low impact* (syntactic) edits in the use case models, while 21% were caused by high impact (semantic) changes, and 12% by both syntactic and semantics changes in the same model. Therefore, more then half of the obsolete set refer to edits that could be easily revised and turned to reusable without much effort (*e.g.*, a step rephrasing, typo fixing).

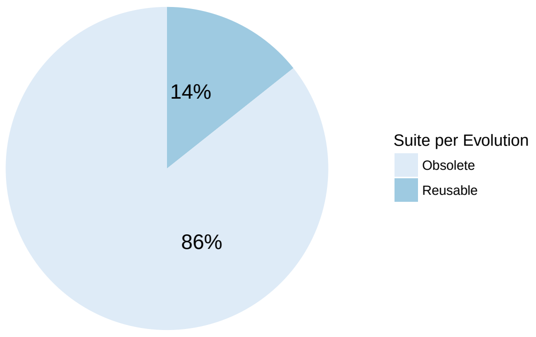

Figure 4. Reusable and obsolete test cases.

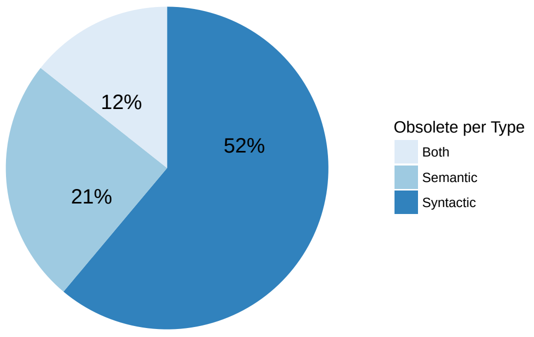

Figure 5. Tests that became obsolete due to a given change type.

RQ2: What is the impact of *low* (syntactic) and *high* (semantic) model edits on a test suite? 52% of the found obsolete tests were caused by *low impact* use case edits (syntactic changes), while 21% were due to *high impact* edits (semantic chances), and 12% by a combination of both.

We also investigated how much of an obsolete test case would need to be revised to avoid discarding. It is important to highlight that this analysis was based only on the number of steps that require revision. We did not measure the complexity of the revision. Figure 6 shows the distribution of the found results. As we can see, both medians were similar (25%). Thus, often a tester needs to review 25% of a test case to turn it reusable, disregarding the impact of the model evolution.

As most *low impact* test cases relate to basic syntactic step updates (e.g., fixing typos, rephrasing), we believe the costs of revisiting them can be minimal. For the highly impacted tests (semantic changes), it is hard to infer the costs, and, in some cases, discarding those tests can still be a valid option. However, a test case discard can be harmful and should be avoided.

RQ3: How much of an obsolete test case needs revision to be reused? In general, a tester needs to revisit 25% of the steps of an obsolete test, regardless of the impact of the model edits.

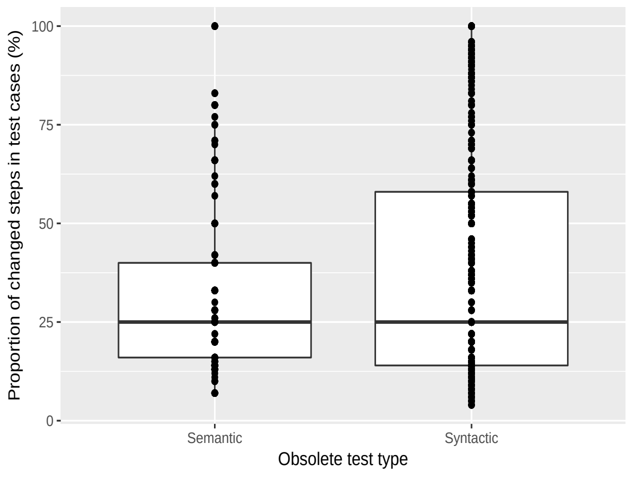

Figure 6. Proportion of the test cases modified by edit type.

5 Distance Functions to Predict the Impact of Test Case Evolution

The study described in Section 4 evidences the challenge of managing MBT suites during software evolution. To test whether distance functions can help to cope with this problem, we ran a second empirical study. The goal of this study was to analyze the use of distance functions to automatically classify changes in use case documents that could impact MBT suites.

5.1 Subjects and Functions

For that, our study was also run in the context of the industrial projects SAFF and BZC. It is important to remember that both projects used agile methodologies to guide the development and updates in the requirement artifacts were frequent. Moreover, their teams used both CLARET (N. Jorge et al., 2017), for use case specification, and LTS-BT (Cartaxo et al., 2008) for generating MBT suites.

As our study focuses on the use of distance functions, we selected a set of ten of the most well-known functions that have been used in different contexts: Hamming (Hamming, 1950), LCS (Han et al., 2007), Cosine (Huang, 2008), Jaro (De Coster et al., 1), Jaro-Winkler (De Coster et al., 1), Jaccard (Lu et al., 2013), Ngram (Kondrak, 2005), Levenshtein (Levenshtein, 1966), OSA (Damerau, 1964), and Sorensen Dice (Sørensen, 1948). To perform systematic analyses, we normalize their results in a way that their values range from zero to one. Values near zero refer to low similarity, while near one values indicate high similarity. We reused open-source implementations of all ten functions[3][4]. To customize and analyze the edits in the context of our study, we created our own tool and scripts that were verified through a series of tests.

We mined the projects' repository and collected all use case edits. Each of these edits would then impact the test cases. We call "impacted" any test case that includes steps that were updated during model maintenance. However, we aim to use distance functions to help us to classify these edits and avoid the test case discard.

[3]https://github.com/luozhouyang/python-string-similarity
[4]https://rosettacode.org/wiki/Category:Programming_Tasks

Table 2. Classification of edits.

Steps Description		
Version 1	Version 2	Classification
"Extract data on offline mode."	"Show page that requires new data."	high impact
"Show page that requires new data."	"Show page that requires new terminal data."	low impact
"Click on Edit button"	"Click on the Edit button"	low impact

To guide our investigation, we defined the following research questions:

- **RQ4**: Can distance functions be used to classify the impact of edits in use case documents?
- **RQ5**: Which distance function presents the best results for classifying edits in use case documents?

