**MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN DATA SCIENCE**

# Improving automated unit test generation for machine learning libraries using structured input data

BERG, Lucas

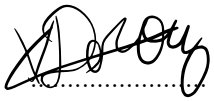
*Award date:*
2024

*Awarding institution:*
University of Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2023-2024

# Improving automated unit test generation for machine learning libraries using structured input data

Lucas Berg


..... (Signature pour approbation du dépôt - REE art. 40)

Promoteur : Xavier Devroey

Mémoire présenté en vue de l'obtention du grade de Master 120 en Sciences Informatiques à finalité spécialisée en Data Science

Faculté d'Informatique – Université de Namur

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

# Résumé

Le domaine de la génération automatique de test s'est considérablement développé ces dernières années pour réduire les coûts des tests logiciels et pour trouver des bugs. Cependant, les techniques de génération automatique de tests pour les librairies de machine learning produisent encore des tests de faible qualité et les articles sur le sujet ont tendance à travailler en Java, alors que la communauté du machine learning a tendance à travailler en Python. Certains articles ont tenté d'expliquer les causes de ces tests de mauvaise qualité et de rendre possible la génération automatique de tests en Python mais ils sont encore assez récents et donc aucune étude n'a encore essayé d'améliorer ces tests en Python. Dans ce mémoire, nous allons introduire 2 améliorations pour Pynguin, un outil de génération automatique de tests pour Python, afin de générer de meilleurs tests pour les librairies de machine learning en utilisant des inputs structurées et de mieux gérer les crashs provenant des modules d'extension C. Sur base d'un ensemble de 7 modules, nous montrerons que notre approche a permis de couvrir des lignes de code inaccessibles avec l'approche traditionnelle et de générer des tests révélant des erreurs. Nous espérons que notre approche servira de point de départ pour intégrer plus facilement les connaissances des testeurs sur les inputs des programmes dans les outils de génération automatique de tests, et pour créer des outils qui trouveront davantage de bugs causant des crashs.



***Mots-clés**: Testing machine learning libraries, Search-based software testing, Test case generation, Grammar-based fuzzing, Crash detection*



# Abstract

The field of automated test case generation has grown considerably in recent years to reduce software testing costs and find bugs. However, the techniques for automatically generating test cases for machine learning libraries still produce low-quality tests and papers on the subject tend to work in Java, whereas the machine learning community tends to work in Python. Some papers have attempted to explain the causes of these poor-quality tests and to make it possible to generate tests in Python automatically, but they are still fairly recent, and therefore, no study has yet attempted to improve these test cases in Python. In this thesis, we introduce 2 improvements for Pynguin, an automated test case generation tool for Python, to generate better test cases for machine learning libraries using structured input data and to manage better crashes from C-extension modules. Based on a set of 7 modules, we will show that our approach has made it possible to cover lines of code unreachable with the traditional approach and to generate error-revealing test cases. We expect our approach to serve as a starting point for integrating testers' knowledge of input data of programs more easily into automated test case generation tools and creating tools to find more bugs that cause crashes.



***Keywords**: Testing machine learning libraries, Search-based software testing, Test case generation, Grammar-based fuzzing, Crash detection*

# Acknowledgements

I would like to thank everyone who helped me complete this master's thesis.

First of all, I want to express my gratitude to Xavier Devroey, my promoter, who helped me throughout the process of writing this master's thesis and allowed me to do a research internship at the Delft University of Technology.

Second, I want to express my thanks to Annibale Panichella, who welcomed me to TU Delft, was my internship supervisor, and also gave me precious advice during the creation of this master's thesis.

Finally, I would like to thank the rest of the SERG group at TU Delft and my friends, with whom I shared the problems I encountered and who helped me think of new solutions.

# Contents

# Acronyms

**CTC** Crash test count. 33, 37, 42

**FPE** Floating-point exception. 24, 25, 33, 37, 41, 42, 55

**GIL** Global Interpreter Lock. 23, 27

**MuT** Module under Test. 17, 18, 23, 27, 28, 35, 36, 41, 50, 55, 58

**OOM** Out of memory. 24, 25, 33, 37, 41, 42

**PCW** Parameter concrete weight. 21, 31, 46, 47, 49, 50

**PW** Parameter weight. 21, 31, 46, 47, 49, 50

**Segfault** Segmentation fault. 24, 25, 33, 37, 41, 42, 55

**SOP** Skip optional parameter. 21, 31, 42–45, 49, 50, 58

**SuT** Software under Test. 3–6

**T/O** Timeout. 24, 25, 33, 37, 42

# Chapter 1

# Introduction

Software testing has long been considered an essential step in the software development cycle [1] as it is the step that ensures software quality and limits the number of bugs. This step is generally considered to be one of the most expensive steps, as manual test execution and manual test case creation are time-consuming processes for testers. To limit these costs, many researchers have developed various automated test case generation techniques [2, 3, 4]. These techniques have mainly been developed in Java because it is a widely used language in research and business. However, specific fields of computer science rarely use Java, which means that the tools created for this language are not necessarily usable or are not adapted to the code of these fields. This is the case in machine learning, which mainly uses Python to create prototypes and train models. Since machine learning has been one of the most popular subjects in recent years with the emergence of LLMs, it is crucial to properly test the libraries used to create them while limiting the costs involved.

## 1.1 Problem description

In the last few years, researchers in software testing have started to develop tools to automatically generate test cases for Python [4, 5]. However, a paper has shown that the code coverage computed from the tests generated by these tools is relatively low for machine learning libraries [6]. This problem has already been identified for this type of library in Java, but for now, no study has attempted to reproduce the results in Python, where these libraries are most common [7]. In addition, this paper proposed specific solutions, such as the use of structured input, which could potentially improve the code coverage of the generated tests, but these solutions have never been tested in the context of machine learning libraries. One paper has tried to use structured inputs by combining approaches of automated test case generation with techniques of grammar-based fuzzing, but this was in the context of JSON file parsers [8]. As a result, there is, for now, no efficient way of generating test cases for machine learning libraries in Python.

## 1.2 Research goal

Based on the problems mentioned in the previous section, we have decided to focus on two main goals:

- Replicate the results found in Java but in Python
- Use structured input data to see if this has a positive impact on automated test case generation for machine learning libraries in Python

These goals are intended to bridge the gap between what has already been found in Java and improve automated test case generation for machine learning libraries in Python. To replicate the results from Java as closely as possible, we decided to use Pynguin [4], a search-based automated test case generation tool that is more or less the equivalent of Evosuite [3], which to our knowledge is the best search-based automated test case generation tool for Java.

## 1.3 Contributions

To achieve our objectives, we have made the following contributions:

- The creation of a plugin system that allows testers to generate inputs suited to their needs
- The implementation of a few plugins to generate structured data to test our plugin system on some modules
- The modification of the architecture of Pynguin to better support test case generation in Python and to create error-revealing test cases
- The release of our results [9] and a replication package [10] to replicate them

## 1.4 Outline

Our study is divided into nine chapters: We will start by explaining the background and related tools linked to our research objectives in Chapter 2. Next, we will talk about our motivation in Chapter 3, propose a methodology in Chapter 4 and present our results in Chapter 5. After that, we will present a second methodology in Chapter 6 aimed at fixing the problems of the first methodology, present our new results in Chapter 7 and discuss them in Chapter 8. Finally, we will conclude in Chapter 9.

# Chapter 2

# Background and related work

In this chapter, we will examine the importance of software testing, the techniques used to automate it, and various terms that will be used later. We will also explore the fields of automated test case generation, machine learning program testing, and grammar-based fuzzing to place this research in the state of the art and show its benefits.

## 2.1 Software testing

For over 50 years, software testing has been considered an essential and very costly step in the software development life cycle [11]. A common definition of this term is that it is a process that ensures that a software product does what it is supposed to and not anything unexpected. For Myers et al. [1], who wrote "The Art of Software Testing", a classic book about software testing, it is a bad definition because it does not emphasise enough the fact that tests are performed to add value to a program by improving software quality and finding errors. According to them, a more correct definition would be "Testing is the process of executing a program with the intent of finding errors." [1] Ensuring that tests are effective and able to detect errors is a real challenge nowadays, as software products are becoming increasingly complex. On top of this, the impact of errors in code has never been so important, as the cost of software failures in operational systems is higher than ever, reaching 1.56 trillion dollars in the US in 2020 for reference [12]. That is why, over the years, researchers have carried out numerous studies on these subjects and have created definitions, methods, metrics and tools to improve software testing.

### 2.1.1 Software testing types

Firstly, software testing can be divided into two main types: manual and automated. Manual testing is a type of testing that requires the active participation of a human being, whereas automatic testing uses separate code to execute and check that the Software under Test produces the expected results.

#### Manual testing

Developers generally use manual testing techniques when they find a bug during programming. They are considered manual because they require a human oracle to verify that the code works as intended.

The simplest form of this type of testing is when a human executes some code and then checks that there are no errors and that the code works as expected. This type of testing is often associated with dynamic testing or even dynamic program analysis and is usually the only solution considered for manual tests. However, this is not the only way to perform manual testing.

Although researchers do not always consider this as software testing, [11] include code reading as testing because it is also a technique that can find bugs. This type of testing is associated with static testing or even static program analysis and can be performed in several different ways. The main method, "desk checking", relies on developers re-reading their code to find errors. However,

this method is unreliable, so inspections and walkthroughs have been proposed to get a group of testers to review the code to find more bugs.

#### Automated testing

Automated testing techniques are widely used nowadays, as they allow to check that a software product produces expected results with the use of specified oracles and without requiring the active participation of a human. These techniques rely on tools or scripts to run the Software under Test in different scenarios or with different inputs and to analyse its behaviour.

Automated testing techniques mainly include the generation of test cases that can be executed automatically, but also other techniques that come from random testing. In automated testing, a test case usually consists of a program that executes part of the Software under Test and checks that the results are the expected ones using assertions. These test cases can be created manually by testers, as is often the case in companies, or generated using some algorithms, as often done in research. However, another test case definition is sometimes used in random testing and fuzzing fields. In these fields, the concept of input is sometimes called test case or fuzz. This test case type is much more general because it can be random data, image binary, JSON-like data, and some sequence of function calls like in automated test case generation.

There is also a last kind of automated testing. If it were possible to consider code reading as a software testing technique in the manual testing section, their automated version would be static code analysis tools since they can detect errors. Although these tools are not often considered in software testing, they can sometimes be as effective as test cases in detecting common errors.

### 2.1.2 Software testing levels

Software testing is usually divided into several levels corresponding to the different levels of abstraction that will be the test subjects [13].

#### Unit testing

First, there is unit testing, which involves testing a component, module, or function in isolation to verify that it is working properly.

#### Integration testing

Secondly, there is integration testing, which aims to test a set of components to ensure that they work well together.

#### System testing

Finally, there is system testing, which aims to test the system as a whole to check that it is operating correctly from an external point of view.

### 2.1.3 Software testing methods

Software testing uses different methods corresponding to the degree of knowledge on the internal structure of the Software under Test.

#### White-box testing

White-box testing is a method for testing a software product's internal structure. It often relies on code instrumentation and software metrics to check the software's implementation details.

#### Black-box testing

Black-box testing is a method that tests a software product without any other information than the software specification.

#### Grey-box testing

Grey-box testing is a mix between white-box and black-box testing. This method does not have access to the software product's code or code instrumentation, but it does have access to the documentation of the algorithms and the internal structures used, which could help find common bugs for these algorithms or internal structures.

### 2.1.4 Software metrics for software testing

Although software testing can be used to find bugs or faults, it is still a complex process. The fact is that even if there is a bug or fault in the code, the right tests still need to be performed to find it. That is why, in an attempt to limit the number of bugs or faults and improve software quality, numerous metrics have been proposed. Some of these metrics aim to highlight certain parts of the code that are more likely to contain errors. These metrics are strongly linked to software testing, as they often use the results of performed tests to be created, and they are used to guide software testing [14]. Among the metrics often used to guide software testing are code coverage, mutation score and cyclomatic complexity.

#### Code coverage

Code coverage measures to which degree the code of a software product is executed when a test suite is run. This metric is most frequently used in association with test case generation, as it can guide the creation of new test cases to have a test suite that can execute as much code as possible without human intervention. Moreover, in a previous study, it was also shown that there is a medium to strong correlation between the effectiveness of an automated test suite and its code coverage [15]. To be computed, this metric essentially uses code instrumentation to obtain runtime information on the executed code and a coverage criteria. The best-known criteria include statement coverage, which determines which statements have been executed and branch coverage, which determines if each branch on a control structure has been executed. Usually, branch coverage is preferred to statement coverage because branch coverage is a superset of code coverage, and the additional instrumentation cost is not that high.

#### Mutation score

The mutation score is a metric used in the context of mutation testing. It measures the number of mutants that a test suites can kill out of all the mutants created. A mutant is just a modification to the Software under Test; it can be a substitution of an addition by a subtraction or a modification of the content of a string, for example. A mutant is considered killed when there is a test that fails when run on it.

#### Cyclomatic complexity

Cyclomatic complexity measures a program's complexity from its source code. It is calculated based on the number of unique paths that can be followed during program execution and tries to highlight code that would be complicated to test properly and where bugs could more easily occur. This metric is usually calculated by static code analysis tools, which report the complexity of a given code to the person using it.

## 2.2 Automated test case generation

Automated test case generation is the process of writing code that can generate test cases for a targeted source code. This is a field of research in which numerous solutions have been proposed because the manual generation of test cases is a task that requires a huge amount of time and effort [1]. To compare these different solutions, the researchers decided to use certain software metrics such as code coverage and mutation score, as these have been shown to demonstrate the effectiveness of a test suite [15].

### 2.2.1 Random-based approaches

Random-based approaches usually use random testing techniques to generate sequences of random function calls and then create test cases with these sequences [2]. These test cases are then executed,

and their results are checked for exceptions or faults. As explained earlier, random testing does not aim to create test cases in the traditional sense. However, some tools will bridge the gap between random testing test cases, which are just another name for inputs, and traditional test cases. To achieve this, they often allow you to export the inputs from random testing techniques into unit testing framework test cases.

In addition, random-based approaches are often used as a baseline to evaluate new techniques [16]. Due to their fairly simple random-based behaviour, these algorithms usually work the least well, so it is easy to see whether a technique does better than random or not.

### 2.2.2 Symbolic-based approaches

Symbolic-based approaches use symbolic execution to determine which part of the program will be executed based on which input. To achieve this, symbolic values are used instead of concrete ones to generate a set of constraints using the different program assignments and conditions. These constraints can then be solved into a set of concrete values used as inputs by the Software under Test, usually using an SMT solver. Finally, these sets of inputs are used to create test cases.

However, symbolic execution has several limitations because the number of constraints to solve explodes when the program's size increases [17]. These problems can sometimes be corrected by using heuristics [18] or parallelization [19] or by grouping similar solutions [20].

### 2.2.3 Concolic-based approaches

Concolic-based approaches combine concrete and symbolic approaches. They achieve this by first executing the program with arbitrary concrete values and instrumentation, then re-executing the program symbolically, taking into account the information gathered by the instrumentation, and finally repeating the algorithm with new concrete values calculated from the symbolic execution.

### 2.2.4 Search-based approaches

Search-based approaches use search optimisation techniques such as genetic algorithms [21, 22, 23] to optimize some objectives and to generate test cases. The most commonly used objectives are branch objectives [22, 24], where an objective is set for each branch in the code and where the branch distance is used to determine how close an executed test case is to the objectives. Only one or several objectives will be optimised simultaneously, depending on the algorithms used.

Modern search-based techniques begin by generating a random population of test cases. Then, the test cases are run with instrumentation related to the chosen objectives. Finally, either all the objectives are achieved, or a new population of test cases is generated using an algorithm such as a genetic algorithm, and the search-based algorithm repeats its previous operations.

These approaches of automated test generation using search optimization techniques are part of a branch of software testing called search-based software testing. This branch of software testing has long been studied by researchers [22], and the literature has already shown that search-based approaches to automated test generation work at the unit [3], integration [25] and system levels [26]. In addition, several algorithms such as WholeSuite [27], MIO [28], MOSA [29] and DynaMosa [30] have been created to try and improve the search for objectives. These algorithms have been implemented in several tools in different languages such as Java [3], Python [31, 4] or Javascript [32].

### 2.2.5 Existing tools

Various automated test generation tools exist. The tools that will be presented here have been selected among those participating in the SBFT Tool Competition 2023 for Java [16] and in the SBFT Tool Competition 2024 for Python [6].

#### Evosuite

Evosuite is an automated search-based unit test generation tool for Java introduced by Fraser and Arcuri [3]. It is, to the best of our knowledge, the most powerful automated search-based test case generation tool available today based on the code coverage it achieves on a large number of Java

projects and its results in the SBFT tool competition 2023 in Java [16]. It was created to introduce a new approach to automated unit test case generation called "Whole test suite generation", which attempts to optimize an entire coverage criterion instead of a single objective [33, 27]. Furthermore, Evosuite was designed to be able to run on Java bytecode from the beginning. This means that the instrumentation offered by Evosuite is much more precise than instrumentation done at the source code level, as some Java structures, such as switch statements, are converted to if statements in the bytecode. In their paper, Fraser and Arcuri [33] showed that their "Whole test suite generation" approach with the "Branch coverage" criteria significantly improved code coverage compared with the classic "Single branch" approach, which was also implemented in Evosuite to compare it with the new approach in a benchmark.

Over the years, Evosuite has become one of the foundations on which many new studies have been based because it already has all the components needed to compare the new approaches with the old ones and to build new tools. Among the new approaches that have been integrated into Evosuite, there are MOSA [29], MIO [28] and DynaMOSA [30], the approach that offers the best performance on Evosuite today.

#### Randoop

Randoop is one of the first automated unit test generation tools for Java [2]. It was introduced in a paper by Pacheco and Ernst [2] and uses feedback-directed random test generation to generate a sequence of method calls and constructors to create test cases. This tool can generate 2 types of test cases: regression test cases and error-revealing test cases. Regression tests are the traditional test cases created by automated test generation tools. They can not find an error immediately but may be able to find one in the future if a bug is introduced into the code. Error-revealing test cases highlight bugs in the code by comparing calls to certain methods with their contract/specification, such as checking the symmetry property of "Object.equals".

Due to its rather simple and pseudo-random approach, Randoop has more or less become the baseline against which new approaches would be compared, as it allows us to see if they are not worse than just random. This is also why Randoop has the lowest code and branch coverage in the SBFT tool competition 2023 for Java [16].

#### UTBot

UTBot is an automated unit test generation tool based on symbolic execution and an SMT solver and can generate test cases in several languages, including Java, Python and many others [34]. It has been developed by Huawei and was introduced in a paper by Ivanov et al. [34] where they explained its implementation and performance at the SBST tool competition 2021 for Java [35]. Since its introduction, UTBot has become open source, more languages have been supported, and new approaches have been tried, such as concolic execution and fuzzing [36].

Due to the very different nature of symbolic execution compared to concrete execution, this type of execution is often less effective than concrete execution in benchmarks. Nonetheless, UTBot still managed to reach the podium of the SBFT tool competition 2023 for Java in 2023 [16] and the SBFT tool competition 2024 for Python [6].

#### Klara

Klara is an automated unit test generation tool for Python that uses AST analysis and an SMT solver [5]. It was developed by usagitoneko97 [5] and, within our knowledge, has not been introduced in a scientific paper. Furthermore, although it has been used in the SBFT tool competition 2024 for Python [6], it is still very experimental and has significant limitations, such as the lack of support for imports, loops and exceptions [37].

#### Hypothesis Ghostwriter

Hypothesis Ghostwriter is a module in the Hypothesis framework designed to generate property-based tests [38]. The Hypothesis framework was first introduced in a paper by MacIver et al. [38] and has since received numerous extensions, including Ghostwriter. Even if this tool was used in the SBFT tool competition 2024 for Python [6], the author of the tool has clarified that it is not the same kind of automated test generation tool as the others [39]. This tool aims to create

a template from which it is possible to start writing tests manually rather than creating tests to achieve the highest code coverage possible.

For example, the listing 2.2 shows what would happen if Ghostwriter tried to generate tests for a basic module called "multiply" defined in the listing 2.1.

```
def multiply(a, b):
    return a * b
```

Listing 2.1: The module "multiply"

```
# This test code was written by the ‘hypothesis.extra.ghostwriter‘ module
# and is provided under the Creative Commons Zero public domain dedication.

import multiply
from hypothesis import given, strategies as st

# TODO: replace st.nothing() with appropriate strategies


@given(a=st.nothing(), b=st.nothing())
def test_fuzz_multiply(a, b):
    multiply.multiply(a=a, b=b)
```

Listing 2.2: The tests generated for the module "multiply" by Ghostwriter

As you have read, the generated test is just a template which calls the function. Given that no type information is present, Ghostwriter gives control back to the tester rather than trying random types. As a result, although this tool was used in the SBFT tool competition 2024 for Python [6], it was not designed for this type of use.

#### Pynguin

Pynguin is an automated search-based unit test generation tool for Python that was introduced by Lukasczyk et al. [31]. It is more or less the equivalent of Evosuite but for automated test case generation in Python and was created to check whether it was possible to generate test cases in a dynamic language like Python. It implements the various algorithms that have been developed over time in Evosuite [27, 29, 30, 28] as well as Randoop's random algorithm [2]. As automated test case generation in Python is still fairly recent, there is still room for improvement, but it has been shown that the algorithms used in Java have potential for test case generation in Python [31, 4]. The authors of Pynguin nevertheless noted that the dynamic nature of Python added certain challenges, such as the fact that the tool's effectiveness depends very much on the type hints available [31]. Despite all this, in the SBFT tool competition 2024 for Python [6], Pynguin was the tool with the best overall performance [6].

## 2.3 Grammar-based fuzzing

Grammar-based fuzzing is a branch of fuzzing that uses grammar to generate inputs that have a certain shape [40, 41]. It is widely used in security testing [42] and can be black-box, grey-box or white-box [40]. As mentioned in the section on automated testing, fuzzers differ from test generation because they generate inputs rather than test cases. This is why these tools are usually more specialized in testing a particular entry point rather than being able to test multiple things at the same time. Nevertheless, this type of fuzzing has already been shown to generate inputs for testing compilers and interpreters successfully [40, 43]. Indeed, the use of grammar allows the creation of inputs with a valid structure but still rather random. As a result, grammars can easily generate inputs that respect a specific text file format, which, therefore, includes code.

## 2.4 Testing Machine Learning Programs

Software products using machine learning are becoming increasingly widespread nowadays. This makes the need to be able to test this kind of software more important than ever. However, these software products can be tested at many levels, making them more difficult to be tested [44].

### 2.4.1 Testing models training data

Model data testing is used to find problems in the quality of the data or in the way the data is used to train a model. Previous papers have shown that errors such as contradictions, missing data or inconsistencies in the data used to train a model harm the model performance [45]. This is why frameworks such as Active Clean [46] and BoostClean [47] have been created to interactively clean model data and remove errors in model training data. Some researchers have also proposed the idea of data linters [48], which are similar to code linters but for training data and aim to find frequent errors in training data.

### 2.4.2 Testing models

Model testing assumes that the code used to run the model is correct and looks for errors in model inference. It can be done using black-box testing or white-box testing.

In the black-box method, the most common technique is to use adversarial datasets to test the model. These datasets can be generated either by generative models (GAN) [49] or by mutating existing inputs [50].

In the white-box method, the techniques work either by using pseudo-oracles such as differential testing to identify errors in the models [51] or by guiding input generation through metrics such as neuron coverage [52, 53] or Multi-granularity Coverage Levels [54, 55].

### 2.4.3 Testing models code

To test the code of machine learning models, classic software testing approaches have already been applied and slightly adapted for machine learning programs, such as property-based testing [56], mutation testing [57] or coverage-guided fuzzing [58].

Automated test case generation approaches have also been tested on Java machine learning libraries, and it has been shown that conventional approaches do not work very well [7]. Such libraries often have many parameters and require inputs with a particular structure. This means some valid behaviours are never tested because parameters with a proper structure are not passed as arguments. This also means that some invalid behaviours are not tested because the tests cannot reach them since they are stuck on previous invalid behaviours.

One of the ideas proposed by Wang et al. [7] to correct the problem is to combine automated test case generation tools with data generation. This has already been tested by another paper where automated test generation was combined with grammar-based fuzzing to test JSON parsers [8], but this has not been tested with machine learning libraries.

# Chapter 3

# Motivation

In this chapter, we will focus on the problems that have already been solved in other papers and on the gaps that exist in the research. After that, we will define some research questions related to our research context, which is automated test case generation for machine learning programs.

## 3.1 Research gap

In the previous chapter, it has been shown that research for testing machine learning programs is mainly focused on input testing, manual test case writing or fuzzing techniques [44]. However, a study still has tried to use Evosuite, an automated test case generation tool, to generate test cases for various machine learning libraries in Java [7]. This study has unfortunately found that traditional approaches do not work well for machine learning libraries due to different reasons but the most important one is the lack of structured data. A paper already tried to combine Evosuite with grammar-based fuzzing in order to solve this problem of generating structured data but it was in the context of JSON parsing and not machine learning [8].

Furthermore, the research has focused on generating test cases for Java programs, whereas the machine learning community tends to use Python. It was shown that there have been recent progresses in the automated test case generation in Python thanks to the creation of Pynguin, which is roughly the equivalent of Evosuite for Python [4], but for now, it does not work very well on machine learning libraries either [6]. As a result, there is a large gap in the research when it comes to using techniques from Evosuite in Java on Pynguin in Python. In particular, the use of techniques that could allow machine learning programs to be tested more effectively would be a major improvement, given that the machine learning community mainly uses Python.

## 3.2 Definitions of the research questions

Given this gap in the research, 3 research questions appeared to us to be worth addressing:

**RQ1. How effective is Pynguin at automatically generating test cases for machine learning libraries?**

This research question aims to reproduce the results found in Java. This is why it is limited to libraries and not to programs in general. In addition, we have chosen to use only Pynguin as our automated test generation tool, since it is the most advanced tool for automatic test generation in Python [6] and the one we intend to build on in the second research question. The purpose of this question is also to see if there are any other problems arising from the differences between Python, which is a dynamic language, and Java, which is a static one.

**RQ2. To what extent does using structured input data improve the efficiency of test case generation in Pynguin for machine learning libraries?**

The aim of this research question is to use structured input data generated by using grammar-based fuzzing techniques and other ones to improve the test case generation of Pynguin for machine learning libraries. It was decided that we would limit ourselves to machine learning libraries in this question for now because they are the foundation of almost all machine learning programs. In addition, a bug in a machine learning library could have a major impact since it could affect all models trained using this library or using the library to work. Another aim of this research question is to see if the use of techniques that have been tried on Evosuite can work with Pynguin.

**RQ3. How much does parameter tuning improve the test cases generated by Pynguin?**

Arcuri and Fraser [59] showed that parameter tuning has an impact on the performance of Evosuite in test case generation. However, they also showed that it was difficult to find significantly better parameters than the default ones. In this research question, we intend to do some parameter tuning to see if we can find some optimisations that work well for improving automated test generation in the context of machine learning libraries. In addition, we also intend to do some parameter tuning of the input data generation algorithms that will be developed to answer the RQ2.

# Chapter 4

# Methodology

This chapter will explain the methodology used to answer the three research questions described in the previous chapter. Therefore, we will describe the changes that will be made to Pynguin, the benchmarks that will be used, and the statistical tools that will be employed. All the changes proposed and tools used in this chapter are available on GitHub in a replication package [10].

## 4.1 Approach and empirical evaluation related to the effectiveness of Pynguin on machine learning libraries (RQ1)

Given that the first research question only consists of reproducing the results found in Java with Evosuite [7] but in Python with Pynguin, our approach for this section will consist of using Pynguin on a small set of modules, doing a statistical analysis and then doing a qualitative analysis of the results.

### 4.1.1 Evaluation dataset

In their paper on automated test case generation for machine learning libraries in Java, Wang et al. [7] were able to measure Evosuite's code coverage on several classes. However, they did show something even more important: the set of reasons that make code coverage relatively low. That is why we decided to evaluate only a small number of modules in this section to be able to analyze the results in greater detail.

Initially, we wanted to use the modules related to machine learning and presented in the SBFT tool competition 2024 for Python [6], as the paper showed that the code coverage for some of them was not very high. However, after analyzing their dataset of modules, it turned out that most of the modules we were interested in had either no functions or only private functions, which greatly compromised their results. It might be interesting to check how common this is with the other modules in their dataset to get a clearer idea of the consequences that it will have and also to try and understand what caused this problem in the first place so that it does not happen again in the future.

For investigation purposes, here is the list of problematic modules in which we were interested:

- dtypes.py
- hdfs_wordcount.py
- logistic_regression_with_elastic_net.py
- normalizer_example.py
- plot_ensemble_oob.py
- plot_logistic.py
- plot_lw_vs_oas.py
- plot_regression.py

- prefetch_op.py
- tensor_spec.py
- tf_idf_example.py
- tf_import_time.py

In the end, we decided to use modules for parsing CSV files into dataframes from the Pandas and Polars libraries, as the CSV format is a structured format that is easily represented by a grammar, and we intend to use these libraries in our dataset for the second research question. Although this is a very small number of modules, they will likely reveal the same problems as in the Java paper due to the nature of the inputs they use. Furthermore, this research question aims to focus on the qualitative analysis rather than on the statistical analysis, so limiting our evaluation to a small number of modules is fine not to have too much data. The dataset of modules we selected and the version of Pynguin that will be used are shown in the table 4.1.

| **Module** | **Pynguin version** |
|---|---|
| pandas.io.parsers.readers | v0.34.0 |
| polars.io.csv.functions | v0.34.0 |

Table 4.1: The dataset of modules that will be analyzed and the version of Pynguin that will be used to answer RQ1

### 4.1.2 Experiment configuration

For this study, we have decided to use the DynaMOSA algorithm implemented in Pynguin as it is the default algorithm in the tool and to keep Pynguin's default search algorithm parameters. It has been shown that with Evosuite, the default parameters of the tool are reasonable for this type of experiment [59], and since Pynguin is very similar to Evosuite, we can expect a similar result for the parameters of Pynguin. Thus, the parameters used are:

- A population size of 50 test cases
- A single-point crossover with a probability of 0.75
- Mutation with a probability of 1/n, where n is the number of statements in the test case
- Tournament selection as selection operator

As for the search budget, we chose a limit of 10 minutes. This is the default budget in Pynguin if the parameter is not set, and it is a higher search budget than Evosuite because Python is much slower than Java. We have also added a timeout after 20 minutes in case there is a bug in Pynguin.

The test cases will be generated on a computer with an AMD Ryzen 5 6600H processor running at 3.3 GHz and having 6 cores and 12 threads. The environment will be partly controlled through the use of Docker, and a 16GB RAM limit will be established. A GitHub repository has been created to reproduce the environment [10].

### 4.1.3 Statistical analysis

Even though we will mainly conduct a qualitative analysis, we still need to repeat the experiments several times, as the experiments are subject to randomness. This will allow us to calculate the code coverage by calculating an average and better understand which lines of code are never covered. Therefore, we decided to repeat the experiment 30 times for each of the modules for this research question to get a good overview and to be able to run them relatively quickly.

This represents 30 runs * 10 minutes * 2 modules of continuous execution, which equals 600 minutes or 10 hours.

### 4.1.4 Qualitative analysis

To understand why some lines of code are not covered, we primarily intend to make a qualitative analysis of the code not executed by the generated tests. This will allow us to see whether, as in

the Java study, these lines of code are not covered because some arguments do not have the right structure.

## 4.2 Approach and empirical evaluation related to the usage of structured input data in Pynguin (RQ2)

This section will explain the evaluation dataset used and the changes made to Pynguin to answer RQ2. Given that the term "machine learning library" is fairly broad, we decided to focus on CSV parsing libraries because these are libraries whose inputs can be easily generated using grammar, and also on libraries that use tensors, as they are also data structures that can be easily generated. Although these libraries represent only a small proportion of the libraries that could benefit from grammar-based input generation and tensor generation, we have ensured that our approach could be easily adapted to other structured inputs using a plugin system.

### 4.2.1 Evaluation dataset

Our approach will be evaluated on modules that would benefit from using grammar-based inputs, like in the paper by Olsthoorn et al. [8], and modules that use tensors. In their paper, Olsthoorn et al. [8] used JSON inputs, but since our focus is on machine learning libraries, we will be using CSV inputs as it is a format that is broadly used in this field. Therefore, we decided to target two libraries widely used for parsing CSV files into dataframes, Pandas and Polars, and two libraries that use tensors, scipy and scikit-learn. However, we have also tried to go further than the approach of Olsthoorn et al. [8]. Indeed, as well as generating strings in CSV format, we also attempted to generate dataframes directly. We chose to do this to test both the parsing modules in particular and to test modules that use dataframes to perform operations on them. As for the Pynguin version used for each module, we used the most recent release and then the branch with CSV inputs, dataframe inputs or tensor inputs, depending on the kind of module being tested. We have decided to do this because our changes aim to allow certain plugins to be activated in cases where they might be useful, and there would be no advantage in using these plugins on irrelevant modules. This is also why we have manually selected some modules from the libraries listed above to create our dataset of modules for this research question. The dataset of modules we selected and the version of Pynguin that will be used are shown in the table 4.2.

| **Module** | **Pynguin version** |
|---|---|
| pandas.io.parsers.readers | v0.34.0 |
| pandas.io.parsers.readers | git @ grammar-based-file-like-objects |
| pandas.core.frame | v0.34.0 |
| pandas.core.frame | git @ grammar-based-file-like-objects |
| polars.io.csv.functions | v0.34.0 |
| polars.io.csv.functions | git @ grammar-based-file-like-objects |
| polars.dataframe.frame | v0.34.0 |
| polars.dataframe.frame | git @ grammar-based-file-like-objects |
| scipy.cluster.hierarchy | v0.34.0 |
| scipy.cluster.hierarchy | git @ tensor-objects |
| scipy.ndimage._fourier | v0.34.0 |
| scipy.ndimage._fourier | git @ tensor-objects |
| sklearn.cluster._kmeans | v0.34.0 |
| sklearn.cluster._kmeans | git @ tensor-objects |

Table 4.2: The dataset of modules that will be analyzed and the version of Pynguin that will be used to answer RQ2

### 4.2.2 Experiment configuration

To answer this research question, we decided to keep almost the same Pynguin parameters as for RQ1 and the same execution environment. The only differences in the parameters are the introduction of the parameters from the plugins, which will be described in a few sections and which have been left at their default values, and the use of simple assertions instead of those

generated by mutation analysis, as this can significantly reduce the total execution time and does not change the coverage code. We have also decreased the timeout to 15 minutes because simple assertions are faster than mutation analysis ones.

### 4.2.3 Statistical analysis

To check if our approach is better than the classic one, a statistical analysis will be performed to see whether there has been an improvement in code coverage and if there are any differences in the lines executed by the 2 approaches. Since both approaches are based on randomness, we expect a fair amount of variation from one run to another. To minimize this problem, each experiment was repeated 30 times so that an average could be calculated. To assess whether the results are statistically significant, we use the Mann-Whitney U-test test with a threshold of 0.05 as recommended by Arcuri and Briand [60]. This non-parametric test is used to determine whether there is a significant difference between the distributions of two independent samples. In addition, we have combined this U-test with the Vargha-Delaney $\hat{A}_{12}$ statistic to measure the effect size, which measures how large the difference between the two approaches is.

This represents 30 runs * 10 minutes * 7 modules * 2 approaches of continuous execution, which equals 4200 minutes or 70 hours.

### 4.2.4 Implementation of the plugin system

As mentioned above, the plugin system that we developed allows plugins to generate structured inputs for certain types. This change is independent of the underlying search algorithm used, but it can modify its results at different stages. Since we use DynaMOSA in this study, we will only explain the differences for this algorithm.

#### At the initialization stage

At the beginning of the evolution process, DynaMOSA creates an initial population of test cases. This population is generated by a procedure that generates test cases using the functions and classes analyzed in the target module. These different test cases are created by successively adding random calls to these functions and the constructors of these classes but first ensuring that there are variables that can be used as their arguments. These variables can be objects that have already been instantiated, primitives that have been randomly generated, or primitives from a constant pool, which is the set of constant values that exist in the source code. Since the number of generated calls is random for each test case in this procedure, DynaMOSA allows the creation of a population of test cases that are all distinct.

Our plugin system modifies this initial behaviour by allowing additional classes and functions to be analyzed on top of those already analyzed and allowing the plugin to take control when Pynguin needs to generate a variable that will be used as an argument for a call. In addition, our plugin system is quite smart, as it allows plugins to define the types they support so that they are not called to generate types that they do not support, and there is a weight system that allows the plugins to be more or less used to generate variables depending on their configuration.

#### At the selection stage

Our plugin system does not really change anything at the selection stage, but the inputs generated by plugins can impact it. Indeed, if an input generated by a plugin allows a given test case to fit the code coverage criteria better, then this test case will be selected by the tournament selection.

#### At the mutation stage

DynaMOSA uses mutations to introduce variations into the test case population. These mutations can be made to statements, where new ones can be added in the same way as at the initialization stage, where they can be modified by changing their parameters, and where they can be deleted. However, DynaMOSA also allows mutations to be made at the data level, where our plugin system can be used. Our plugin system allows new statement types to be added so that their data can be mutated as the plugins want.

### 4.2.5 Structure of a plugin

To explain more clearly how the plugin system works and what it can do, the structure of a plugin is shown in the listing 4.1, and we will explain it in the next sections.

```
NAME: str

def parser_hook(parser: ArgumentParser) -> None: ...


def configuration_hook(plugin_config: Namespace) -> None: ...


def types_hook() -> list[type | tuple[type, ModuleType]]: ...


def test_cluster_hook(test_cluster: TestCluster) -> None: ...


def variable_generator_hook(
    generators: dict[VariableGenerator, float]
) -> None: ...


def ast_transformer_hook(
    transformer_functions: dict[type, StatementToAstTransformerFunction]
) -> None: ...


def statement_remover_hook(
    remover_functions: dict[type,
    UnusedPrimitiveOrCollectionStatementRemoverFunction]
) -> None: ...
```

Listing 4.1: The structure of a Pynguin plugin

#### The "NAME" attribute

The "NAME" attribute is used to display a more explicit name in the logs. Given that the only information required to load a plugin is a file path or a module, this makes it possible to have a clearly defined name that is not necessarily linked to the name of the file in which it is defined. For example, if 2 plugins were defined at the file paths “csv/plugin.py” and “json/plugin.py”, using just the file name would have given the name “plugin” 2 times in the logs, even though it concerns 2 different plugins.