5.2 Study Setup and Procedure

Since all use case documents were CLARET files, we reused the data collected in the study of Section 4. Therefore, a total of 79 pairs of use case versions were analyzed in this study, with a total of 518 edits. Table 1 summarizes the data.

After that, we manually analyzed each edit and classified them between *low impact* and *high impact*. A **low impact** edit refers to changes that do not alter the system behavior (a pure synthetic edit), while a **high impact** edit refers to changes in the system expected behavior (semantic edit). Table 2 exemplifies this classification. While the edit in the first line changes the semantics of the original requirement, the next two refer to edits performed for improving readability and fixing typos. During our classification, we found 399 *low impact* and 27 *high impact* edits for the SAFF system, and 92 *low* and 11 *high impact* for BZC. This result shows that use cases often evolve for basic description improvements, which may not justify the great number of discarded test cases in MBT suites.

After that, for each edit (original and edited versions), we ran the distance functions using different configuration values and observed how they classified the edits compared to our manual validation.

5.3 Metrics

To help us evaluate the results, and answer our research questions, we used three of the most well-known metrics for checking binary classifications: *Precision*, which is the rate of relevant instances among the found ones; *Recall*, calculates the rate of relevant retrieved instances over the total of relevant instances; and *Accuracy*, which combines Precision and Recall. These metrics have been used in several software engineering empirical studies (e.g., (Nagappan et al., 2008; Hayes et al., 2005; Elish & Elish, 2008)). Equations 3, 4 and 5 present those metrics, where TP refers to the number of cases a distance function classified an edit as low impact and the manual classification confirms it; TN refers to the number of matches regarding high impact edits; FP refers to when the automatic classification reports low impact edits when in fact high impact edits were found; and FN is when the automatic classification reports high impact when in fact should be low impact edits.

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5}$$

5.4 Results and Discussion

To answer RQ4, we first divided our dataset of use case edits into two (low and high impact edits), according to our manual classification. Then, we ran the distance functions and plotted their results. Figures 7 and 8 show the box-plot visualization of this analysis considering found low (Figure 7) and high impacts (Figure 8). As we can see, most low impact edits, in fact, refer to low distance values (median lower than 0.1), for all distance functions. This result gives us evidence that low distance values can relate to low impact edits and, therefore, can be used for predicting low impact changes in MBT suites. On the other hand, we could not find a strong relationship between high impact edits and distance values. Therefore we can answer RQ4 stating that distance functions, in general, can be used to classify low impact.

RQ4: Can distance functions be used to classify the impact of edits in use case documents? Low impact edits are often related to lower distance values. Therefore, distance functions can be used for classifying low impact edits.

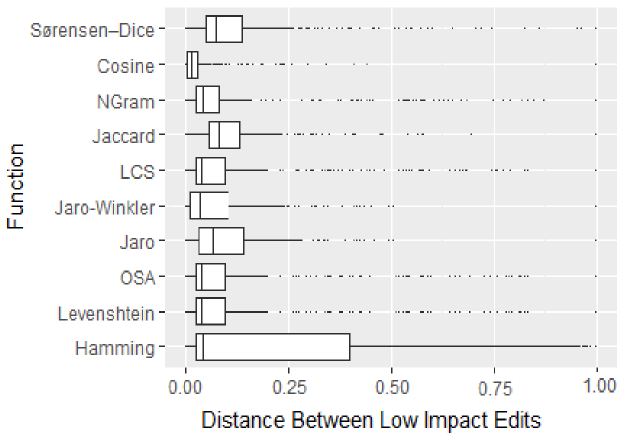

Figure 7. Box-plot for low impact distance values.

As for automatic classification, we need to define an effective *impact threshold*, for each distance function, we run an exploratory study to find the optimal configuration for using each function. By impact threshold, we mean the distance value for classifying an edit as low or high impact. For instance, consider a defined impact threshold of x% to be used with function *f*. When analyzing an edit from a specification document, if *f* provides a value lower than *x*, we say the edit is *low impact*, otherwise it is *high impact*. Therefore, we design a study where, for each function, we vary the defined *impact threshold* and we observed how it would impact Precision and Recall. Our goal with this analysis is to identify the more effective configuration for each function. We range the impact threshold between $[0; 1]$.

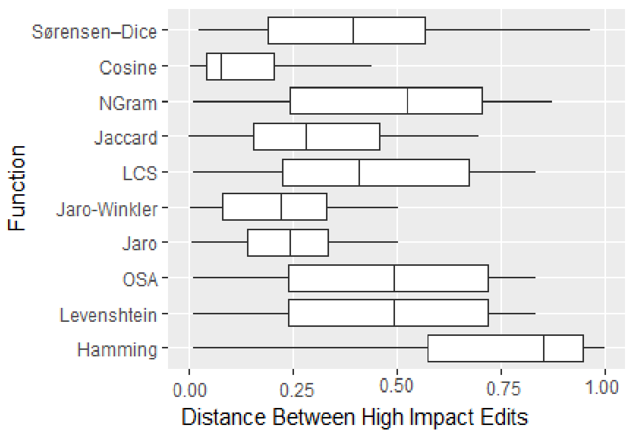

Figure 8. Box-plot for high impact distance values.

To find this optimal configuration, we consider the interception point between the Precision and Recall curves, since it reflects a scenario with less mistaken classifications (false positives and false negatives). Figure 9 presents the analysis for the Jaccard functions. Its optimal configuration is highlighted (impact threshold of 0.33) – the green line refers to the Precision curve, the blue line to the Recall curve, and the red circle shows the point both curves meet. Figure 10 presents the analysis for the other functions.