#### The "parser" hook

The "parser" hook aims to define new arguments in Pynguin to configure the plugin. This hook is very important as it makes it possible to do parameter tuning of the plugin’s parameters, which is required for RQ3, without modifying the plugin’s code, thus making plugins more user-friendly.

#### The "configuration" hook

The “configuration” hook is the continuation of the “parser” hook and is used to store arguments that have been parsed where the plugin needs them. Usually, this just means to store parser arguments in global variables.

#### The "types" hook

The “type” hook is the first hook that will actually affect Pynguin. It is called when the Module under Test is analyzed and allows additional types or functions to be analyzed. This is very important in certain cases, such as when interfaces/protocols or abstract classes are used or when no type information is provided in the Module under Test. Indeed, if Pynguin needs to generate a variable of the type of an abstract class, Pynguin will search for subclasses among the analyzed types. However, the analyzed classes only contain the Module under Test and its dependencies. If no sub-classes are present among all these classes, Pynguin will try to use random types rather than a type that would make more sense. Therefore, this hook makes it possible to add certain implementations of abstract classes if the plugin uses them later. It is also useful in the absence of

type information, as when generating a variable of type Any, Pynguin will choose a random type from the analyzed types to generate the variable. If the required type is not present among the analyzed types, Pynguin will never be able to generate it.

### The "test_cluster" hook

A test cluster is a Pynguin data structure storing all the information about the Module under Test, such as the analyzed functions and classes. The "test_cluster" hook allows the modification of this structure before the automated test generation has begun. This hook could do the same thing as the "type" hook but in a more complicated way, which is why they are separated. This hook is, therefore, mainly used for another purpose: modifying the weights of a type when converting an abstract type into a concrete one. For example, it allows testers to give more weight to certain types when Pynguin has to choose a random type when encountering an Any type.

### The "variable_generator" hook

The "variable_generator" hook allows testers to define other ways of generating variables. This is achieved by subclassing the "VariableGenerator" class and defining the types of variables it can generate, as well as a function for generating a variable in a test case. The listing 4.2 shows an example of a variable generator.

```
class CsvVariableGenerator(VariableGenerator):
    """A CSV variable generator."""

    @property
    def supported_types(self) -> SupportedTypes:  # noqa: D102
        return csv_supported_types

    def generate_variable(  # noqa: D102
        self,
        test_factory: TestFactory,
        test_case: TestCase,
        parameter_type: ProperType,
        position: int,
        recursion_depth: int,
        *,
        allow_none: bool,
    ) -> VariableReference | None:
        columns_number = randomness.next_int(
            csv_min_columns_number, csv_max_columns_number
        )

        csv_grammar = create_csv_grammar(
            columns_number=columns_number,
            min_field_length=csv_min_field_length,
            max_field_length=csv_max_field_length,
            min_rows_number=csv_min_rows_number,
            max_rows_number=csv_max_rows_number,
            number_column_probability=csv_number_column_probability,
            include_header=not csv_no_header,
        )

        csv_grammar_fuzzer = GrammarFuzzer(
            csv_grammar,
            csv_min_non_terminal,
            csv_max_non_terminal,
        )

        string_io_ret = test_case.add_variable_creating_statement(
            GrammarBasedFileLikeObjectStatement(test_case, csv_grammar_fuzzer),
            position,
        )
        string_io_ret.distance = recursion_depth

        return string_io_ret
```

Listing 4.2: Example of a variable generator that can create CSV file-like object

This hook also allows customized statements to be created. To model test cases, Pynguin uses "Statement" classes, which can represent all the types of statements in a test case, from assigning

a primitive to a variable to calling a class constructor. In the case of a plugin that generates structured inputs, it is, therefore, possible to generate new kinds of "Statement" classes that work like primitives. The listing 4.3 shows an example of a custom statement.

```
class GrammarBasedFileLikeObjectStatement(VariableCreatingStatement):
    """A statement that creates a grammar based file-like object."""

    def __init__(
        self,
        test_case: TestCase,
        fuzzer: GrammarFuzzer,
        derivation_tree: GrammarDerivationTree | None = None,
    ) -> None:
        """Create a new grammar based file-like object statement.

        Args:
            test_case: The test case
            fuzzer: The fuzzer to use
            derivation_tree: The derivation tree to use
        """
        if derivation_tree is None:
            derivation_tree = fuzzer.create_tree()

        self._derivation_tree = derivation_tree
        self._fuzzer = fuzzer
        self._csv_string = str(derivation_tree)

        string_io_type_info = test_case.test_cluster.type_system.to_type_info(
            io.StringIO
        )

        string_io_instance = Instance(string_io_type_info)

        super().__init__(
            test_case,
            vr.VariableReference(test_case, string_io_instance),
        )

    @property
    def csv_string(self) -> str:
        """The CSV string representation.

        Returns:
            The CSV string representation.
        """
        ...

    def mutate(self) -> bool:  # noqa: D102
        mutated = self._fuzzer.mutate_tree(self._derivation_tree)

        if mutated:
            self._csv_string = str(self._derivation_tree)

        return mutated

    ...
```

Listing 4.3: Example of a custom statement that model grammar-based file-like objects.

### The "ast_transformer" hook

The "ast_transformer" hook can be used to add new functions to convert a "Statement" class into a Python AST. This hook is only useful if custom statement types are used in the "variable_generator" hook.

### The "statement_remover" hook

The "statement_transformer" hook can be used to add new functions for deleting unnecessary statements before the current statement. This hook is only useful if custom statement types are used in the "variable_generator" hook.

### 4.2.6 Implementation of the grammar plugins

To partially answer the RQ2, 3 plugins have been created: 1 general plugin that generates CSV file-like objects and 2 plugins that generate dataframes for Pandas and Polars. These 3 plugins use the same grammar and grammar fuzzer to work. Unlike basic implementations of grammar fuzzers that only do manipulations on strings, this implementation uses derivation trees as recommended by Zeller et al. [61]. This has several advantages, particularly because it allows mutations to be made at the derivation tree level instead of on strings.

#### Implementation of the grammar

The grammar has been modelled using a set of frozen dataclasses that can be used by a visitor. This approach has resulted in classes that are high-level and, thanks to a visitor, easy to traverse. The listing 4.4 shows an example of a simple CSV grammar modelled using our classes.

```
Grammar(
    "csv",
    frozendict(
        csv=Repeat(RuleReference("row")),
        row=Sequence(rules=(RuleReference("field"), RuleReference("field"))),
        field=Repeat(AnyChar.letters_and_digits()),
    ),
)
```

Listing 4.4: Example of a simple CSV grammar.

In the introduction of this sub-section, it was said that all plugins use the same grammar; in reality, it is a little more complicated than that. Since a valid CSV file must have the same number of columns for each row, it is tricky to make a grammar with a random number of columns without using constraints. The need not to use constraints comes from the fact that it would take longer to create inputs, so it is not really appropriate in this case, where time is precious. That is why it was decided to create CSV grammars dynamically, with the possibility of randomly defining the number of columns, rows, and data length. This means there are as many grammars in the program as there are CSV "Statement" classes, and they are all generated with random parameters. This solution, therefore, solves the problem at the cost of a tiny bit of memory. As a bonus, each plugin has a set of parameters to make the grammar generation more or less random, thus enabling parameter tuning to be made.

#### Implementation of the fuzzer

For the fuzzer, we decided to use a heuristic based on the depth cost of the grammar rules to generate inputs of a certain size. Given that grammar generation takes a certain amount of time, the heuristic needed to be kept fairly simple so that it did not take too long but was still easily configurable. The implementation of this heuristic is based on the book by Zeller et al. [61], and its parameters can be configured using plugin arguments.

### 4.2.7 Implementation of the tensor plugin

A plugin that can generate tensors has been created to answer the rest of the RQ2. This plugin is much simpler than those using grammars, as it only adds a new type of primitive statement that generates tensors of random shape and that can be configured using the plugin's arguments.

## 4.3 Approach and empirical evaluation related to the parameter tuning of Pynguin (RQ3)

This section will explain the evaluation dataset used and how we will configure Pynguin to answer RQ3.

### 4.3.1 Evaluation dataset

The experiments in this section will use the same modules as for RQ2. This is necessary because the end goal is to be able to compare the results of RQ2 with RQ3 to see the implications of

parameter tuning. The dataset of modules we selected, the version of Pynguin that will be used and the parameters that will be tuned are shown in the table 4.3.

| **Module** | **Pynguin version** | **Parameter tuning** |
|---|---|---|
| pandas.io.parsers.readers | v0.34.0 | SOP |
| pandas.io.parsers.readers | git @ grammar-based-file-like-objects | SOP |
| pandas.io.parsers.readers | git @ grammar-based-file-like-objects | PW/PCW |
| pandas.io.parsers.readers | git @ grammar-based-file-like-objects | SOP/PW/PCW |
| pandas.core.frame | git @ grammar-based-file-like-objects | PW/PCW |
| polars.io.csv.functions | v0.34.0 | SOP |
| polars.io.csv.functions | git @ grammar-based-file-like-objects | SOP |
| polars.io.csv.functions | git @ grammar-based-file-like-objects | PW/PCW |
| polars.io.csv.functions | git @ grammar-based-file-like-objects | SOP/PW/PCW |
| polars.dataframe.frame | git @ grammar-based-file-like-objects | PW/PCW |
| scipy.cluster.hierarchy | git @ tensor-objects | PW/PCW |
| scipy.ndimage._fourier | git @ tensor-objects | PW/PCW |
| sklearn.cluster._kmeans | v0.34.0 | SOP |
| sklearn.cluster._kmeans | git @ tensor-objects | SOP |
| sklearn.cluster._kmeans | git @ tensor-objects | PW/PCW |
| sklearn.cluster._kmeans | git @ tensor-objects | SOP/PW/PCW |

Table 4.3: The dataset of modules that will be analyzed and the version of Pynguin that will be used to answer RQ3

### 4.3.2 Experiment configuration

To answer this research question, we decided to keep almost the same Pynguin parameters as for RQ2 and the same execution environment. The only difference is that we are going to do some parameter tuning. The parameters that we believe should be tuned include the probability of an optional parameter not being replaced by a generated value (SOP), the weight of CSVs, dataframes and tensors (PW), and the concrete weight of CSVs, dataframes and tensors (PCW). The first parameter speaks for itself, but we will explain the others. The weight of CSVs, dataframes or tensors is the value that will cause the CSV, dataframe or tensor plugin to be used when Pynguin encounters a plugin-supported type. For example, if Pynguin needs to generate a variable of type "Dataframe", it will have to choose between using the plugin or generating it in the usual way. By default, this choice is split equally between the usual Pynguin method and the plugins that can generate this type. Therefore, the weight parameter can be used to increase or decrease the importance of the plugins. The concrete weight parameter is used to increase or decrease the chance that Pynguin will select the plugin type rather than other types when it needs to convert an Any type into a concrete type.

### 4.3.3 Statistical analysis

As for the RQ2, a statistical analysis will be performed to see whether there has been an improvement in code coverage and to see if there are any differences in the lines that were executed depending on the parameter values. We will use the same statistical tests as for RQ2, so the Mann-Whitney U-test test with a threshold of 0.05 as recommended by Arcuri and Briand [60] and the Vargha-Delaney $\hat{A}_{12}$ statistic to measure the effect size. As for RQ2, we are also going to run each experiment 30 times, to be able to take averages and minimize the randomness of the algorithms.

This represents 30 runs * 10 minutes * 16 parameter tuning configurations of continuous execution, which equals 4800 minutes or 80 hours.

# Chapter 5

# Results

This chapter will examine the test case generation results and perform statistical and qualitative analyses. The results mentioned in this chapter are available in a Zenodo repository [9].

## 5.1 Results related to the effectiveness of Pynguin on machine learning libraries (RQ1)

When Pynguin tried to generate test cases for Pandas, it crashed every time with the error shown in the listing 5.1. After a quick manual review, we found that this was due to the fact that we were using the version "0.34.0" of Pynguin and that it had a bug that caused Pynguin to crash on certain libraries. This bug had already been noticed and was partially corrected, but only on the main branch of the GitHub repository [62].

```
TypeError: unsupported operand type(s) for |: 'NoneType' and 'NoneType'
```

Listing 5.1: First error that caused Pynguin to crash

Therefore, we tried running the test case generation on the main branch instead of on version "0.34.0". However, another bug has caused Pynguin to crash again, with the error shown in the listing 5.2 this time. This one was due to a limitation of Pynguin related to module aliases. Thus, we had to create a solution and it has been accepted on the project repository [63].

```
ModuleNotFoundError: No module named 'numpy.char'
```

Listing 5.2: Second error that caused Pynguin to crash

As always, after testing the test case generation on the fixed branch, another crash occurred, with the error shown in the listing 5.3 this time. It was a bug that has already been mentioned [64] and was due to the fact that Python is unable to infer the signature of C extension modules. So once again we have proposed a solution that involves using a very general signature in this kind of case and which has also been accepted on the project repository [65].

```
ValueError: no signature found for builtin <slot wrapper '__init__' of
'pyarrow.lib.ExtensionType' objects>
```

Listing 5.3: Third error that caused Pynguin to crash

Finally, Pynguin was operational and began its iterations to generate test cases. However, after a few minutes, other types of crashes appeared, as shown in table 5.1. Out of 30 runs for the "pandas.io.parsers.readers" module, none ended successfully. The problem is that this time, it was not coming from bugs in Pynguin directly, it was coming from Pandas. Indeed, some Pandas bugs, such as memory leaks, Python GIL deadlocks, or segmentation faults, can have repercussions on Pynguin given that it executes the Module under Test in the same process as the one in which it is executed and using the same Python interpreter.

| Success | Timeout | Out of memory | Segmentation fault |
|---|---|---|---|
| 0 | 16 | 13 | 1 |

Table 5.1: The results of the test case generation for the "pandas.io.parsers.readers" module using Pynguin version "0.35.0"

In response to this problem, we have decided to stop using the current methodology and try to find a new one that will not be vulnerable to these types of problems. This choice was also motivated by the fact that other small tests were carried out on Polars and other modules using C-extension modules and many of them created these types of faults. These small tests nevertheless revealed some bugs in Pynguin concerning assertion generation and AST analysis, and these have been fixed in the "main-fixed" branch [66].

To be complete, we still attempted to run the test case generation on the "main-fixed" branch and the results can be seen in the table 5.2.

| Experiment | Coverage | Iterations | Total time | Search time | Mutation score | Success | Failure | T/O | Segfault | OOM | FPE | Other crashes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pandas_parser_main | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 13 | 0 | 17 | 0 | 0 |
| polars_parser_main | 0.04 | 75.03 | 61.12 | 40.04 | 0.07 | 2 | 0 | 27 | 1 | 0 | 0 | 0 |

Table 5.2: The results of the test case generation collected to answer RQ1

Pynguin managed to run twice on Polars, but this was so rare that we did not carry out a qualitative study until the methodology had been redesigned.

## 5.2 Results related to the usage of structured input data in Pynguin (RQ2)

Even though the results for RQ1 showed that very few runs ended successfully for the Pandas and Polars parsers. It is still interesting to see whether the use of grammar or tensors changes the nature of the crashes or if these results also happen in other modules. The results that we collected can be seen in the table 5.3.

| Experiment | Coverage | Iterations | Total time | Search time | Success | Failure | T/O | Segfault | OOM | FPE | Other crashes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| pandas_dataframe_main_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 30 | 0 | 0 | 0 | 0 | 0 |
| pandas_dataframe_main_grammar_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 30 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_main_simple | 0.09 | 32.33 | 121.21 | 102.56 | 2 | 1 | 13 | 1 | 13 | 0 | 0 |
| pandas_parser_main_grammar_simple | 0.16 | 38.67 | 181.82 | 161.06 | 8 | 0 | 13 | 3 | 6 | 0 | 0 |
| polars_dataframe_main_simple | 0.13 | 95.73 | 189.54 | 180.27 | 8 | 2 | 5 | 3 | 12 | 0 | 0 |
| polars_dataframe_main_grammar_simple | 0.30 | 217.07 | 466.48 | 440.61 | 20 | 2 | 1 | 0 | 7 | 0 | 0 |
| polars_parser_main_simple | 0.21 | 289.03 | 181.79 | 180.13 | 9 | 0 | 21 | 0 | 0 | 0 | 0 |
| polars_parser_main_grammar_simple | 0.26 | 381.13 | 242.41 | 240.15 | 12 | 0 | 18 | 0 | 0 | 0 | 0 |
| scipy_cluster_hierarchy_main_simple | 0.01 | 12.73 | 21.62 | 20.24 | 0 | 0 | 1 | 1 | 28 | 0 | 0 |
| scipy_cluster_hierarchy_main_tensor_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 0 | 0 | 30 | 0 | 0 |
| scipy_ndimage_fourier_main_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 0 | 1 | 29 | 0 | 0 |
| scipy_ndimage_fourier_main_tensor_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 0 | 1 | 29 | 0 | 0 |
| sklearn_cluster_kmeans_main_simple | 0.00 | 27.47 | 62.59 | 60.24 | 4 | 0 | 1 | 1 | 4 | 20 | 0 |
| sklearn_cluster_kmeans_main_tensor_simple | 0.01 | 57.30 | 103.68 | 100.19 | 6 | 0 | 0 | 2 | 1 | 20 | 1 |

Table 5.3: The results of the test case generation collected to answer RQ2

We can see that, as in the RQ1 results, very few runs ended successfully. The exception to this rule is the "polars.dataframe.frame" module on the grammar branch, which managed to get run 20 times. This module also contains 2 failures, but they are due to an out of file descriptor error and not to an error in Pynguin. Furthermore, we can also see that all the runs on the "pandas.core.frame" module have failed. This is due to a conflict between the metaprogramming that Pandas does to generate its documentation and Pynguin's dynamic analysis.