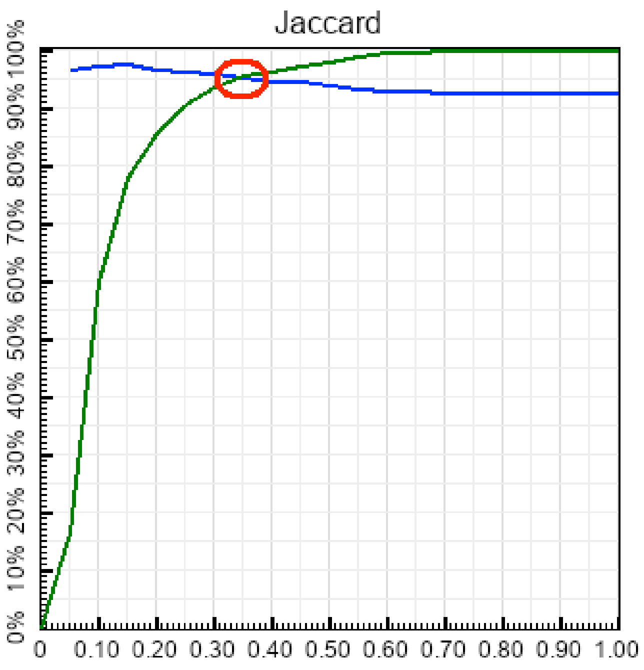

Figure 9. Best impact threshold for the Jaccard function.

Table 3 presents the optimal configuration for each function and the respective precision, recall, and accuracy values. These results reinforce our evidence to answer RQ4 since all functions presented accuracy values greater than 90%. Moreover, we can partially answer RQ5, since now

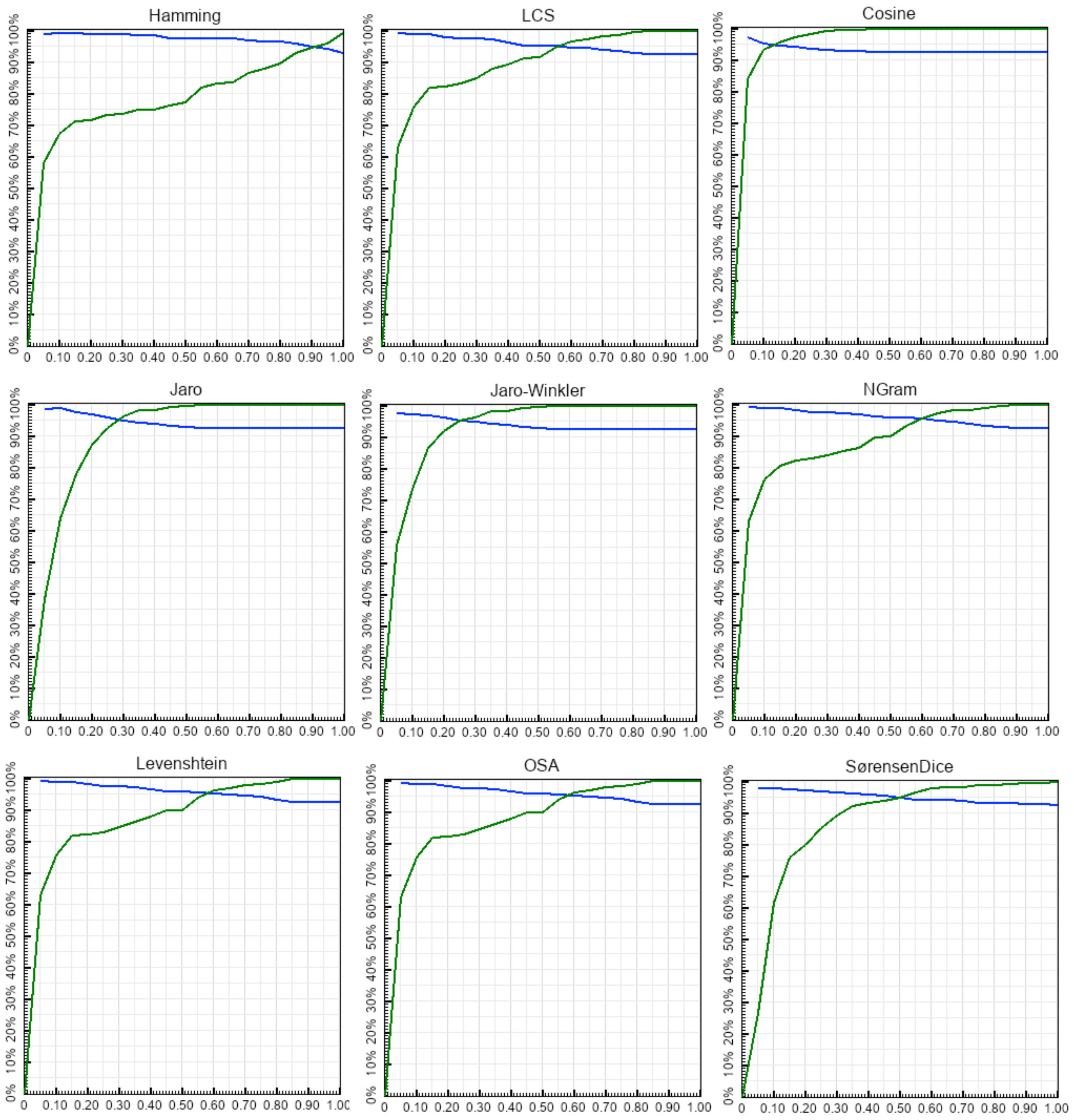

Figure 10. Exploratory study for precision and recall per distance function.

we found, considering our dataset, the best configuration for each distance function. To complement our analysis, we went to investigate which function performed the best. First, we run proportion tests considering both the functions all at once and pair-to-pair. Our results show, with 95% of confidence, could not find any statistical differences among the functions. This means that distance function for automatic classification of edits impact is effective, regardless of the chosen function (RQ5). Therefore, in practice, one can decide which function to use based on convenience aspects (e.g., easier to implement, faster).

RQ5: Which distance function presents the best results for classifying edits in use case documents? Statistically, all ten distance functions performed similarly when classifying edits from use case documents.

6 Case Study

To reassure the conclusions presented in the previous section, and to provide a more general analysis, we ran new studies considering a different object, TCOM. TCOM is an industrial software also developed in the context of our cooperation with the Ingenico Brasil Ltda. It controls the execution and manages testing results of a series of hardware parts. It

Table 3. Best configuration for each function and respective precision, recall and accuracy values.

Function	Impact Threshold	Precision	Recall	Accuracy
Hamming	0.91	94.59%	94.79%	90.15%
Levenshtein	0.59	95.22%	95.42%	91.31%
OSA	0.59	95.22%	95.42%	91.31%
Jaro	0.28	95.01%	95.21%	90.93%
Jaro-Winkler	0.25	95.21%	95.21%	91.12%
LCS	0.55	94.99%	94.79%	90.54%
Jaccard	0.33	95.22%	95.42%	91.31%
NGram	0.58	95.41%	95.21%	91.31%
Cosine	0.13	95%	95%	90.73%
Sørensen–Dice	0.47	94.99%	94.79%	90.54%

Table 4. Summary of the artifacts for the TCOM system.

	#Use Cases	#Versions	#Edits
TCOM	7	32	133

is important to highlight that a different team ran this project, but in a similar environment: CLARET use cases for specification and generated MBT suites. The team also reported similar problems concerning volatile requirements, and frequent test case discards.

First, similar to the procedure applied in Section 5.2, we mined TCOM's repository and collected all versions of its use case documents and their edits. Table 4 summarizes the collected data from TCOM. Then, we manually classified all edits between low and high impact to serve as validation for the automatic classification. Finally, we ran all distance functions considering the optimal *impact thresholds* (Table 3 - second column) and calculated Precision, Recall and Accuracy for each configuration (Table 5).

Table 5. TCom - Evaluating the use of the found impact threshold for each function and respective precision, recall and accuracy values.

Function	Impact Threshold	Precision	Recall	Accuracy
Hamming	0.91	87.59%	94%	84.96%
Levenshtein	0.59	87.85%	94%	85.71%
OSA	0.59	87.85%	94%	85.71%
Jaro	0.28	89.52%	94.00%	87.22%
Jaro-Winkler	0.25	94.00%	89.52%	87.22%
LCS	0.55	89.62%	95%	87.97%
Jaccard	0.33	89.52%	94%	87.22%
NGram	0.58	87.85%	94%	85.71%
Cosine	0.13	88.68%	94%	86.47%
Sørensen–Dice	0.47	88.68%	94%	86.47%

As we can see, the found impact thresholds presented high precision, recall, and accuracy values when used in a different system and context (all above 84%). This result gives as evidence that, distance functions are effective for automatic classification of edits (RQ4) and that the found impact thresholds performed well for a different experimental object (RQ5).

In a second moment, we used this case study to evaluate how our approach (using distance functions for automatic classification) can help reducing test discards:

- **RQ6**: Can distance function be used for reducing the discard of MBT tests?

To answer RQ6, we considered TCOM's MBT test cases generated from its CLARET files. Since all distance functions behave similarly (Section 5.4), in this case study we used only Levenshtein's function to automatically classify the edits and to check the impact of those edits in the tests. In a common scenario, which we want to avoid, any test case that contains an updated step would be discarded. Therefore, in the context of our study, we used the following strategy *"only test cases that contain high impact edits should be discarded, while test cases with low impact edits are likely to be reused with no or little updating"*. The rationale behind this decision is that low impact edits often imply on little to no changes to the system behavior. Considering system-level black-box test suites (as the ones from the projects used in our study), those tests should be easily reused. We used this strategy and we first applied Oliveira's et al.'s classification (Oliveira Neto et al., 2016) that divided TCOM's tests among three sets: *obsolete* – test cases that include impacted steps; *reusable* – test cases that were not impacted by the edits; and *new* – test cases that include new steps.

Table 6. Example of a low impacted test case.

...	...
step 1: operator presses the terminal approving button.	step 1: operator presses the terminal approving button.
step 2: system goes back to the terminal profiling screen.	step 2: system redirects the terminal to its profiling screen.
...	...

A total of 1477 MBT test cases were collected from TCOM's, where 333 were found *new* (23%), 724 *obsolete* (49%), and 420 *reusable* (28%). This data reinforces Silva et al. (2018)'s conclusions showing that, in an agile context, most of an MBT test suite became obsolete quite fast.

In a common scenario, all "obsolete" test cases (49%) would be discarded throughout the development cycles. To cope with this problem, we ran our automatic analysis and we reclassified the 724 obsolete test cases among *low impacted* – test cases that include unchanged steps and updated steps classified by our strategy as "low impact"; *highly impacted* – test cases that include unchanged steps and "high impact" steps; and *mixed*, test cases that include at least one "high impact" step and at least one "low impact" step.

From this analysis, 109 test cases were *low impacted*. Although this number seems low (15%), those test cases would be wrongly discarded when in fact they could be easily turned into reusable. For instance, Table 6 shows a simplified version of a "low impacted" test case from TCOM. As we can see, only step 2 was updated to better phrase a system response. This was an update for improving specification readability, but it does not have any impact on the system's behavior. We believe that *low impacted* test cases could be easily reused with little or no effort. In our case study, most of them need small updating that could be easily done in a single updating round, or even during test case execution. For the tester point of view, this kind of update may not be urgent and should not lead to a test case discard.

The remaining test cases were classified as follows: 196 "highly impacted" (27%), and 419 "mixed" (58%). Table 7 and 8 show examples of highly impacted and mixed tests,

Table 7. Example of a highly impacted test case.

... step 3: operator presses camera icon. step 4: system redirects to photo capture screen. ... step 9: operator takes a picture and presses the Back button. ...	... step 3: operator selects a testing plan. step 4: system redirects to the screen that shows the selected tests. ... step 9: operator sets a score and press Ok. ...

Table 8. Example of a mixed test case.

... step 2: operator presses button CANCEL to mark there is no occurrence description. ... step 7: operator presses the button SEND. ...	... step 2: operator presses the button CANCEL to mark there is no occurrence description. ... step 7: operator takes a picture of the hardware. ...

respectively. In Table 7, we can see that steps 3, 4, and 9 were drastically changed, which infer to a test case that requires much effort to turn it into reusable. On the other hand, in the test in Table 8, we have both an edit for fixing a typo (step 2) and an edit with a requirement change (step 7).

To check whether our classification was in fact effective we present its confusion matrix (Table 9). In general, our classification was 66% effective (Precision). A smaller precision was expected, when compared to the precision classification from Section 5, since here we consider all edits that might affect a test case, while in Section 5 we analyzed and classified each edit individually. However, we can see, our classification was highly effective for *low impacted* test cases, and most mistaken classification relates to *mixed* one (tests that combine low and high impact edits). Those were, in fact, test cases that were affected in a great deal by different types of use case editions.