## 5.3 Results related to the parameter tuning of Pynguin (RQ3)

Like what was said about the RQ2 results, it may also be interesting to see whether parameter tuning makes a difference or not. The results that we collected can be seen in the table 5.4.

| Experiment | Coverage | Iterations | Total time | Search time | Success | Failure | T/O | Segfault | OOM | FPE | Other crashes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| pandas_dataframe_main_grammar_weights_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 30 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_main_grammar_weights_simple | 0.12 | 43.50 | 138.51 | 124.34 | 4 | 0 | 11 | 5 | 10 | 0 | 0 |
| pandas_parser_main_finetuned_simple | 0.05 | 7.83 | 72.91 | 62.63 | 2 | 0 | 14 | 9 | 5 | 0 | 0 |
| pandas_parser_main_finetuned_grammar_simple | 0.02 | 1.70 | 26.51 | 21.22 | 1 | 0 | 5 | 19 | 5 | 0 | 0 |
| pandas_parser_main_finetuned_grammar_weights_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 2 | 26 | 2 | 0 | 0 |
| polars_dataframe_main_grammar_weights_simple | 0.37 | 229.93 | 442.98 | 420.71 | 19 | 2 | 4 | 3 | 2 | 0 | 0 |
| polars_parser_main_grammar_weights_simple | 0.38 | 435.13 | 405.13 | 400.41 | 20 | 0 | 10 | 0 | 0 | 0 | 0 |
| polars_parser_main_finetuned_simple | 0.20 | 289.13 | 262.35 | 260.21 | 13 | 0 | 17 | 0 | 0 | 0 | 0 |
| polars_parser_main_finetuned_grammar_simple | 0.27 | 409.30 | 362.99 | 360.42 | 18 | 0 | 11 | 1 | 0 | 0 | 0 |
| polars_parser_main_finetuned_grammar_weights_simple | 0.28 | 401.40 | 423.67 | 420.36 | 21 | 0 | 8 | 1 | 0 | 0 | 0 |
| scipy_cluster_hierarchy_main_tensor_weights_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 0 | 1 | 11 | 0 | 18 |
| scipy_ndimage_fourier_main_tensor_weights_simple | 0.13 | 181.97 | 109.00 | 103.45 | 4 | 0 | 0 | 0 | 26 | 0 | 0 |
| sklearn_cluster_kmeans_main_tensor_weights_simple | 0.00 | 28.63 | 62.33 | 60.26 | 5 | 0 | 1 | 12 | 0 | 4 | 8 |
| sklearn_cluster_kmeans_main_finetuned_simple | 0.01 | 50.53 | 103.93 | 100.48 | 5 | 0 | 2 | 0 | 2 | 21 | 0 |
| sklearn_cluster_kmeans_main_finetuned_tensor_simple | 0.01 | 57.03 | 103.44 | 100.41 | 5 | 0 | 5 | 0 | 1 | 19 | 0 |
| sklearn_cluster_kmeans_main_finetuned_tensor_weights_simple | 0.02 | 27.30 | 41.44 | 40.14 | 3 | 0 | 0 | 16 | 0 | 4 | 7 |

Table 5.4: The results of the test case generation collected to answer RQ3

We can see that, unlike in the RQ2 results, a few modules were able to end successfully more than 50% of the time. In addition, the number of fault segmentation and other crashes has also increased on some modules.

# Chapter 6

# New methodology

This chapter will explain the new methodology used to answer the 3 research questions described in Chapter 3. It is an improvement on the previous methodology and aims to fix the problems that prevent test cases from being generated for modules that cause failures such as segmentation faults, Python GIL deadlocks or memory leaks.

## 6.1 Approach and empirical evaluation related to the effectiveness of Pynguin on machine learning libraries (RQ1)

This section will present the changes made to the previous approach and empirical evaluation for RQ1.

### 6.1.1 Introduction of subprocesses

In the previous chapter, a series of problems were detected when generating test cases for a module, in particular memory leaks, segmentation faults and Python GIL deadlocks. These problems are complicated to fix because they come from the Module under Test and not from Pynguin.

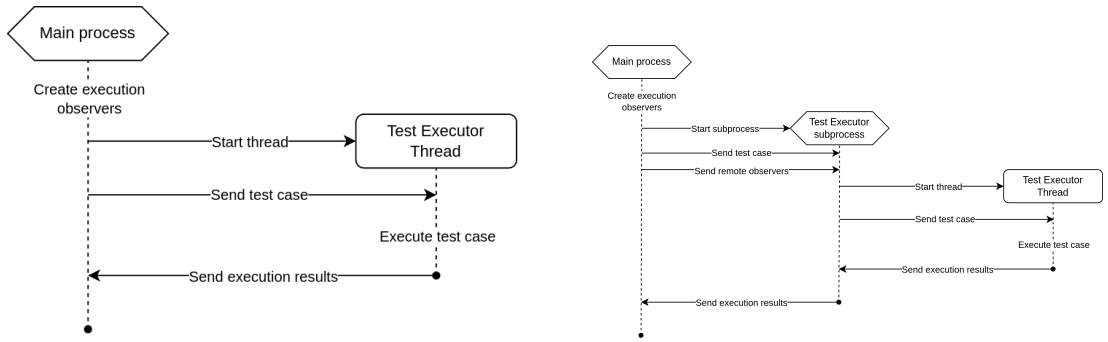
After much thought, we have come up with just one solution to the problem: the use of subprocesses. Indeed, using subprocesses solves all the above-mentioned problems, even if it introduces other less important problems that we will discuss later. First, memory leaks are fixed because the OS can automatically kill a process using too much memory and then automatically free up the process's memory. Secondly, by using a subprocess, it is possible to observe the return code of the subprocess to check whether there has been a segmentation fault and without the segmentation fault affecting the main process. Lastly, it is possible to wait for subprocesses for a limited time and to kill any subprocess that exceeds this time, thus allowing to deal with Python GIL deadlocks.

### 6.1.2 Implementation of the architecture using subprocesses

The original architecture of Pynguin is very dynamic, using a lot of observers to monitor different parts of the test case execution to calculate metrics or gather information. This complicates the introduction of subprocesses because an observer in the main process cannot observe the execution of a test case in a subprocess. Fortunately, the observers used in Pynguin were already designed to be more or less isolated from the rest of the application when observing the execution of a test case because they were meant to be used in another thread. Thanks to this detail, even though the change in architecture was quite big, it was not too hard to achieve.

Therefore, our idea was to separate Pynguin observers into 2 types: Remote and main process observers. Remote observers use the dynamic nature of Python and the dill and multiprocess components of the Pathos framework [67, 68] to be sent to a subprocess, observe the execution of a test case and then return the results to the main process. This technique allows us to reconcile the use of observers with the use of subprocesses. Main process observers observe test cases before and after their execution in subprocesses. They are intended to provide a bridge with the old

Pynguin architecture. Each main process observer can contain a remote observer that will be used to gather information about test case execution in a subprocess if needed. A comparison of the 2 architectures is shown in figure 6.1.




(a) The original architecture (b) The architecture using subprocesses

Figure 6.1: Comparison between the original architecture and the architecture using subprocesses

### 6.1.3 Advantages of the architecture using subprocesses

A major advantage of modifying the architecture this way is that test cases can continue to be run without subprocesses. Indeed, remote observers can be executed in the main process just as easily without causing problems. As a result, this new architecture allows testers to choose between executing the test cases in the main process or some subprocesses with just one configuration variable.

In addition, now that it is possible to check whether the execution of a test case has caused a failure, it is possible to create error-revealing test cases. It is a concept commonly used in tools that use random testing, such as Randoop [2], and involves creating test cases that deliberately crash to reveal a bug in the code. Often, this type of tool uses specifications from the standard library to assert that there is a bug in certain cases. In our case, we use the fact that the MuT crashes with a segmentation fault, a memory leak or something else, as these are things that are not supposed to happen.

### 6.1.4 Drawbacks of the architecture using subprocesses

The biggest problem with this new architecture is the need to send data between the main process and the subprocess. While sending data to a subprocess is quite fast, as you only need to make a fork to launch the subprocess and retrieve the data, sending data to the main process is more complicated. Indeed, to send data back to the main process, the new architecture uses the dill and multiprocess libraries, which are components of the Pathos framework [67, 68] designed to transfer data between processes easily. The problem is that dill, which converts the data into binary, is written in pure Python and is very slow. Furthermore, dill is sometimes unable to convert certain objects into binary, so it is sometimes necessary to sacrifice some non-essential data to avoid crashing Pynguin when sending data back. Nevertheless, these are fairly minor drawbacks in comparison with not being able to generate test cases at all.

### 6.1.5 Evaluation dataset

This new approach will be evaluated in the same way as it was in the previous methodology, using the Pandas and Polars parser classes. The aim is still to focus on the qualitative analysis rather than the statistical analysis and attempt to reproduce the results found in Java [7]. The dataset of modules we selected and the version of Pynguin that will be used are shown in the table 6.1.

| Module | Pynguin version |
|---|---|
| pandas.io.parsers.readers | git @ process-based-tests-execution |
| polars.io.csv.functions | git @ process-based-tests-execution |

Table 6.1: The dataset of modules that will be analyzed and the version of Pynguin that will be used to answer RQ1

### 6.1.6 Experiment configuration

To answer this research question, we decided to keep almost the same Pynguin parameters as for the previous approach and empirical evaluation for RQ1 and the same execution environment. The only differences are that test cases will be executed in some subprocesses and that the timeout will be increased to 4500s for Pandas and 3000s for Polars since the mutation analysis is much slower when using subprocesses.

### 6.1.7 Statistical analysis

Like in the the previous methodology, we still need to repeat the experiments several times, as the experiments are subject to randomness. Therefore, we decided to repeat the experiment 30 times for each module, like in the previous statistical analysis for RQ1.

This represents 30 runs * 10 minutes * 2 modules of continuous execution, which equals 600 minutes or 10 hours.

### 6.1.8 Qualitative analysis

Like in the the previous methodology, we will still do a qualitative analysis to try to understand why some lines of code are not covered and to try to see if we found the same results as in Java [7].

## 6.2 Approach and empirical evaluation related to the usage of structured input data in Pynguin (RQ2)

This section will present the changes made to the previous approach and empirical evaluation for RQ2.

### 6.2.1 Integration of the architecture using subprocesses with the plugin system

Integrating the architecture using subprocesses with the plugin system was easy since the 2 changes concern different parts of the code. The only thing required was to merge the 2 branches and fix the small differences.

### 6.2.2 Evaluation dataset

This new approach will be evaluated in the same way as it was in the previous methodology. Therefore, we decided to keep targeting two libraries widely used for parsing CSV files into dataframes, Pandas and Polars, as well as 2 libraries that use tensors, scipy and scikit-learn. The dataset of modules we selected and the version of Pynguin that will be used are shown in the table 6.2.

| Module | Pynguin version |
|---|---|
| pandas.io.parsers.readers | git @ process-based-tests-execution |
| pandas.io.parsers.readers | git @ subprocess+plugin-system+grammar-plugin |
| pandas.core.frame | git @ process-based-tests-execution |
| pandas.core.frame | git @ subprocess+plugin-system+grammar-plugin |
| polars.io.csv.functions | git @ process-based-tests-execution |
| polars.io.csv.functions | git @ subprocess+plugin-system+grammar-plugin |
| polars.dataframe.frame | git @ process-based-tests-execution |
| polars.dataframe.frame | git @ subprocess+plugin-system+grammar-plugin |
| scipy.cluster.hierarchy | git @ process-based-tests-execution |
| scipy.cluster.hierarchy | git @ subprocess+plugin-system+tensor |
| scipy.ndimage._fourier | git @ process-based-tests-execution |
| scipy.ndimage._fourier | git @ subprocess+plugin-system+tensor |
| sklearn.cluster._kmeans | git @ process-based-tests-execution |
| sklearn.cluster._kmeans | git @ subprocess+plugin-system+tensor |

Table 6.2: The dataset of modules that will be analyzed and the version of Pynguin that will be used to answer RQ2

### 6.2.3 Experiment configuration

To answer this research question, we decided to keep almost the same Pynguin parameters as for the previous approach and empirical evaluation for RQ2 and the same execution environment. The only difference is that test cases will be executed in some subprocesses.

### 6.2.4 Statistical analysis

Like in the the previous methodology, we still need to repeat the experiments several times, as the experiments are subject to randomness. Therefore, we decided to repeat the experiment 30 times for each module, like in the previous statistical analysis for RQ2. We will also use the same statistical tests, so the Mann-Whitney U-test test with a threshold of 0.05 as recommended by Arcuri and Briand [60] and the Vargha-Delaney $\hat{A}_{12}$ statistic to measure the effect size.

This represents 30 runs * 10 minutes * 7 modules * 2 approaches of continuous execution, which equals 4200 minutes or 70 hours.

## 6.3 Approach and empirical evaluation related to the parameter tuning of Pynguin (RQ3)

This section will present the changes made to the previous approach and empirical evaluation for RQ3.

### 6.3.1 Evaluation dataset

The experiments in this section will use the same modules as for RQ2. The only difference is that they will use different parameters. The end goal is to be able to compare the results of RQ2 with RQ3 to see the implications of parameter tuning. Regarding the parameter tuning, it was decided to keep the same parameters as in the previous methodology. The dataset of modules we selected, the version of Pynguin that will be used and the parameters that will be tuned are shown in the table 6.3.

| Module | Pynguin version | Parameter tuning |
|---|---|---|
| pandas.io.parsers.readers | git @ process-based-tests-execution | SOP |
| pandas.io.parsers.readers | git @ subprocess+plugin-system+grammar-plugin | SOP |
| pandas.io.parsers.readers | git @ subprocess+plugin-system+grammar-plugin | PW/PCW |
| pandas.io.parsers.readers | git @ subprocess+plugin-system+grammar-plugin | SOP/PW/PCW |
| pandas.core.frame | git @ subprocess+plugin-system+grammar-plugin | PW/PCW |
| polars.io.csv.functions | git @ process-based-tests-execution | SOP |
| polars.io.csv.functions | git @ subprocess+plugin-system+grammar-plugin | SOP |
| polars.io.csv.functions | git @ subprocess+plugin-system+grammar-plugin | PW/PCW |
| polars.io.csv.functions | git @ subprocess+plugin-system+grammar-plugin | SOP/PW/PCW |
| polars.dataframe.frame | git @ subprocess+plugin-system+grammar-plugin | PW/PCW |
| scipy.cluster.hierarchy | git @ subprocess+plugin-system+tensor | PW/PCW |
| scipy.ndimage._fourier | git @ subprocess+plugin-system+tensor | PW/PCW |
| sklearn.cluster._kmeans | git @ process-based-tests-execution | SOP |
| sklearn.cluster._kmeans | git @ subprocess+plugin-system+tensor | SOP |
| sklearn.cluster._kmeans | git @ subprocess+plugin-system+tensor | PW/PCW |
| sklearn.cluster._kmeans | git @ subprocess+plugin-system+tensor | SOP/PW/PCW |

Table 6.3: The dataset of modules that will be analyzed and the version of Pynguin that will be used to answer RQ3

### 6.3.2 Experiment configuration

To answer this research question, we decided to keep almost the same Pynguin parameters as for the previous approach and empirical evaluation for RQ3 and the same execution environment. The only difference is that test cases will be executed in some subprocesses.

### 6.3.3 Statistical analysis

Like in the the previous methodology, we still need to repeat the experiments several times, as the experiments are subject to randomness. Therefore, we decided to repeat the experiment 30 times for each module, like in the previous statistical analysis for RQ3. We will also use the same statistical tests, so the Mann-Whitney U-test test with a threshold of 0.05 as recommended by Arcuri and Briand [60] and the Vargha-Delaney $\hat{A}_{12}$ statistic to measure the effect size.

This represents 30 runs * 10 minutes * 16 parameter tuning configurations of continuous execution, which equals 4800 minutes or 80 hours.

# Chapter 7

# New results

This chapter will examine the new test case generation results and perform statistical and qualitative analysis. The results mentioned in this chapter are available in a Zenodo repository [9].

## 7.1 Results related to the effectiveness of Pynguin on machine learning libraries (RQ1)

This section will describe the collected results to answer RQ1. This includes aggregated experiment data over several runs, graphs to represent better the code coverage, and code snippets of interest to understand these graphs better.

### 7.1.1 Aggregated experiment data

First, we will describe the aggregated experiment data aimed at answering RQ1. They are shown in the table 7.1.

| Experiment | Coverage | Iterations | Total time | Search time | Mutation score | CTC | Success | Failure | T/O | Segfault | OOM | FPE | Other crashes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pandas_parser_subprocess | 0.35 | 16.63 | 2619.64 | 615.59 | 0.99 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess | 0.38 | 70.10 | 927.60 | 604.45 | 1.00 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 7.1: The aggregated experiment data collected to answer RQ1

The results show that code coverage is 35% for Pandas and 38% for Polars. These values are below those found by Lukasczyk and Fraser [4] in their paper introducing Pynguin, where an average code coverage of 68% was found with the DynaMOSA algorithm on 118 modules.

Furthermore, we can observe that, contrary to the previous results, all the experiments were successful, and there were no failures.

Finally, it can be noted that although some crashes were detected in the previous results, the Crash test count (CTC) indicates that no test cases designed to reveal a crash have been generated.

### 7.1.2 Line hit frequency graphs

Graphs representing the line hit frequency have been created and are shown in the figures 7.1 and 7.2. This makes it possible to see if some lines have never been covered or which lines have been covered only by some of the generated test suites. These graphs show the line numbers of each instruction in the code on the y-axis and the frequency at which each instruction was covered by the test suites generated on the x-axis. For example, if an instruction has only been covered by half of the test suites generated, then its line hit frequency will be 0.5.

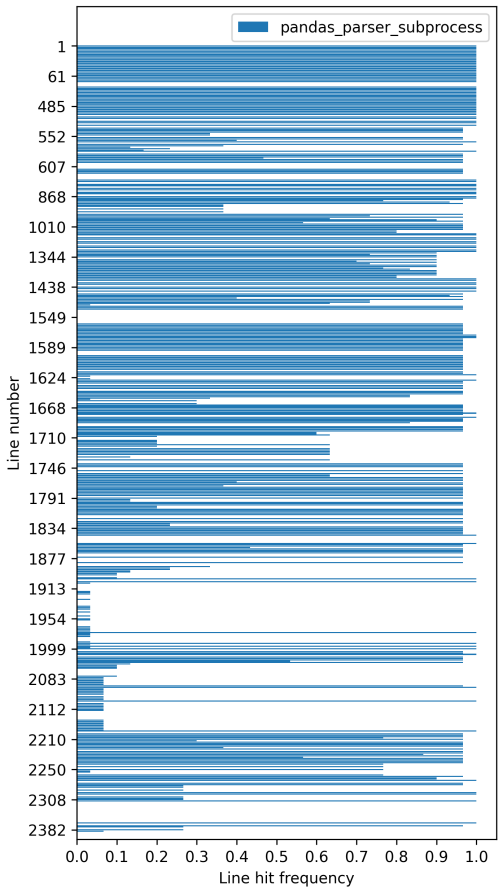



Figure 7.1: The line hit frequency of 30 generated test suites for the "pandas.io.parsers.readers" module using the traditional approach

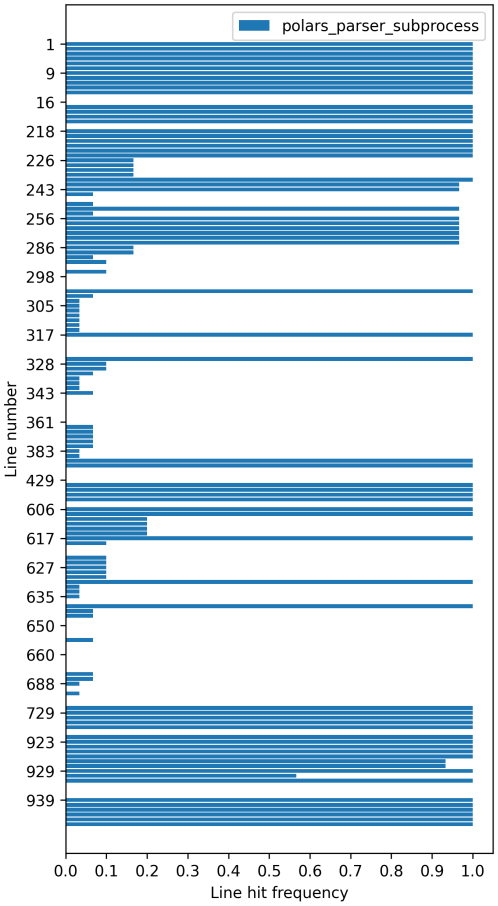



Figure 7.2: The line hit frequency of 30 generated test suites for the "polars.io.csv.functions" module using the traditional approach

On these graphs, we can see that some lines are always executed by all runs, some lines are only executed by some runs, and some have never been executed after 30 runs. This behaviour is only visible because these graphs show the line hit frequency across several runs, but it would not be possible to see this by just looking at the average code coverage.

### 7.1.3 Qualitative analysis

Using the line hit frequency graphs presented above, it is possible to examine the lines of code that have never been executed. The results are shown in the tables 7.2 and 7.3.

| **Type** | **Line numbers** |
|---|---|
| Unreachable code (type checking block) | 74-82 |
| Unreachable code (overload body) | 514;519;524;685;745;805;865; 1083;1140;1197;1254;1346;1420 ;1435;1450 |
| Uncovered docstring | 530;560;586;1464;2000;2107; 2144;2268;2315;2366 |
| Optional parameter validation | 1542-1555;1643;1658;1681;1790;2381 |
| Unused optional feature (invalid optional parameter value) | 552;602-608;961;965;967;1593-1597;1617;1695-1696;1710-1711;1722;1728;1730;1732-1737;1746-1751;1818-1820;1826;1868-1869;1877-1879;1905;1909-1914;1931-1936;1943;1946-1947;1952;1957-1964;1982-1984;2073-2079;2100;2116-2122;2126;2247;2249;2280 |
| Unreachable code (invalid parameter) | 622-626;1624;1842-1846;1890-1891;1901;1926-1928;2284;2330-2362 |
| Uncovered variable declaration | 1631;2057 |

Table 7.2: Analysis of the uncovered lines of the "pandas.io.parsers.readers" module

| **Type** | **Line numbers** |
|---|---|
| Unreachable code (type checking block) | 15-16 |
| Uncovered docstring | 57;466;763 |
| Unused optional feature (invalid optional parameter value) | 247;293;346-361;658-670;689;692-694;934-937 |
| Optional parameter validation | 318-320;619-620;649-651 |
| Unreachable code (invalid parameters) | 298-300;325;427-429;640;654 |

Table 7.3: Analysis of the uncovered lines of the "polars.io.csv.functions" module

Among these results, we can first see that some lines that were not executed are caused by limitations of unit testing, like some docstrings or the lines used for type checking. Then, many lines representing optional features are not executed because of invalid values of optional parameters. This is a result that Wang et al. [7] also found in Java, even if the optional parameters are slightly different in this language. Finally, we can see that there are lines of code that are not accessible because of invalid parameters. Among these, it should be noted that there are lines designed to create dataframes, which is the main purpose of the 2 MuT. The listings 7.1, 7.2, and 7.3 show an overview of these lines of code.

```
620     parser = TextFileReader(filepath_or_buffer, **kwds)
621
622     if chunksize or iterator:
623         return parser
624
625     with parser:
626         return parser.read(nrows)
```

Listing 7.1: Overview of lines 620 to 626 of the module "pandas.io.parsers.readers"

```
297         df = pl.DataFrame._from_arrow(tbl, rechunk=rechunk)
298         if new_columns:
299             return _update_columns(df, new_columns)
300         return df
```

Listing 7.2: Overview of lines 297 to 300 of the module "polars.io.csv.functions"

```
390     with _prepare_file_arg(
391         source,
392         encoding=encoding,
```

```
393         use_pyarrow=False,
394         raise_if_empty=raise_if_empty,
395         storage_options=storage_options,
396     ) as data:
397         df = pl.DataFrame._read_csv(
398             data,
399             has_header=has_header,
400             columns=columns if columns else projection,
401             separator=separator,
402             comment_prefix=comment_prefix,
403             quote_char=quote_char,
404             skip_rows=skip_rows,
405             dtypes=dtypes,
406             schema=schema,
407             null_values=null_values,
408             missing_utf8_is_empty_string=missing_utf8_is_empty_string,
409             ignore_errors=ignore_errors,
410             try_parse_dates=try_parse_dates,
411             n_threads=n_threads,
412             infer_schema_length=infer_schema_length,
413             batch_size=batch_size,
414             n_rows=n_rows,
415             encoding=encoding if encoding == "utf8-lossy" else "utf8",
416             low_memory=low_memory,
417             rechunk=rechunk,
418             skip_rows_after_header=skip_rows_after_header,
419             row_index_name=row_index_name,
420             row_index_offset=row_index_offset,
421             sample_size=sample_size,
422             eol_char=eol_char,
423             raise_if_empty=raise_if_empty,
424             truncate_ragged_lines=truncate_ragged_lines,
425         )
426
427     if new_columns:
428         return _update_columns(df, new_columns)
429     return df
```

Listing 7.3: Overview of lines 390 to 429 of the module "polars.io.csv.functions"

### 7.1.4 Findings

Our results and analysis can be summarized as follows:

**Finding 1. The code coverage achieved by Pynguin on certain machine learning modules is lower than the average code coverage achieved by Pynguin on other modules.**

**Finding 2. The generated test cases do not cover many features of each MuT because Pynguin does not generate the right inputs when calling the functions of the MuT.**

## 7.2 Results related to the usage of structured input data in Pynguin (RQ2)

In this section, we will describe the collected results to answer RQ2. This includes aggregated experiment data over several runs, statistical comparison between the approaches and graphs to represent better the code coverage.

### 7.2.1 Aggregated experiment data

First, we will describe the aggregated experiment data aimed at answering RQ2. They are shown in the table 7.4.

| Experiment | Coverage | Iterations | Total time | Search time | CTC | Success | Failure | T/O | Segfault | OOM | FPE | Other crashes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pandas_dataframe_subprocess_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 30 | 0 | 0 | 0 | 0 | 0 |
| pandas_dataframe_subprocess_grammar_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 30 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_subprocess_simple | 0.33 | 16.63 | 666.51 | 617.53 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_subprocess_grammar_simple | 0.36 | 17.00 | 670.41 | 619.63 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_dataframe_subprocess_simple | 0.02 | 4.10 | 694.67 | 636.27 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_dataframe_subprocess_grammar_simple | 0.02 | 4.27 | 699.95 | 642.54 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess_simple | 0.36 | 55.30 | 683.85 | 605.38 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess_grammar_simple | 0.38 | 56.60 | 694.83 | 604.76 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| scipy_cluster_hierarchy_subprocess_simple | 0.12 | 9.50 | 706.08 | 631.85 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| scipy_cluster_hierarchy_subprocess_tensor_simple | 0.13 | 9.80 | 699.36 | 628.41 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| scipy_ndimage_fourier_subprocess_simple | 0.66 | 91.00 | 614.98 | 603.74 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| scipy_ndimage_fourier_subprocess_tensor_simple | 0.64 | 98.47 | 593.86 | 583.73 | 1 | 29 | 1 | 0 | 0 | 0 | 0 | 0 |
| sklearn_cluster_kmeans_subprocess_simple | 0.04 | 15.23 | 649.05 | 618.44 | 3 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| sklearn_cluster_kmeans_subprocess_tensor_simple | 0.04 | 16.17 | 647.65 | 618.93 | 1 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 7.4: The aggregated experiment data collected to answer RQ2

First, we can see that the code coverages are still lower than the average code coverage of 68% found by Lukasczyk and Fraser [4] except for the "scipy.ndimage._fourier" module, which is quite close.

Then, we can observe that, unlike in the the previous results, all the experiments were successful, and there were no failures except for the "pandas.core.frame" module. The causes of this failure are the same as the ones explained in the previous results.

Finally, we can see that the Crash test count (CTC) is greater than 0 for the module "sklearn.cluster._kmeans", indicating that some tests revealing errors have been created. Nevertheless, the original approach created more crash tests than our approach. The listing 7.4 shows one of the generated crash test cases, and the listing 7.5 shows its output.

```
# Test cases automatically generated by Pynguin (https://www.pynguin.eu).
# Please check them before you use them.
import sklearn.cluster._kmeans as module_0
import scipy.ndimage._morphology as module_1
import scipy.optimize._optimize as module_2


def test_case_0():
    none_type_0 = None
    k_means_0 = module_0.KMeans(init=none_type_0, algorithm=none_type_0)
    var_0 = module_1.binary_propagation(none_type_0, mask=k_means_0, output=
    none_type_0)
    var_1 = var_0.__enter__()
    var_2 = var_1.__new__(k_means_0, k_means_0)
    var_3 = var_2.moment(none_type_0, *k_means_0)
    var_4 = var_3.view(type=k_means_0)
    var_5 = var_4.sort()
    var_6 = var_5.var()
    var_7 = var_6.__repr__()
    mini_batch_k_means_0 = module_0.MiniBatchKMeans(batch_size=none_type_0, tol=
    var_7)
    var_8 = var_3.fit_predict(var_1, sample_weight=none_type_0)
    var_9 = var_6.fit(mini_batch_k_means_0)
    var_10 = var_1.rvs(var_9, var_5, size=var_0, random_state=var_1)
    var_11 = mini_batch_k_means_0.fit(var_1, var_3, var_7)
    var_12 = var_6.__call__(var_3, var_8)
    var_13 = module_2.fmin_bfgs(
        var_4, var_5, args=var_8, disp=var_6, retall=var_0, c1=var_12
    )
    var_14 = var_8.visit_ListComp(mini_batch_k_means_0)
    var_15 = var_9.fit_predict(none_type_0)
    mini_batch_k_means_1 = module_0.MiniBatchKMeans(
        init=var_14, verbose=var_0, init_size=var_8, n_init=var_3
    )
    var_16 = k_means_0.transform(var_3)
    var_17 = var_11.__sklearn_clone__()
    k_means_1 = module_0.KMeans(random_state=var_9, copy_x=var_14, algorithm=var_6)
```

Listing 7.4: The crash test generated by our approach

```
platform linux -- Python 3.10.14, pytest-8.2.0, pluggy-1.5.0
rootdir: /home/lucas/Documents/GitHub/pynguin-for-ML-libraries/results/
    sklearn_cluster_kmeans_subprocess_tensor_simple/28
collected 1 item

```

```
crash_test_4131952985032437545.py Fatal Python error: Floating point exception

Current thread 0x00007f8f639c7380 (most recent call first):
  File "/home/lucas/.conda/envs/pynguin-for-ML-libraries/lib/python3.10/site-
    packages/scipy/ndimage/_morphology.py", line 257 in _binary_erosion
  File "/home/lucas/.conda/envs/pynguin-for-ML-libraries/lib/python3.10/site-
    packages/scipy/ndimage/_morphology.py", line 520 in binary_dilation
  File "/home/lucas/.conda/envs/pynguin-for-ML-libraries/lib/python3.10/site-
    packages/scipy/ndimage/_morphology.py", line 1033 in binary_propagation
...
```

Listing 7.5: The output of the crash test generated by our approach

### 7.2.2 Code coverage comparison

In this section, we will compare the code coverage of the original approach with our approach using structured inputs. To do this, we will use the statistical tools presented in the chapter 6.

| Module | Baseline coverage | Structured inputs coverage | p-value | $\hat{A}_{12}$ |
|---|---|---|---|---|
| pandas.core.frame | 0.00 | 0.00 | 1.00 | Negligible (0.50) |
| pandas.io.parsers.reader | 0.33 | 0.36 | 0.17 | Small (0.40) |
| polars.dataframe.frame | 0.02 | 0.02 | 0.49 | Negligible (0.55) |
| polars.io.csv.functions | 0.36 | 0.38 | 0.12 | Small (0.38) |
| scipy.cluster.hierarchy | 0.12 | 0.13 | 0.39 | Small (0.44) |
| scipy.ndimage._fourier | 0.66 | 0.64 | 0.89 | Negligible (0.51) |
| sklearn.cluster._kmeans | 0.04 | 0.04 | 0.63 | Negligible (0.53) |

Table 7.5: Comparison between the code coverage of the baseline and our approach that uses structured input data

The table 7.5 shows a slight improvement in code coverage for some modules, but this improvement remains “small” or “negligible”, and the p-value is above our threshold of 0.05.

### 7.2.3 Line hit frequency graphs

In this section, we will stack the same type of graph presented in the new results to answer RQ1. This will make it easier to compare the different lines of code covered depending on the approach used.

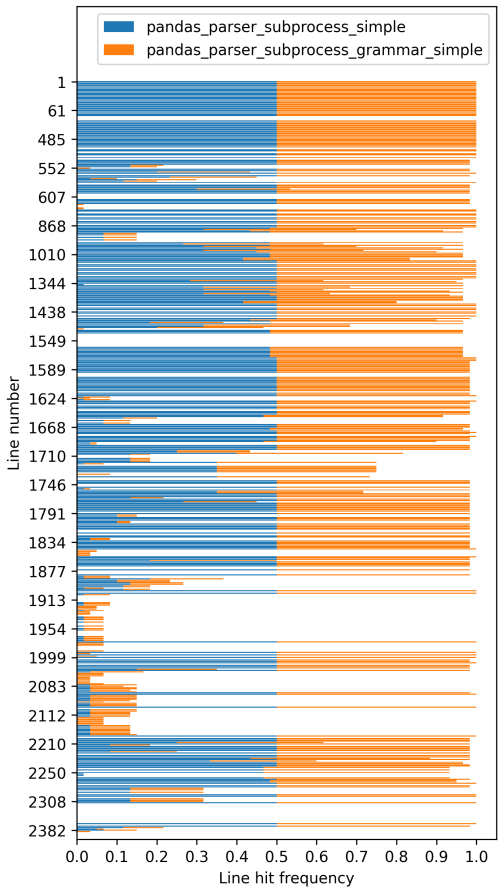



Figure 7.3: The line hit frequency of 60 generated test suites for the "pandas.io.parsers.readers" module using the traditional approach 30 times and our approach 30 times

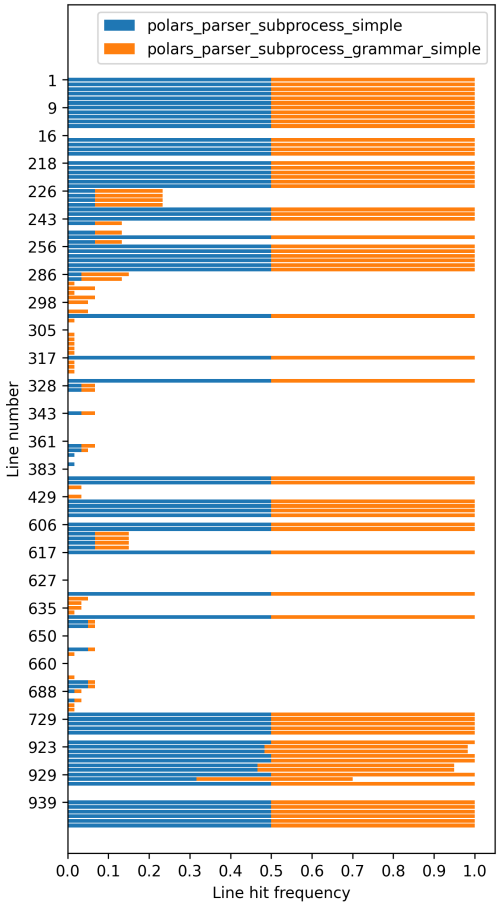



Figure 7.4: The line hit frequency of 60 generated test suites for the "polars.io.csv.functions" module using the traditional approach 30 times and our approach 30 times

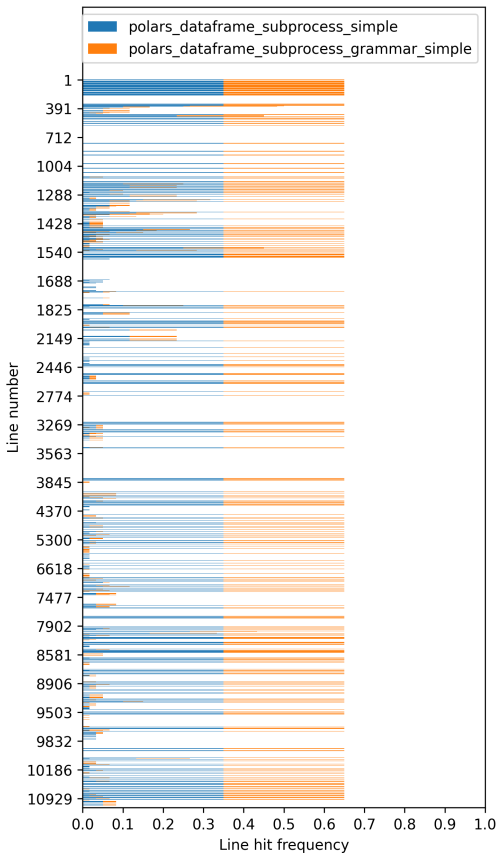



Figure 7.5: The line hit frequency of 60 generated test suites for the "polars.dataframe.frame" module using the traditional approach 30 times and our approach 30 times

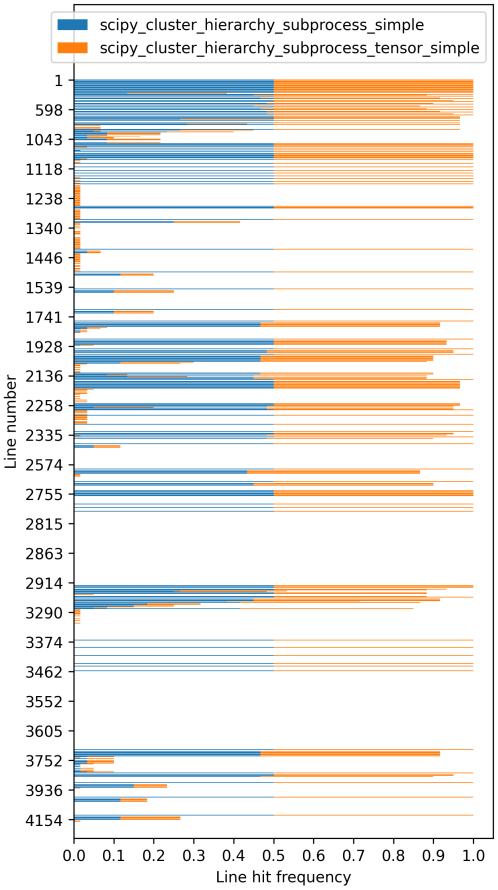



Figure 7.6: The line hit frequency of 60 generated test suites for the "scipy.cluster.hierarchy" module using the traditional approach 30 times and our approach 30 times