Back to our strategy, we believe that *highly impacted* or *mixed* classifications indicate test cases that are likely to be discarded, since they refer to tests that would require much effort to be updated, while *low impacted* tests can be reused with little to no effort. Overall, our strategy correctly classified the test cases in 66% of the cases (Precision). Regarding *low impacted* tests, we correctly classified 63% of them. Therefore, from the 724 "obsolete" test cases, our strategy automatically inferred that 9.53% of them should not be discarded. We believe this rate can get higher when we better analyze the *mixed* set. A mixed test combines low and high impact edits. However, when we manually analyzed those cases, we found several examples where, although *high impact* edits were found, most test case impacts were related to *low impact* edits. For instance, there was a test case composed of 104 execution steps where only one of those steps needed revision due to a *high impact* use case edit, while the number of *low impact* edits was seven. In a practical scenario, although we still classify it as a *mixed* test case, we would say the impact of the edits was still quite small, which may indicate a manageable revision effort. Thus, we state that mixed tests need better analysis before discarding. The same approach may also work for *highly impacted* tests when related to a low number of edits.

Table 9. Confusion Matrix.

	Predicted			
	Low	High	Mixed	
Actual Low	69	3	37	109
Actual High	4	37	155	196
Actual Mixed	21	27	371	419
	94	67	563	724

Finally, we can answer RQ6 by saying that an automatic classification using distance functions can, in fact, reduce the number of discarded test cases by at least 9.53%. However, this rate tends to be higher when we consider *mixed* tests.

RQ6: Can distance function be used for reducing the discard of MBT tests? The use of distance functions can reduce the number of discarded test cases by at least 9.53%.

7 Combining Machine learning and Distance Values

In previous sections, we showed that distance functions alone could help to identify low impact edits that lead to test cases that can be updated with little effort. Moreover, the case study in Section 6 showed that this strategy could reduce test case discard by 9.53%. Although very promising, we believe those results could be improved, especially regarding the classification of *high impact* edits. In this sense, we propose a complementary strategy that combines distance values and machine learning.

To apply machine learning and avoid test case discard, we used Keras (Gulli & Pal, 2017). Keras is a high-level Python neural networks API that runs on top of TensorFlow (Abadi et al., 2016). It focuses on efficiency and productivity; therefore, it allows easy and fast prototyping. Moreover, it has a stronger adoption in both the industry and the research community (Géron, 2019).

Keras provides two types of models *Sequential*, and *Model with the functional API*. We opted to use a Sequential model due to its easy configuration and effective results. A sequential model is composed of a linear stack of layers. Each layer contains a series of nodes and performs calculations. A node is activated only when a certain threshold is achieved. In the context of our model, we used the REctified Linear Units (ReLU) and Softmax functions. Both are known to be a good fit for classification problems (Agarap, 2018). Dense layer nodes are connected to all nodes from the next layer, while Dropout layer nodes are more selective.

Our model classifies whether two versions of a given test case step refer to a *low* or *high impact* edit. Although techniques such as Word2Vec (Rong, 2014) could be used to transform step descriptions to numeric vectors, due to previous promising results (Sections 5 and 6), we opted to use a classification based on a combination of different distance values. Therefore, to be able to use the model, we first pre-process the input (versions of a test step), run the ten functions (Hamming (Hamming, 1950), LCS (Han et al., 2007), Cosine (Huang, 2008), Jaro (De Coster et al., 1), Jaro-Winkler (De Coster et al., 1), Jaccard (Lu et al., 2013), Ngram (Kondrak, 2005), Levenshtein (Levenshtein, 1966), OSA (Damerau, 1964), and Sorensen Dice (Sørensen, 1948)), and collect their distance values. Those values are then provided to our model that starts with a Dense layer, followed by four hidden layers, and returns as output a size two probability array O. O's first position refers to the found probability of a given edit be classified as *high impact*, while the second refers to the probability for a *low impact* edition. The highest of those two values will be the final classification of our model.

Suppose two versions of the test step *s* (*s* and *s'*). First, our strategy runs the ten distance functions considering the pair *(s; s')* and generates its input model set (e.g., $I = 0.67; 0.87; 0.45; 0.78; 0.34; 0.6; 0.5; 0.32; 0.7; 0.9$). This set is then provided to our model that generates the output array (e.g., $O = [0.5; 0.9]$). For this example, O indicates that the edits that transformed *s* to *s'* are *high impact*, with 50% chances, and *low impact*, with 90% chances. Therefore, our final classification is that the edits were *low impact* edit.

For training, we created a dataset with 78 instances of edits randomly collected from both SAFF and BZC projects. To avoid possible bias, we worked with a balanced training dataset (50% *low impact* and 50% *high impact* edits). Moreover, we reused the manual classification discussed in Sections 4 and 5 as reference answers for the model. In a Notebook Intel Core i3 with 4GB of RAM, the training phase was performed in less than 10 minutes.

7.1 Model Evaluation

Similar to the investigation described in Sections 5 and 6, we proceeded an investigation to validate whether the strategy that combines machine learning with distance values is effective and promotes the reduction of test case discard. For that, we set the following research questions:

- **RQ7**: Can the combination of machine learning and distance values improve the classification of edits' impact in use case documents?
- **RQ8**: Can the combination of machine learning and distance values reduce the discard of MBT tests?

To answer RQ7, and evaluate our model, we first ran it against two data sets: (i) the model edits combined of the SAFF and BZC projects (Table 1); and (ii) the model edits collected from the TCOM project (Table 4). While the first provides a more comprehensive set, the second allows us to test the model in a whole new scenario. It is important to highlight that both data sets contain only real model edits performed by the teams. Moreover, they contains both *low* and *high impact* edits. Again, we reused our manual classification to validate the model's output. Table 10 presents the results of this evaluation. As we can see, our strategy performed well for predicting the edits impact, especially for TCOM, it provided an accuracy of 95%. These results give us evidence of our model efficiency.

Table 10. Results of our model evaluation using TCOM's dataset.