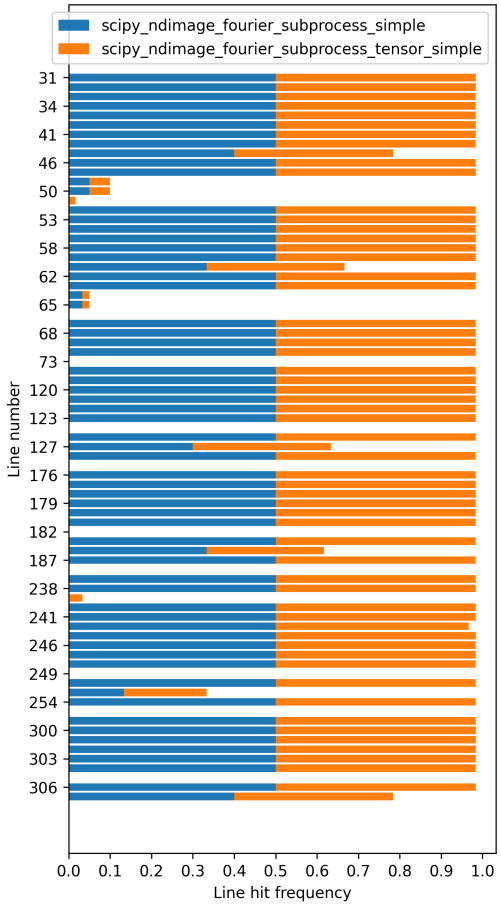



Figure 7.7: The line hit frequency hit of 60 generated test suites for the "scipy.ndimage._fourier" module using the traditional approach 30 times and our approach 30 times

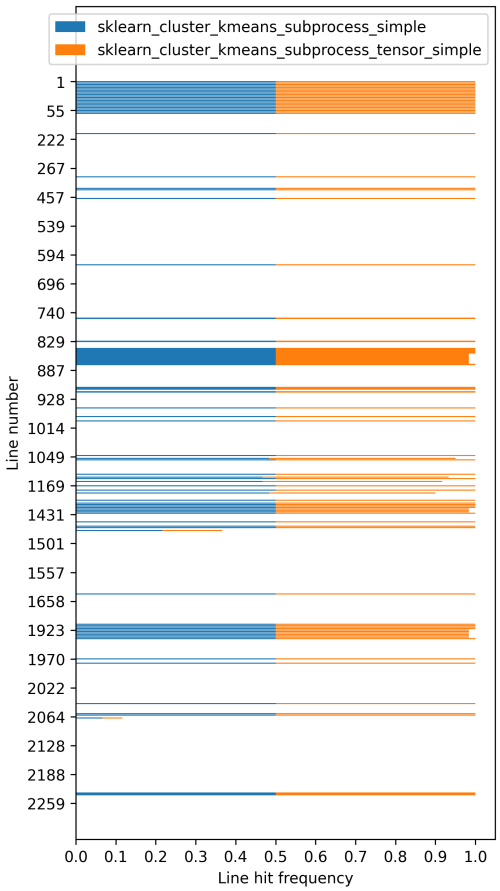



Figure 7.8: The line hit frequency of 60 generated test suites for the "sklearn.cluster._kmeans" module using the traditional approach 30 times and our approach 30 times

These graphs show that for most of the lines, half of the frequency is due to the original approach and the other half to our approach. This means that both approaches covered these lines the same number of times. However, it should be noted that there are sometimes differences between the lines covered, depending on the approach. If we focus on the figures 7.3, 7.4, 7.5, 7.6 and 7.7, we can see that some lines are covered only using our approach but very infrequently, while some other lines are covered only using the original approach but also very infrequently. There is also the figure 7.8 that shows almost no differences between the approaches.

### 7.2.4 Findings

Our results and analysis can be summarized as follows:

**Finding 3. Structured input data allows Pynguin to cover other parts of the MuT that are normally not covered using the original approach, but it does not necessarily increase code coverage.**

**Finding 4. The architecture using subprocesses allows error-revealing test cases to be created, revealing crashes caused by Segmentation fault, Floating-point exception or Out of memory bugs.**

## 7.3 Results related to the parameter tuning of Pynguin (RQ3)

In this section, we will describe the collected results to answer RQ3. This includes aggregated experiment data over several runs, statistical comparison between the approaches and graphs to represent better the code coverage.

### 7.3.1 Aggregated data of the experiment

First, we will describe the aggregated experiment data aimed at answering RQ3. They are shown in the table 7.6.

| Experiment | Coverage | Iterations | Total time | Search time | CTC | Success | Failure | T/O | Segfault | OOM | FPE | Other crashes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pandas_dataframe_subprocess_grammar_weights_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 30 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_subprocess_finetuned_simple | 0.38 | 13.63 | 680.98 | 622.67 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_subprocess_grammar_weights_simple | 0.38 | 18.40 | 670.21 | 614.66 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_subprocess_finetuned_grammar_simple | 0.39 | 12.53 | 684.36 | 618.42 | 2 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| pandas_parser_subprocess_finetuned_grammar_weights_simple | 0.42 | 13.10 | 695.92 | 627.45 | 16 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_dataframe_subprocess_grammar_weights_simple | 0.00 | 0.00 | 0.00 | 0.00 | 0 | 0 | 2 | 28 | 0 | 0 | 0 | 0 |
| polars_dataframe_subprocess_grammar_weights_no_assertion | 0.22 | 8.07 | 661.54 | 635.05 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess_finetuned_simple | 0.27 | 43.60 | 656.60 | 606.05 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess_grammar_weights_simple | 0.38 | 60.80 | 691.34 | 605.06 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess_finetuned_grammar_simple | 0.28 | 46.33 | 655.59 | 606.98 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| polars_parser_subprocess_finetuned_grammar_weights_simple | 0.27 | 45.87 | 655.20 | 606.87 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| scipy_cluster_hierarchy_subprocess_tensor_weights_simple | 0.23 | 13.73 | 692.72 | 597.54 | 7 | 29 | 1 | 0 | 0 | 0 | 0 | 0 |
| scipy_ndimage_fourier_subprocess_tensor_weights_simple | 0.69 | 186.43 | 612.73 | 602.02 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| sklearn_cluster_kmeans_subprocess_finetuned_simple | 0.05 | 15.13 | 650.48 | 620.64 | 0 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| sklearn_cluster_kmeans_subprocess_tensor_weights_simple | 0.05 | 19.87 | 646.27 | 615.48 | 22 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| sklearn_cluster_kmeans_subprocess_finetuned_tensor_simple | 0.05 | 14.90 | 650.75 | 620.41 | 3 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |
| sklearn_cluster_kmeans_subprocess_finetuned_tensor_weights_simple | 0.06 | 19.67 | 648.32 | 617.41 | 5 | 30 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 7.6: The aggregated experiment data collected to answer RQ3

In the table 7.6, we can see that the code coverages are still lower than the average code coverage of 68% found by Lukasczyk and Fraser [4] except for the "scipy.ndimage._fourier" module which is quite close.

Then, we can observe that, unlike in the the previous results, all the experiments were successful, and there were no failures except for the "pandas.core.frame" and "polars.dataframe.frame" modules. The causes of the failures of "pandas.core.frame" are the same as the ones explained in the previous results. Concerning the failures and timeouts of the “polars.dataframe.frame” module, they come from the assertion generation and deactivating this generation solves the problem; an investigation would be necessary to know the exact causes.

Finally, we can see that the Crash test count (CTC) is greater than 0 for the modules "pandas.io.-parsers.reader", "scipy.cluster.hierarchy" and "sklearn.cluster._kmeans", indicating that some tests revealing errors have been created. These tests are more frequent than in the new results to answer RQ2 and are always observed in experiments involving parameter tuning of our approach using structured inputs.

### 7.3.2 Code coverage comparison between the original approach and our approach using SOP parameter tuning

In this section, we will compare the code coverage of the original approach with our approach only using parameter tuning of the SOP. To do this, we will use the statistical tools presented in the chapter 6.

| Module | Baseline coverage | SOP coverage | p-value | $\hat{A}_{12}$ |
|---|---|---|---|---|
| pandas.io.parsers.reader | 0.33 | 0.38 | <0.01 | Large (0.25) |
| polars.io.csv.functions | 0.36 | 0.27 | <0.01 | Large (0.95) |
| sklearn.cluster._kmeans | 0.04 | 0.05 | 0.05 | Small (0.42) |

Table 7.7: Comparison between the code coverage of the baseline and our approach that uses optional parameter skipping

The table 7.7 shows that the SOP parameter tuning has positive or negative impacts depending on the module tested. Indeed, the module "pandas.io.parsers.reader" increased code coverage,

while the module "polars.io.csv.functions" decreased it. Moreover, the p-value is below our threshold of 0.05, and the effect size is large for these 2 modules.

### 7.3.3 Line hit frequency graphs of the original approach and the approach using SOP parameter tuning

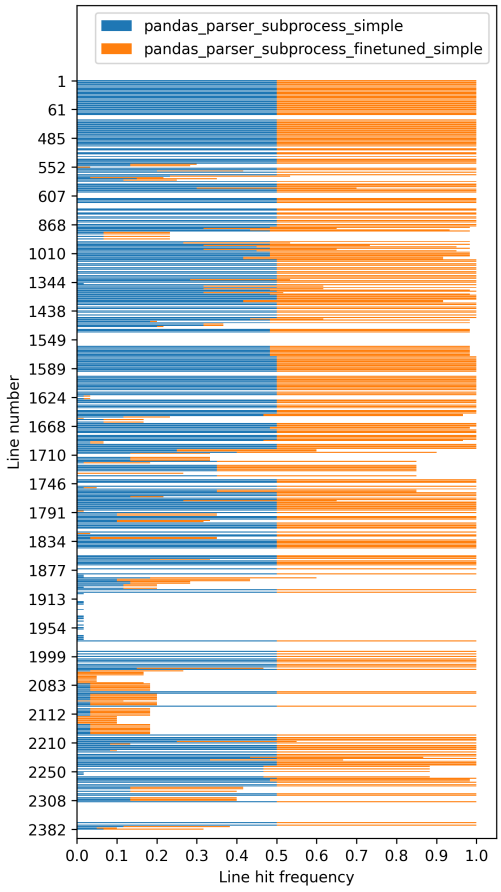



Figure 7.9: Line hit frequency of 60 generated test suites for the "pandas.io.parsers.readers" module using the traditional approach 30 times and our approach with optional parameter skipping 30 times

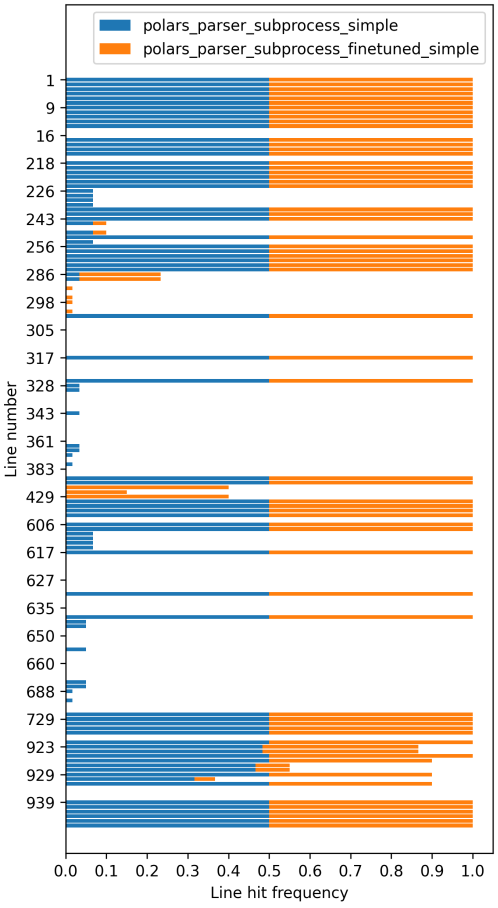



Figure 7.10: Line hit frequency of 60 generated test suites for the "polars.io.csv.functions" module using the traditional approach 30 times and our approach with optional parameter skipping 30 times

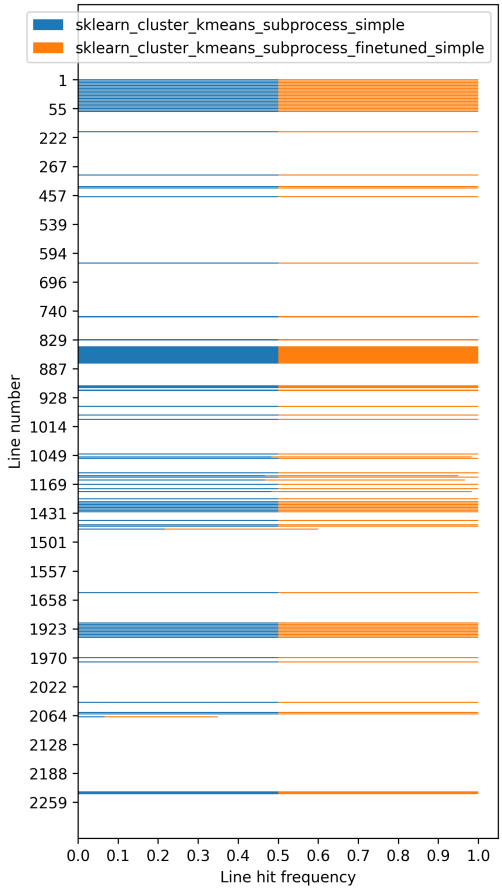



Figure 7.11: Line hit frequency of 60 generated test suites for the "sklearn.cluster._kmeans" module using the traditional approach 30 times and our approach with optional parameter skipping 30 times

The figures 7.9 and 7.10 show an increase in frequency for lines that were only covered using our approach in the results for RQ2. There is also the figure 7.11 that shows almost no differences between the approaches.

### 7.3.4 Code coverage comparison between the original approach and our approach using structured input data and SOP parameter tuning

In this section, we will compare the code coverage of the original approach with our approach using structured input data and SOP parameter tuning. To do this, we will use the statistical tools presented in the chapter 6.

| Module | Baseline coverage | SOP coverage | p-value | $\hat{A}_{12}$ |
|---|---|---|---|---|
| pandas.io.parsers.reader | 0.33 | 0.39 | <0.01 | Large (0.20) |
| polars.io.csv.functions | 0.36 | 0.28 | <0.01 | Large (0.93) |
| sklearn.cluster._kmeans | 0.04 | 0.05 | 0.01 | Small (0.40) |

Table 7.8: Comparison between the code coverage of the baseline and our approach that uses structured input data and optional parameter skipping

The table 7.8 shows that the SOP parameter tuning has also positive or negative impacts depending on the module tested when it is done on a version of Pynguin that uses structured input data. Indeed, the module "pandas.io.parsers.reader" increased code coverage, while the module

"polars.io.csv.functions" decreased it. Moreover, the p-value is below our threshold of 0.05, and the effect size is large for these 2 modules.

### 7.3.5 Line hit frequency graphs of the original approach and the approach using structured input data and SOP parameter tuning

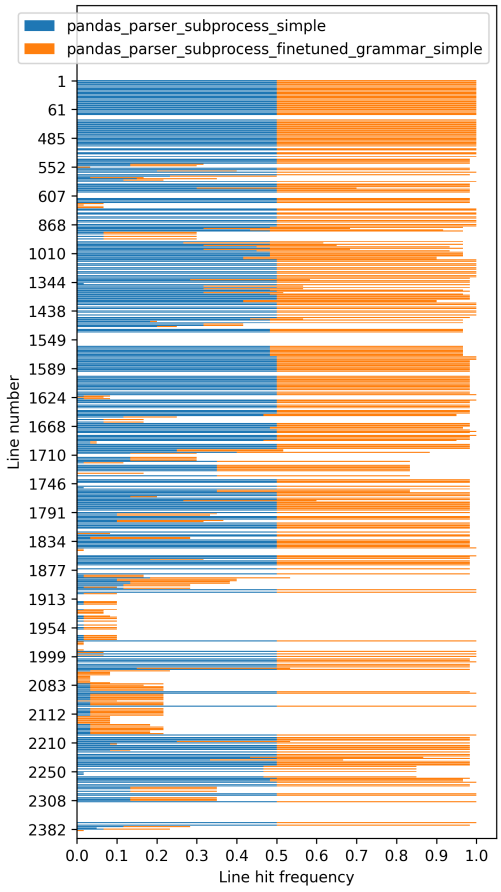



Figure 7.12: Line hit frequency of 60 generated test suites for the "pandas.io.parsers.readers" module using the traditional approach 30 times and our approach with structured input data and optional parameter skipping 30 times

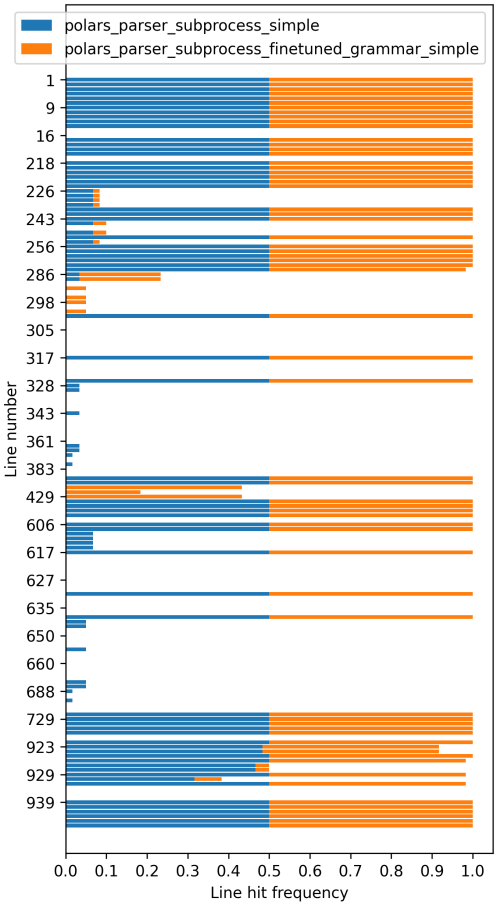



Figure 7.13: Line hit frequency of 60 generated test suites for the "polars.io.csv.functions" module using the traditional approach 30 times and our approach with structured input data and optional parameter skipping 30 times

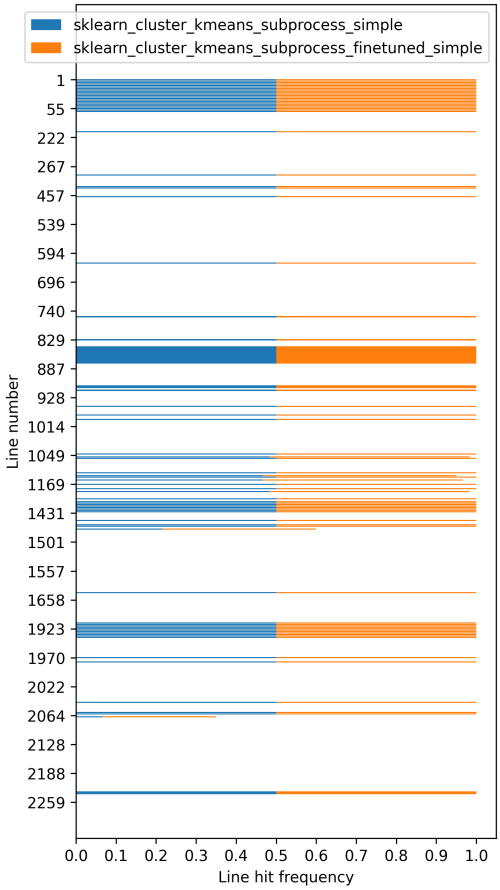



Figure 7.14: Line hit frequency of 60 generated test suites for the "sklearn.cluster._kmeans" module using the traditional approach 30 times and our approach with structured input data and optional parameter skipping 30 times

The figures 7.12 and 7.13 also show an increase in frequency for lines that were only covered using our approach in the results for RQ2. There is also the figure 7.14 that shows almost no differences between the approaches.

### 7.3.6 Code coverage comparison of the approach using structured input data and PW/PCW parameter tuning

In this section, we will compare the code coverage of the original approach with our approach using structured input data and PW/PCW parameter tuning. To do this, we will use the statistical tools presented in the chapter 6.

| Module | Baseline coverage | PW/PCW coverage | p-value | $\hat{A}_{12}$ |
|---|---|---|---|---|
| pandas.core.frame | 0.00 | 0.00 | 1.00 | Negligible (0.50) |
| pandas.io.parsers.readers | 0.33 | 0.38 | <0.01 | Large (0.28) |
| polars.dataframe.frame | 0.02 | 0.22 | <0.01 | Large (0.00) |
| polars.io.csv.functions | 0.36 | 0.38 | 0.03 | Medium (0.34) |
| scipy.cluster.hierarchy | 0.12 | 0.23 | <0.01 | Large (0.03) |
| scipy.ndimage._fourier | 0.66 | 0.69 | <0.01 | Large (0.26) |
| sklearn.cluster._kmeans | 0.04 | 0.05 | 0.83 | Negligible (0.51) |

Table 7.9: Comparison between the code coverage of the baseline and our approach that uses structured input data with increased weights and concrete weights

This table shows that the PW/PCW parameter tuning increased the code coverage in every case. This increase can be very small, as with module "sklearn.cluster._kmeans", where the p-value is 0.83 and the effect size is negligible, but it can also be large, as with module "polars.dataframe.frame", where the p-value is <0.01, the effect size is large and the code coverage increases by 20%.

### 7.3.7 Line hit frequency graphs of the original approach and the approach using structured input data and PW/PCW parameter tuning

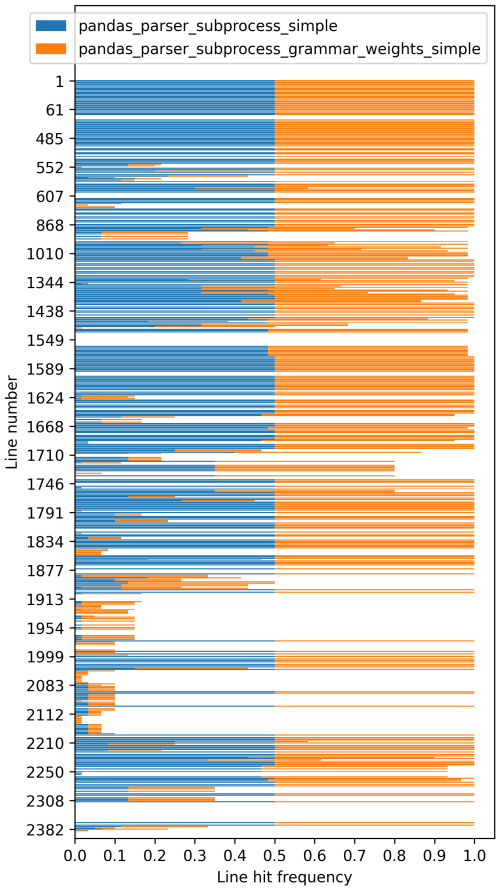



Figure 7.15: Line hit frequency of 60 generated test suites for the "pandas.io.parsers.reader" module using the traditional approach 30 times and our approach with structured input data and increased weights 30 times

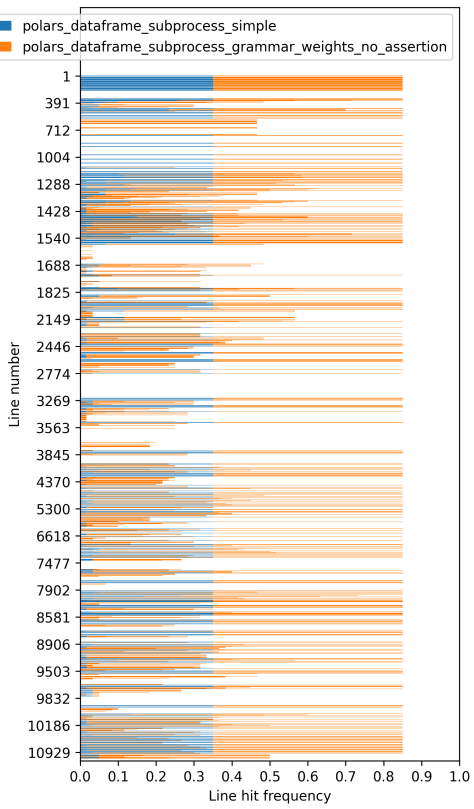



Figure 7.16: Line hit frequency of 60 generated test suites for the "polars.dataframe.frame" module using the traditional approach 30 times and our approach with structured input data and increased weights 30 times

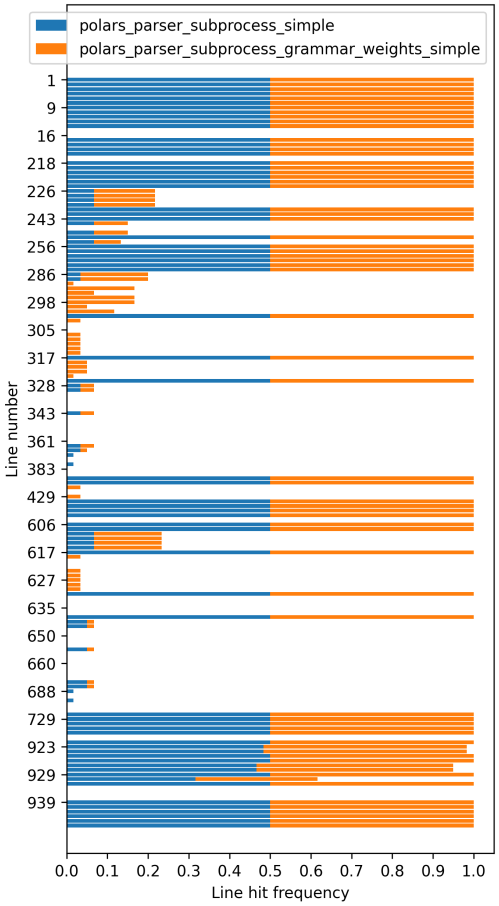



Figure 7.17: Line hit frequency of 60 generated test suites for the "polars.io.csv.functions" module using the traditional approach 30 times and our approach with structured input data and increased weights 30 times

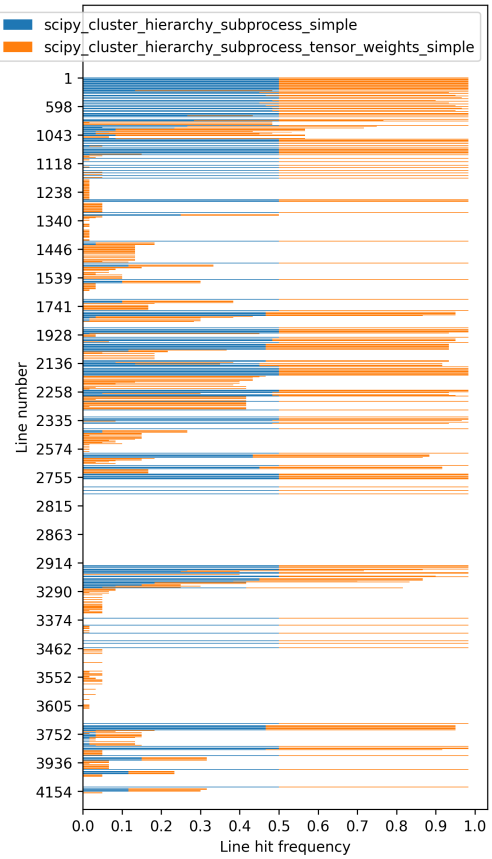



Figure 7.18: Line hit frequency of 60 generated test suites for the "scipy.cluster.hierarchy" module using the traditional approach 30 times and our approach with increased weights 30 times

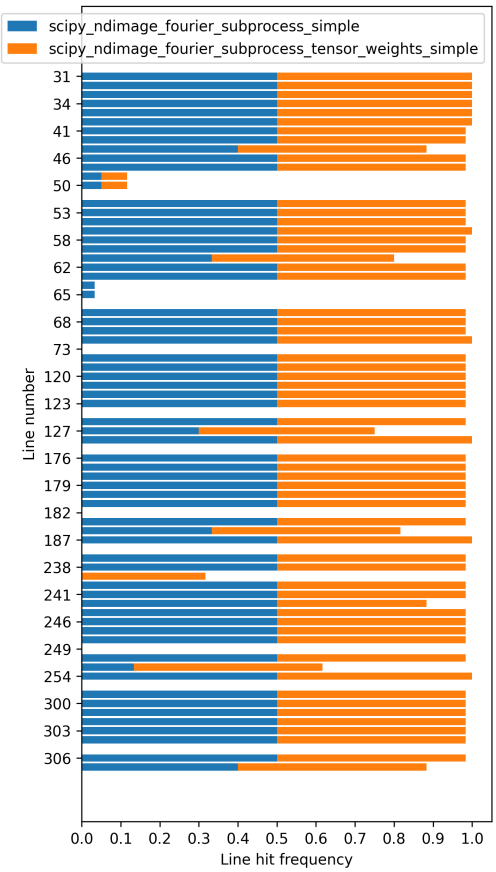



Figure 7.19: Line hit frequency of 60 generated test suites for the "scipy.ndimage._fourier" module using the traditional approach 30 times and our approach with increased weights 30 times

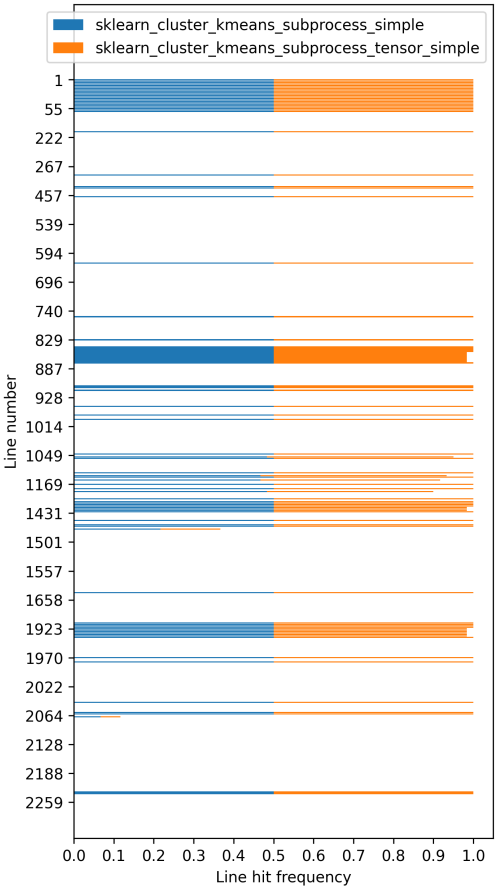



Figure 7.20: Line hit frequency of 60 generated test suites for the "sklearn.cluster._kmeans" module using the traditional approach 30 times and our approach with increased weights 30 times

The figures 7.15, 7.16, 7.17, 7.18, and 7.20 also show an increase in frequency for lines that were only covered using our approach in the results for RQ2. There is also the figure 7.20 that shows almost no differences between the approaches.

### 7.3.8 Code coverage comparison between the original approach and our approach using structured input data and SOP/PW/PCW parameter tuning

In this section, we will compare the code coverage of the original approach with our approach using structured input data and SOP/PW/PCW parameter tuning. To do this, we will use the statistical tools presented in the chapter 6.

| Module | Baseline coverage | SOP/PW/PCW coverage | p-value | $\hat{A}_{12}$ |
|---|---|---|---|---|
| pandas.io.parsers.reader | 0.33 | 0.42 | <0.01 | Large (0.06) |
| polars.io.csv.functions | 0.36 | 0.27 | <0.01 | Large (0.94) |
| sklearn.cluster._kmeans | 0.04 | 0.06 | <0.01 | Medium (0.32) |

Table 7.10: Comparison between the code coverage of the baseline and the use of optional parameter skipping and of weights and concrete weights

As in the section on SOP parameter tuning and the section on SOP parameter tuning with structured input data, parameter tuning can have a positive or negative impact, depending on

the MuT. However, this impact is more pronounced in this section than in the above-mentioned sections. Indeed, the improvements ranged from medium to large, with a maximum improvement of 9% and p-values $< 0.01$, but still with a large decrease of 9% in one case.

### 7.3.9 Line hit frequency graphs of the approach using structured input data SOP/PW/PCW parameter tuning

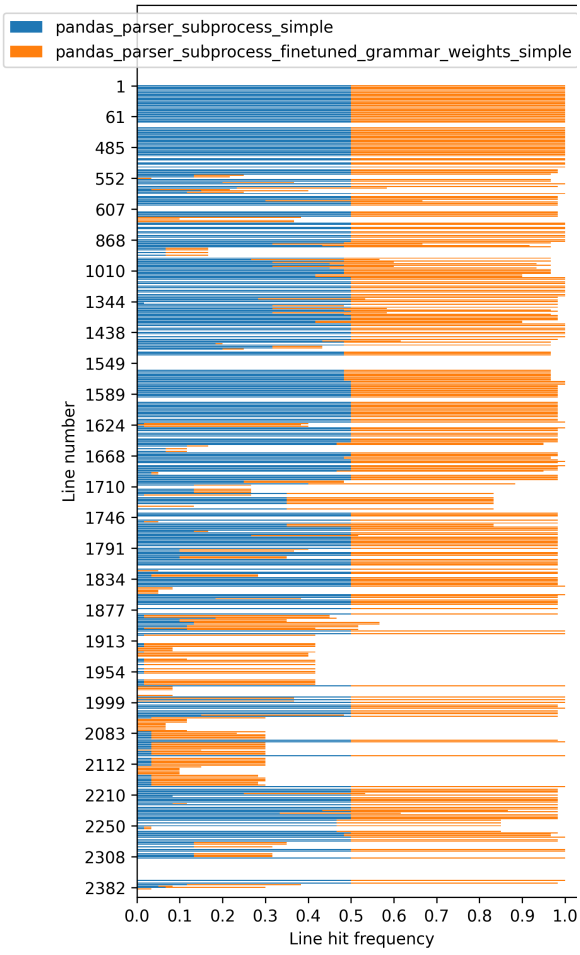



Figure 7.21: Line hit frequency of 60 generated test suites for the "pandas.io.parsers.readers" module using the traditional approach 30 times and our approach with optional parameter skipping and increased weights 30 times

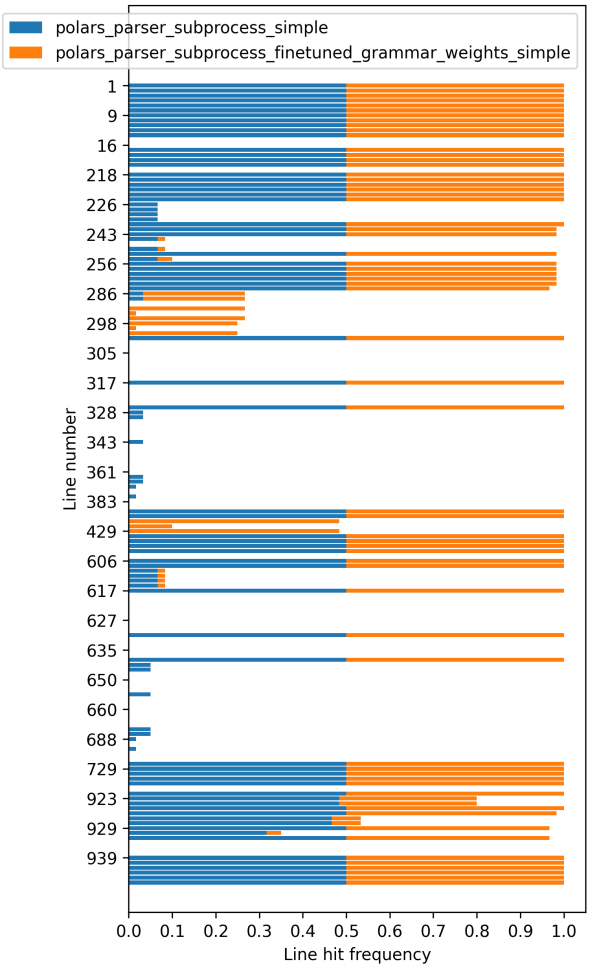



Figure 7.22: Line hit frequency of 60 generated test suites for the "polars.io.csv.functions" module using the traditional approach 30 times and our approach with optional parameter skipping and increased weights 30 times

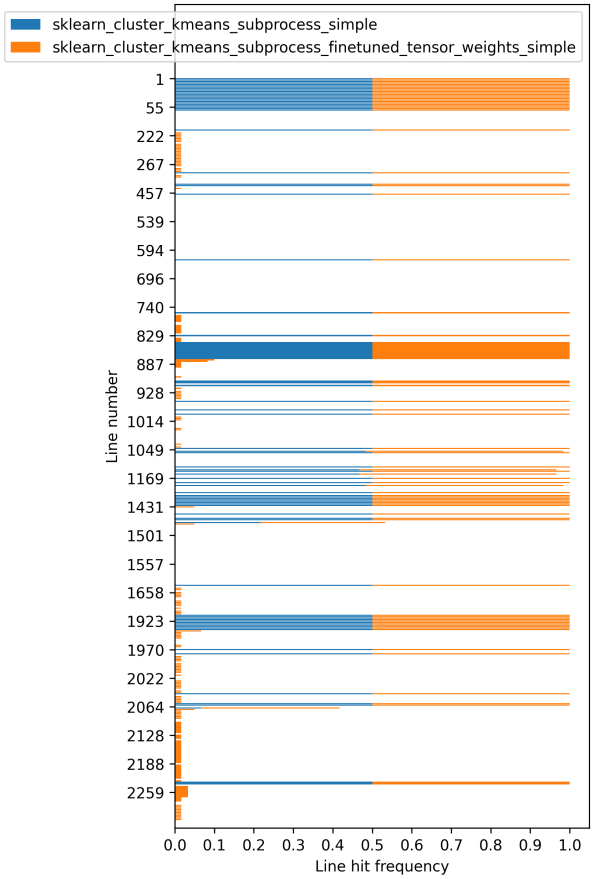



Figure 7.23: Line hit frequency of 60 generated test suites for the "sklearn.cluster._kmeans" module using the traditional approach 30 times and our approach with optional parameter skipping and increased weights 30 times

The figures 7.21, 7.22 and 7.23 show that this configuration is the one that increases the most the frequency of the lines that were only covered using our approach in the results for RQ2.

### 7.3.10 Findings

Our results and analysis can be summarized as follows:

**Finding 5. Parameter tuning can greatly affect the code coverage achieved by Pynguin and the lines of code that can be covered.**

# Chapter 8

# Discussion

This chapter will discuss our methodology and interpret the results obtained in the last chapter.

## 8.1 Testing machine learning libraries

This section will discuss the problems of testing machine learning libraries and whether classic automated test generation benefits these library types.

### 8.1.1 A data challenge

It is generally accepted that machine learning programs are very data-oriented. This kind of design poses a challenge for automated test generation tools because most of their algorithms and techniques have been tested on object-oriented designs and are not optimised for data-oriented designs.