	Precision	**Recall**	**Accuracy**
SAFF+BZC	81%	97%	80%
TCOM	94%	99%	95%

RQ7: Can the combination of machine learning and distance values improve the classification of edits' impact in use case documents? Our strategy was able to classify edits with accuracy above 80%, an improvement of 7% when compared to the classification using only distance functions. This result reflects its efficiency.

To answer RQ8, we considered TCOM's MBT test cases generated from its CLARET files. We reused the manual classification from Section 6, and ran our strategy to automatic reclassify the 724 obsolete test cases among *low impacted* – test cases that include unchanged steps and updated steps classified by our strategy as "low impact"; *highly impacted* – test cases that include unchanged steps and "high impact" steps; and *mixed*, test cases that include at least one "high impact" step and at least one "low impact" step.

From the 109 actual *low impacted* test cases, our strategy was able to detect 75 (69%), an increase of 6% when compared to the classification using a single distance function. Those would be test cases that should be easily revised to avoid discarding as model changes were minimal (Figure 6). Table 9 presents the confusion matrix for our model classification. Out of the 724 obsolete test cases (according to Oliveira et al.'s classification (Oliveira Neto et al., 2016)), our model would help a tester to automatically save 10.4% from discarding.

As we can see, overall, our classification was 69% effective, an increase of 3% when compared to the classification using a single distance function (Table 9). Although this improvement may be low, it is important to remember that those would be the actual test that would be saved from a wrong discard.

On the other hand, we can see a great improvement in the *high impact* classification (from 19% to 86%). This data indicates that different from the strategy using a single distance function, our model can be a great help to automatically identify both reusable and in fact obsolete test cases. On the other hand, the classification for *mixed* test cases performed worse (from 88% to 61%). However, we believed that mixed test cases are the ones that require a manual inspection to check whether it is worth updating for reusing or should be discarded.

It is important to highlight that our combined strategy was able to improve the performance rates for the most important classifications (*low* and *high impacted*), which are related to major practical decisions (to discard or not a test case). Moreover, when wrongly classifying a test case, our model often sets it as *mixed*, which we recommend a manual inspection. Therefore, our automatic classification tends to be accurate and not misleading.

Finally, we can answer RQ8 by saying that the combined strategy was, in fact, effective for reducing the discard of MBT tests. The rate of saved tests was 10.4%. Moreover, it improved, when compared to the strategy using a single distance function, the detection rate by 6% for *low impacted* test cases, and by 67% for *high impact* ones.

RQ8: Can the combination of machine learning and distance values reduce the discard of MBT tests? Our combined strategy helped us to reduce the discard of test cases by 10.4%, an increase of 0.9%. However, it correctly identifies test cases that should in fact be discarded.

8 General Discussion

In previous sections, we proposed two different strategies for predicting the impact of model edits and avoiding test case discarding: (i) a pure distance function-based; and (ii) a strategy that combines machine learning with distance values. Both were evaluated in a case study with real data. The first strategy applies a simpler analysis, which may infer lower costs. Though simple, it was able to correctly identify 63% of the *low impacted* test cases, and to rescue 9.53% of the test cases that would be discarded. However, it did not perform well when classifying *highly impacted* tests (19%). Our second approach, though more complex (it requires a set of distance values as inputs to the model), generated better results for classifying *low impacted* and *highly impacted* test cases, 68%, and 86% precision, respectively. Moreover, it helped us to avoid the discard of 10.4% of the test cases. Therefore, if running several distance functions for each model edit is not an issue, we recommend the use of (ii) since it is in fact the best option for automatically classify test cases that should be reused (*low impacted*) or be discarded (highly impacted). Moreover, regarding time, our prediction model responses were almost instant. Regarding *mixed* tests, our suggestion is always to inspect them to decide whether it is worth updating.

9 Threats to Validity

Most of the threats for validity to the drew conclusions refer to the number of projects, use cases, and test cases used in our empirical studies. Those numbers were limited to the artifacts created in the context of the selected projects. Therefore, our results cannot be generalized beyond the three projects (SAFF, BZC, and TCOM). However, it is important to highlight that all used artifacts are from real industrial systems from different contexts.

As for conclusion validity, our studies deal with a limited data set. Again, since we chose to work with real, instead of artificial artifacts, the data available for analysis were limited. However, the data was validated by the team engineers and by the authors.

One may argue that since our study deals only with CLARET use cases and test cases, our results are not valid for other notations. However, CLARET resembles traditional specification formats (e.g., UML Use Cases). Moreover, CLARET test cases are basically a sequence of pairs of steps (user input - system response), which can relate to most manual testing at the system level.

Regarding internal validity, we collected the changed set from the project's repositories, and we manually classify each change according to its impact. This manual validation was performed by at least two of the authors and, when needed, the project's members were consulted. Moreover, we reused open-source implementations of the distance functions[5]. These implementations were also validated by the first author.

10 Related Work

The practical gains of regression testing are widely discussed (e.g., (Aiken et al., 1991; Leung & White, 1989; Wong et al., 1997; Ekelund & Engström, 2015)). In the context of agile development, this testing strategy plays an important role by working as safety nets changes are performed (Martin, 2002). Parsons et al. (2014) investigate regression testing strategies in agile development teams and identify factors that can influence the adoption and implementation of this practice. They found that the investment in automated regression testing is positive, and tools and processes are likely to be beneficial for organizations. Our strategies (distance functions, distance functions and machine learning) are automatic ways to enable the preservation of regression test cases.

Ali et al. (2019) propose a test case prioritization and selection approach for improving regression testing in agile projects. Their approach prioritizes test cases by clustering the ones that frequently change. Here, we see a clear example of the importance of preserving test cases.

Some work relates agile development to Model-Based Testing, demonstrating the general interest in these topics. Katara & Kervinen (2006) introduce an approach to generate tests from use cases. Tests are translated into sequences of events called action-words. This strategy requires an expert to design the test models. Puolitaival (2008) present a study on the applicability of MBT in agile development. They refer to the need for technical practitioners when performing MBT activities and specific adaptations. Katara & Kervinen (2006) discuss how MBT can support agile development. For that, they emphasize the need for automation aiming that MBT artifacts can be manageable and with little effort to apply.

Cartaxo et al. (2008) propose a strategy/tool for generating test cases from ALTS models and selecting different paths. Since the ALTS models reflect use cases written in natural language, the generated suites encompass the problems evidenced in our study (a great number of obsolete test cases), as the model evolves.

Oliveira Neto et al. (2016) discuss a series of problems related to keeping MBT suites updated during software evolution. To cope with this problem, they propose a test selection approach that uses test case similarity as input when collecting test cases that focus on recently applied changes. Oliveira Neto et al.'s approach refers to obsolete all test cases that are impacted in any way by edits in the requirement model. However, as our study found, a great part of those tests can be little

[5] https://github.com/luozhouyang/python-string-similarity

impacted and could be easily reused, avoiding the discard of testing artifacts.

The test case discard problem is not restricted to CLARET artifacts. Other similar cases are discussed in the literature (e.g.,(Oliveira Neto et al., 2016; Nogueira et al., 2007)). Moreover, this problem is even greater with MBT test cases derived from artifacts that use non-controlled language (Pinto et al., 2012).

Other works also deal with test case evolution (e.g., (Katara & Kervinen, 2006; Pinto et al., 2012)). They discuss the problem and/or propose strategies for updating the testing code. Those strategies do not apply to our context, as we work with MBT test suite evolution generated from use case models.

Distance functions have been used in different software engineering scenarios (e.g., (Runkler & Bezdek, 2000; Okuda et al., 1976; Lubis et al., 2018)). For instance, Runkler & Bezdek (2000) use the Levenshtein function to automatically extract keywords from documents. In the context of MBT, Coutinho et al. (2016) investigated the effectiveness of a series of distance functions when used combined with strategies for suite reduction based on similarity. Although in a different context, their results go according to ours where all distance functions performed in a similar way.

The use of machine learning techniques in software engineering is not new. Baskeles et al. (2007) propose a model for estimating development effort aiming at overcoming problems related to budget and schedule extension. Gondra (2008) uses an artificial neural network to determine the importance of software metrics for predicting fault-proneness.

Durelli et al. (2019) present a systematic mapping study on machine learning applied to software testing. From 48 selected primary studies, they found that machine learning has been used mainly for test-case generation, refinement, and evaluation. For instance, Strug & Strug (2012) use a KNN-learner to reduce the set of mutants to be executed in mutation testing. It predicts when a test can kill certain mutants. Fraser & Walkinshaw (2015) propose an approach based on machine learning algorithms to evaluate test suites using behavioral coverage. It receives data from a test generation tool and predicts the behavior of the program for the given inputs. Zhu et al. (2008) propose a model for estimating test execution efforts based on testing data such as the number of test cases, test complexity, and knowledge of the system under testing. Chen et al. (2011) present a machine learning approach for selection regression test cases. Their learner clusters similar test cases based on an input function and constraints. Our work differs from the others since it uses machine learning and distance functions to predict the impact of a given use case update and to avoid the discard of MBT test cases.

11 Concluding Remarks

In this paper, we describe a series of empirical studies ran on industrial systems for evaluating the use of distance functions to classify the impact of edits in use case files automatically. Our results showed that distance functions are effective in identifying low impact editions. Therefore, we propose two variations of its use: as a classification strategy itself (Section 5), and combined with a machine learning model (Section 7).

We also found that low impact editions often refer to test cases that can be easily updated without any effort. Our strategies helped to both identify low impact and high impact test cases. We believe those results can help testers to better work with MBT artifacts in the context of software evolution and avoid the discard of test cases.

As future work, we plan to expand our study with a broader set of systems. We also consider developing a tool that, using distance functions, can help testers to identify and update low impact test cases. Finally, we plan to investigate the use of different approaches (e.g., other machine learning techniques, dictionaries) to improve our classification rates and better help testers when updating highly impacted test cases.

Acknowledgements

This research was partially supported by a cooperation between UFCG and two companies Viceri Solution LTDA and Ingenico do Brasil LTDA, the latter stimulated by the Brazilian Informatics Law n. 8.248, 1991. Second and fourth authors are supported by National Council for Scientific and Technological Development (CNPq)/Brazil (processes 429250/2018-5 and 315057/2018-1). First and third authors were supported by UFCG/CNPq and CAPES, respectively.

References

Abadi M., et al., 2016, in 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). pp 265–283

Agarap A. F., 2018, arXiv preprint arXiv:1803.08375

Aiken L. S., West S. G., Reno R. R., 1991, Multiple regression: Testing and interpreting interactions. Sage

Ali S., Hafeez Y., Hussain S., Yang S., 2019, Software Quality Journal, pp 1–27

Anderson J., Salem S., Do H., 2014, in Proceedings of the 11th Working Conference on Mining Software Repositories. pp 142–151

Baskeles B., Turhan B., Bener A., 2007, in 2007 22nd international symposium on computer and information sciences. pp 1–6

Beck K., Gamma E., 2000, Extreme programming explained: embrace change. addison-wesley professional

Bouquet F., Grandpierre C., Legeard B., Peureux F., Vacelet N., Utting M., 2007, in Proceedings of the 3rd international workshop on Advances in model-based testing. pp 95–104

Cai L., Tong W., Liu Z., Zhang J., 2009, in 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing. pp 103–108

Cartaxo E. G., Andrade W. L., Neto F. G. O., Machado P. D., 2008, in Proceedings of the 2008 ACM symposium on Applied computing. pp 1540–1544

Chen S., Chen Z., Zhao Z., Xu B., Feng Y., 2011, in 2011

Fourth IEEE International Conference on Software Testing, Verification and Validation. pp 1–10

Cohen W. W., Ravikumar P., Fienberg S. E., et al., 2003, in IIWeb. pp 73–78

Coutinho A. E. V. B., Cartaxo E. G., de Lima Machado P. D., 2016, Software Quality Journal, 24, 407

Dalal S. R., Jain A., Karunanithi N., Leaton J. M., Lott C. M., Patton G. C., Horowitz B. M., 1999, in Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002). pp 285–294, doi:10.1145/302405.302640

Damerau F. J., 1964, Commun. ACM, 7, 171

De Coster X., De Groote C., Destiné A., Deville P., Lamouline L., Leruitte T., Nuttin V., 1

Diniz T., Alves E. L., Silva A. G., Andrade W. L., 2019, in Proceedings of the XXXIII Brazilian Symposium on Software Engineering. pp 337–346

Durelli V. H., Durelli R. S., Borges S. S., Endo A. T., Eler M. M., Dias D. R., Guimaraes M. P., 2019, IEEE Transactions on Reliability, 68, 1189

Ekelund E. D., Engström E., 2015, in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp 449–457

Elish K. O., Elish M. O., 2008, Journal of Systems and Software, 81, 649

Frakes W., 1994, in Proceedings of 1994 3rd International Conference on Software Reuse. pp 2–3

Fraser G., Walkinshaw N., 2015, Software Testing, Verification and Reliability, 25, 749

Géron A., 2019, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. O'Reilly Media

Gondra I., 2008, Journal of Systems and Software, 81, 186

Gulli A., Pal S., 2017, Deep learning with Keras. Packt Publishing Ltd

Hamming R. W., 1950, The Bell system technical journal, 29, 147

Han T. S., Ko S.-K., Kang J., 2007, in International Workshop on Machine Learning and Data Mining in Pattern Recognition. pp 585–600

Harrold M. J., 2000, in Proceedings of the Conference on the Future of Software Engineering. pp 61–72

Hayes J. H., Dekhtyar A., Sundaram S., 2005, in ACM SIGSOFT Software Engineering Notes. pp 1–5

He Z., Shu F., Yang Y., Li M., Wang Q., 2012, Automated Software Engineering, 19, 167

Huang A., 2008, in Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand. pp 9–56

Itkonen J., Mantyla M. V., Lassenius C., 2009, in 2009 3rd International Symposium on Empirical Software Engineering and Measurement. pp 494–497, doi:10.1109/ESEM.2009.5314240

Katara M., Kervinen A., 2006, in Haifa Verification Conference. pp 219–234

Kondrak G., 2005, in Consens M. P., Navarro G., eds, Lecture Notes in Computer Science Vol. 3772, SPIRE. Springer, pp 115–126, `http://dblp.uni-trier.de/db/conf/spire/spire2005.html#Kondrak05`

Kruskal J. B., 1983, SIAM review, 25, 201

Kumar D., Mishra K., 2016, Procedia Computer Science, 79, 8

Leung H. K., White L., 1989, in Proceedings. Conference on Software Maintenance-1989. pp 60–69

Levenshtein V. I., 1966, Soviet Physics Doklady, 10, 707

Lu J., Lin C., Wang W., Li C., Wang H., 2013. pp 373–384, doi:10.1145/2463676.2465313

Lubis A. H., Ikhwan A., Kan P. L. E., 2018, International Journal of Engineering & Technology, 7, 17

Malhotra R., Jain A., 2012, Journal of Information Processing Systems, 8, 241

Martin R. C., 2002, Agile software development: principles, patterns, and practices. Prentice Hall

Michie D., Spiegelhalter D. J., Taylor C., et al., 1994, Neural and Statistical Classification, 13, 1

Myers G. J., Sandler C., Badgett T., 2011, The art of software testing. John Wiley & Sons

N. Jorge D., Machado P., L. G. Alves E., Andrade W., 2017, in Proceedings of the 24th Tools Session / 8th Brazilian Conference on Software: Theory and Practice. , doi:10.1109/RE.2018.00041

N. Jorge D., Machado P., L. G. Alves E., Andrade W., 2018. pp 336–346, doi:10.1109/RE.2018.00041

Nagappan N., Murphy B., Basili V., 2008, in 2008 ACM/IEEE 30th International Conference on Software Engineering. pp 521–530

Nogueira S., Cartaxo E., Torres D., Aranha E., Marques R., 2007, in 1st Brazilian Workshop on Systematic and Automated Software Testing.

Noor T. B., Hemmati H., 2015, in 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). pp 58–68

Okuda T., Tanaka E., Kasai T., 1976, IEEE Transactions on Computers, 100, 172

Oliveira Neto F. G., Torkar R., Machado P. D., 2016, Information and Software Technology, 80, 124

Parsons D., Susnjak T., Lange M., 2014, Software Quality Journal, 22, 717

Pinto L. S., Sinha S., Orso A., 2012, in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. p. 33

Pressman R., 2005, Software Engineering: A Practitioner's Approach, 6 edn. McGraw-Hill, Inc., New York, NY, USA

Puolitaival O.-P., 2008, Adapting model-based testing to agile context: Master's thesis. VTT Technical Research Centre of Finland

Rong X., 2014, arXiv preprint arXiv:1411.2738

Runkler T. A., Bezdek J. C., 2000, in Ninth IEEE International Conference on Fuzzy Systems. FUZZ-IEEE 2000 (Cat. No. 00CH37063). pp 636–640

Sedano T., Ralph P., Péraire C., 2017, in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp 130–140

Shepperd M., Bowes D., Hall T., 2014, IEEE Transactions on Software Engineering, 40, 603

Silva A. G., Andrade W. L., Alves E. L., 2018, in Proceedings of the III Brazilian Symposium on Systematic and Automated Software Testing. SAST '18. ACM, New York, NY, USA, pp 49–56, doi:10.1145/3266003.3266009, `http://doi.acm.org/10.1145/3266003.3266009`

Sørensen T., 1948, Biol. Skr., 5, 1

Srinivasan K., Fisher D., 1995, IEEE Transactions on Software Engineering, 21, 126

Strug J., Strug B., 2012, in IFIP International Conference on Testing Software and Systems. pp 200–214

Sutherland J., Sutherland J., 2014, Scrum: the art of doing twice the work in half the time. Currency

Tretmans J., 2008, in , Formal methods and testing. Springer, pp 1–38

Utting M., Legeard B., 2007, Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Utting M., Pretschner A., Legeard B., 2012, Software Testing, Verification and Reliability, 22, 297

Von Mayrhauser A., Mraz R., Walls J., Ocken P., 1994, in Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors. pp 484–491

Wong W. E., Horgan J. R., London S., Agrawal H., 1997, in PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering. pp 264–274

Zhang D., Tsai J. J., 2003, Software Quality Journal, 11, 87

Zhu X., Zhou B., Hou L., Chen J., Chen L., 2008, in 2008 The 9th International Conference for Young Computer Scientists. pp 1193–1198