For example, the results obtained to answer the first research question showed that Pynguin has difficulty generating test cases for some modules that require some specific values. After our qualitative analysis of the lines of code that were not covered, we found that these lines are usually optional features that are not used or lines that construct objects using structured data. If these are the lines that are not covered, this means that the lines that are covered are mainly lines that validate the parameters of the functions. Thus, the more parameters there are, the more difficult it is to avoid generating an invalid parameter and, therefore, the more difficult it is to reach these lines.

This observation is problematic but becomes even worse when a parameter requires a specific structure since the probability of randomly generating a specific structure is practically zero. In the second and the third research questions, we can observe this problem to some extent because there is an improvement in code coverage and line hit frequency on certain lines when using techniques that create structured data. Therefore, one additional difficulty is to generate data that is somewhat structured.

In addition, the creation of structured data is rather complicated because validation is not done bit by bit but at the end. As testers, we can easily consider the specifications of a function to enforce a structure on a string or a list before passing it as an argument. However, this is not the case for algorithms because they use exceptions to create data bit by bit.

For example, suppose a function takes a matrix as a parameter. In that case, an automated test case generation tool can more easily generate a valid input if this matrix is represented as a class rather than a list of lists. If it is just a class that takes a shape to be instantiated and where it is then possible to modify the elements via their indexes, it is easy for the algorithm to generate such a thing. However, if it is a list of lists, then the algorithm must generate a list of lists of the same size, which is highly unlikely due to the randomness of the generation algorithms. That is why, in RQ2 and RQ3, we have set up techniques to generate this kind of input all at once, even if it is only useful in certain cases.

### 8.1.2 Benefits of generating unit tests for machine learning libraries

Usually, automated test case generation tools are used to generate regression tests that try to cover as much code as possible. However, some functions in machine learning libraries use models designed to make errors in particular cases if this enables better generalization. So, it is worth asking whether generating test cases for these functions is useful. Given that, by slightly modifying a model, it is possible that a particular case will not run afterwards, this could be counterproductive. Moreover, these models require data with a specific structure and, as explained in the previous section, this is something that automated test generation tools fail to generate.

Nevertheless, regression test generation can still be useful for testing utility classes and functions used in algorithms. Indeed, for an algorithm to work, it is just as important that it uses utility functions and classes that are bug-free; otherwise, the algorithm will have bugs too.

## 8.2 Combining search-based test generation tools with fuzzing

This section will discuss the challenges and problems of combining search-based test generation tools with fuzzing.

### 8.2.1 Challenges

It is fairly easy to mix the 2 approaches, as fuzzing only comes into play during input data generation, and there is already a function where data is generated in automated test case generation algorithms. The challenge lies in combining the 2 approaches in a clean and extensible way. Generally, this is done by forking the automated test case generation tool and then modifying the input data generation function. Nevertheless, this design means that each time a new type of structured data input needs to be generated, a new fork needs to be created. That is why our approach proposes a system of plugins so that we can keep the same test generation tool while being able to add new input types.

### 8.2.2 Utilisation of testers' knowledge

Adding fuzzing techniques to an automated test case generation algorithm makes incorporating testers' knowledge into the algorithm possible. Since fuzzing usually requires data to be generated in a specific way, testers need to be able to provide their knowledge about the data to the algorithm. This can be done using a plugin system as in RQ2. However, it has the drawback of having testers who must create these plugins. In addition, it would be possible to improve this plugin system so that testers could add new assertions to check invariants.

### 8.2.3 When is it better?

Our approach is not always better. For example, when testing a particular function, it is probably always better to use fuzzing to find bugs. Indeed, our approach first requires a call to the right function to be generated before inputs can be fuzzed, so it is less effective when testing only one function. When creating regression tests, using the classic approach to generating test cases is probably enough, as creating structured inputs is often more time-consuming than creating random values.

However, when it is required to create regression test cases for classes or functions that need structured data, our approach is usually better since it can be configured to use the right structured input data. Our approach can also find bugs that neither traditional automated test generation tools nor fuzzers could find. Indeed, if there exists an oracle that can detect failure, our approach could find failures caused by a sequence of function calls with different structured input data. Since it is a sequence of calls that causes bugs, fuzzers would not be able to find them as they only focus on inputs, and since it is bugs that require structured data, traditional automated test generation tools would not find them either.

Our approach could also be beneficial if combined with the traditional approach. Indeed, since the code covered by the 2 approaches is often different, combining the tests generated by the traditional approach with those of our approach could create test suites with better code coverage.

## 8.3 Architecture using subprocess

Based on our study, we can say that subprocesses are almost necessary to test Python libraries using C-extension modules. Indeed, bugs in these C-extension modules could cause the Python interpreter to crash instead of returning an exception to the Python program. This behaviour was highlighted by the results of chapter 4, where the interpreter crashed a lot of times due to Segmentation fault or Floating-point exception. There could potentially be functions where it is normal for them to crash the Python interpreter if their preconditions are not respected; however, this is very unusual and is not the recommended behaviour.

In addition, if used properly, subprocesses can be used as an oracle to create error-revealing test cases. For example, in our study, we do not ignore crashes; we create test cases to reproduce them.

However, as explained in chapter 6, the architecture using subprocesses is much slower due to the launch of subprocesses and data transfer. By comparing the new results from chapter 6 with those from chapter 4, we can see that this architecture can generate up to 40 times fewer iterations than the classic architecture. Since search-based approaches rely heavily on the number of iterations, this can make a big difference. Nevertheless, it is always better to have an architecture that can generate test cases rather than one that will crash at the first bug from C.

Finally, we can mention the fact that this is an architecture that is probably only suitable for Python. Indeed, one of Python's distinctive characteristics is that a large part of its ecosystem contains C-extension modules. Since it is these C-extension modules that create the need for this architecture, this is a type of architecture that is potentially less useful in other languages. For example, if we look at Java, there is also the JNI for executing C code that could create the same kind of crashes, but most of the Java ecosystem does not use it.

## 8.4 Problems with code coverage

One of the big problems with code coverage is that it does not offer a way of comparing approaches that try to cover different parts of the code in priority. Indeed, since this metric only compares the part of the code that has been executed at the file level, 2 code coverage values may be similar when, in fact, the code covered is very different. For example, our approach mostly tries to follow the happy path as far as possible by creating structured inputs, whereas the classic approach will stumble across more code that validates badly structured inputs. This resulted in having 2 approaches that covered different parts of the code, even though the code coverage had more or less the same value.

To solve part of the problem, it is possible to use other metrics at the line level, such as the Line Hit Frequency. However, this is much harder to compare because there is as many values as lines of code.

## 8.5 Problems with default settings

Our results showed that the default parameters of Pynguin and its plugins were sometimes not adapted to generate test cases for machine learning libraries automatically. However, these parameters are fairly easy to tune with just a little knowledge of the MuT. For example, depending on the number of optional parameters in the MuT, it is easy to increase or decrease the probability of not passing an argument that will replace them so as not to test too many optional parameters simultaneously.

## 8.6 Threats to validity

In this section, we will discuss the threats to the validity of our study.

### 8.6.1 Threats to construct validity

To compare the approaches, we used several metrics. Some of these are very well established, such as the branch coverage, the number of iterations or the running time, but we also used metrics

such as the number of crashes, the number of crash tests generated or the line hit frequency. These last metrics are not often used to compare automated test case generation approaches, but they are often used in the fuzzing world or to obtain better information on a test suite. In addition, we have chosen the running time as a stopping condition to have a metric that fairly compares approaches that make a lot of fast iterations with approaches that make fewer slower iterations.

### 8.6.2 Threats to internal validity

Although we have made all possible efforts to limit the number of bugs in our improvements, we cannot guarantee that they will be bug-free. In addition, our improvements have been built on a Pynguin that may also have bugs. Our results could, therefore, be influenced by potential bugs in Pynguin or in our improvements. To minimize this risk, a test suite and a linter have been set up to find as many bugs as possible, and the Pynguin code and our improvements have been made open-source so that other researchers can find bugs and replicate our experiments.

### 8.6.3 Threats to external validity

This threat is crucial in our study as we deliberately chose a set of 7 modules from 4 libraries on which our approach could work. The reason is that the aim was to demonstrate that using structured data could impact certain well-chosen modules. Further experiments would be necessary to improve our confidence that our results could be generalized to other modules using the same structured data or other modules using different structured data. We have restricted our benchmark because we had a lot of configurations to test, and the cost of running an experiment was very high.

### 8.6.4 Threats to conclusion validity

These threats are caused by the fact that automated test case generation algorithms use randomness. To minimize these threats, we ran each experiment 30 times with different random seeds and performed a statistical analysis to compare the approaches as recommended by Arcuri and Briand [60] in their guideline for such comparisons. Therefore, we used the Mann-Whitney U-test to calculate our p-values and the Vargha-Delaney $\hat{A}_{12}$ effect size to estimate the significance of our results.

# Chapter 9

# Conclusion

In this chapter, we will look again at the three research questions raised in chapter 3 and summarise the contribution of our study. Then, we will discuss things that could be improved in our research and ideas for further work.

## 9.1 Summary

In this section, we will use the results of chapter 5 and chapter 7 and the discussion in chapter 8 to summarise our findings that answer the research questions.

**RQ1. How effective is Pynguin at automatically generating test cases for machine learning libraries?**

In chapter 5, we showed that Pynguin crashed most of the time when we were trying to generate tests on machine learning modules due to bugs, limitations and its architecture. This means that in its current state, Pynguin has problems generating tests for machine learning modules. Although this research question was focused on just 2 modules, the crashes detected are typical of classic C programming errors. Hence, it is likely that a study on more different modules would come up with the same errors. The analysis of the results concerning the other research questions in chapter 5 shows precisely this behaviour in different types of modules.

Nevertheless, our study went further by correcting the bugs and limitations as well as trying to see the effectiveness of Pynguin after a change of architecture allowing tests to be run in subprocesses. This made it possible to eliminate crashes and show that the average code coverage for this architecture was 37%, compared with the average code coverage of 68% found by Lukasczyk and Fraser [4] in their paper introducing Pynguin. In addition, after a manual analysis of the results, we were able to conclude that the lines of code not covered concerned optional features but also instructions for creating dataframes, which showed that the happy path was sometimes not followed to the end.

To summarise our answer to the RQ1, we can say that the current version of Pynguin is not able to automatically generate tests for machine learning libraries, within the limits of the one we tested.

**RQ2. To what extent does using structured input data improve the efficiency of test case generation in Pynguin for machine learning libraries?**

To answer this question, we set up a system of plugins and implemented several plugins to generate structured inputs based on grammar or just code. These plugins are specialised in generating CSVs, dataframes and tensors, and have been used on modules that could potentially benefit from them.

In chapter 5, we were able to collect data showing some differences between the original approach and our approach using plugins, but given the large number of crashes, these results were primarily useful to identify the type of crashes.

However, in chapter 7, we were able to obtain results by reusing our architecture using subprocesses. These results showed that there was an improvement ranging from negligible to small, with a maximum code coverage increase of 3% and with p-values ranging from 0.12 to 0.89. Note that these results were obtained with the plugins' default settings, which means that they are only used in very specific cases. Nevertheless, we also noticed that although the code coverage did not change much, our plugin system was able to cover certain lines of code far into the happy path and inaccessible by the original approach. These results have also shown that it is possible to generate error-revealing test cases based on crashes that can be detected by the new architecture.

To summarise our answer to the RQ2, we can say that the use of structured input data with fairly strict default parameters does not necessarily increase the code coverage achieved by Pynguin, but it does allow other parts of the MuT to be explored.

**RQ3. How much does parameter tuning improve the test cases generated by Pynguin?**

To answer this question, we tuned some Pynguin parameters that handle the probability of skipping optional function parameters and the probability of using plugins in more cases.

In chapter 5, we were able to collect data showing some differences between the original approach and our approach using plugins, but given the large number of crashes, these results were primarily useful to identify the type of crashes.

However, in chapter 7, we were able to obtain results by reusing our architecture using subprocesses. The results showed that modifying the SOP parameter of Pynguin could have either a very positive or a very negative impact. In one case, it increased coverage by 4% with a p-value $< 0.01$; in another, it decreased coverage by 9% with a p-value $< 0.01$; and in yet another, it only increased coverage by 1% with a p-value of 0.05. Furthermore, by modifying the plugin weights, we obtained improvements ranging from negligible to large, with a maximum code coverage increase of 20% and p-values $< 0.05$ in almost all cases. In this case, changing this parameter had a positive effect on all the MuT, with an increase in code coverage in all cases. Finally, we tried to combine the different configurations that we tested and came up with results similar to those of the SOP parameter tuning, except that they were even more accentuated. In these results, the improvements ranged from medium to large, with a maximum code coverage increase of 9% and p-values $< 0.01$, but still with a large decrease of 9% in one case. These results also showed that using parameter tuning in experiments with structured data created more error-revealing test cases than both the original approach and our approach without the use of parameter tuning. In addition, these results raised the question of combining the tests generated by the traditional approach with the tests generated by our approach. Since they test different parts of the MuT, their combination could improve the final code coverage.

To summarise our answer to the RQ3, we can say that the use of parameter tuning can greatly improve the code coverage achieved by Pynguin on certain modules, but depending on the parameters, it can also greatly reduce it.

## 9.2 Future work

One of the big improvements that could be made to our approach is to redesign the architecture using subprocesses in a more fundamental way to significantly reduce the amount of data that needs to be transferred from one process to another, thus improving performance. The problem with the current architecture using subprocesses is that it is very slow, which means it can only make a small number of iterations in a given period. This means that the genetic algorithm cannot make many mutations, and so the coverage code remains fairly low. However, another possibility would just be to test our approach over a longer or shorter search time to see how it affects our results.

Another potential study would be to test our approach on more modules and create new plugins capable of handling more different structured inputs. It does not necessarily have to be only machine learning libraries; other types of libraries can also be used as long as they use structured inputs.

Finally, a last area of research would be to combine more automated test generation techniques with fuzzing techniques, particularly in Python. Indeed, due to its highly dynamic nature and its ecosystem which uses a lot of C code for performance reasons, we are convinced that certain bugs can only be discovered with a sequence of instructions with particular inputs. This is what we have already partially found in our results in chapter 7, but it could be extended to test more critical libraries to directly find bugs that are potentially more serious.

# Bibliography


[1] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[2] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[3] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[4] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[5] usagitoneko97. GitHub - usagitoneko97/klara: Automatic test case generation for python and static analysis library — github.com. `https://github.com/usagitoneko97/klara`, 2021. [Accessed 13-05-2024].

[6] Nicolas Erni, Al-Ameen Mohammed Ali Mohammed, Christian Birchler, Pouria Derakhshanfar, Stephan Lukasczyk, and Sebastiano Panichella. Sbft tool competition 2024–python test case generation track. *arXiv preprint arXiv:2401.15189*, 2024.

[7] Song Wang, Nishtha Shrestha, Abarna Kucheri Subburaman, Junjie Wang, Moshi Wei, and Nachiappan Nagappan. Automatic unit test generation for machine learning libraries: How far are we? In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1548–1560. IEEE, 2021.

[8] Mitchell Olsthoorn, Arie van Deursen, and Annibale Panichella. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1224–1228, 2020.

[9] Lucas Berg. Improving automated unit test generation for machine learning libraries using structured input data - results. *Zenodo*, June 2024. doi:10.5281/ZENODO.11480203. URL `https://zenodo.org/doi/10.5281/zenodo.11480203`.

[10] Berg. GitHub - BergLucas/pynguin-for-ML-libraries — github.com. `https://github.com/BergLucas/pynguin-for-ML-libraries`, 2024. [Accessed 22-05-2024].

[11] Winston W Royce. Managing the development of large software systems (1970). 2021.

[12] Herb Krasner. The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, pages 1–46, 2021.

[13] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, pages 177–182. IEEE, 2016.

[14] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th international conference on software engineering*, pages 72–82, 2014.

[15] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*, pages 560–564. IEEE, 2015.

[16] Gunel Jahangirova and Valerio Terragni. Sbft tool competition 2023-java test case generation track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 61–64. IEEE, 2023.

[17] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 367–381. Springer, 2008.

[18] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*, pages 95–111. Springer, 2011.

[19] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194, 2010.

[20] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[21] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.

[22] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[23] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104:236–256, 2018.

[24] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings 7*, pages 93–108. Springer, 2015.

[25] Pouria Derakhshanfar, Xavier Devroey, Annibale Panichella, Andy Zaidman, and Arie van Deursen. Generating class-level integration tests using call site information. *IEEE Transactions on Software Engineering*, 49(4):2069–2087, 2022.

[26] Andrea Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1):1–37, 2019.

[27] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[28] Andrea Arcuri. Test suite generation with the many independent objective (mio) algorithm. *Information and Software Technology*, 104:195–206, 2018.

[29] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[30] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[31] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*, pages 9–24. Springer, 2020.

[32] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. Guess what: Test case generation for javascript with unsupervised probabilistic type inference. In *International Symposium on Search Based Software Engineering*, pages 67–82. Springer, 2022.

[33] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE, 2011.

[34] Dmitry Ivanov, Nikolay Bukharev, Alexey Menshutin, Arsen Nagdalian, Gleb Stromov, and Artem Ustinov. Utbot at the sbst2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 34–35. IEEE, 2021.

[35] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. Sbst tool competition 2021. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 20–27. IEEE, 2021.

[36] Dmitry Ivanov, Alexey Menshutin, Denis Fokin, Yury Kamenev, Sergey Pospelov, Egor Kulikov, and Nikita Stroganov. Utbot java at the sbst2022 tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing*, pages 39–40, 2022.

[37] Limitation - klara 0.6.3 documentation — klara-py.readthedocs.io. `https://klara-py.readthedocs.io/en/latest/limitation.html`, 2021. [Accessed 21-05-2024].

[38] David R MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[39] Ghostwriting tests for you - 2014; Hypothesis 6.102.4 documentation — hypothesis.readthedocs.io. `https://hypothesis.readthedocs.io/en/latest/ghostwriter.html#a-note-for-test-generation-researchers`, 2024. [Accessed 21-05-2024].

[40] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[41] Hyunguk Yoo and Taeshik Shon. Grammar-based adaptive fuzzing: Evaluation on scada modbus protocol. In *2016 IEEE International conference on smart grid communications (SmartGridComm)*, pages 557–563. IEEE, 2016.

[42] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of {TLS} implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, 2015.

[43] Jingbo Yan, Yuqing Zhang, and Dingning Yang. Structurized grammar-based fuzz testing for programs with highly structured inputs. *Security and Communication Networks*, 6(11): 1319–1330, 2013.

[44] Houssem Ben Braiek and Foutse Khomh. On testing machine learning programs. *Journal of Systems and Software*, 164:110542, 2020.

[45] Zhixin Qi, Hongzhi Wang, Jianzhong Li, and Hong Gao. Impacts of dirty data: and experimental evaluation. *arXiv preprint arXiv:1803.06071*, 2018.

[46] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment*, 9(12):948–959, 2016.

[47] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning. *arXiv preprint arXiv:1711.01299*, 2017.

[48] Nick Hynes, D Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ml data sets. In *NIPS MLSys Workshop*, volume 1, 2017.

[49] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 132–142, 2018.

[50] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[51] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[52] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[53] Houssem Ben Braiek and Foutse Khomh. Deepevolution: A search-based testing approach for deep neural networks. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 454–458. IEEE, 2019.

[54] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 120–131, 2018.

[55] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 146–157, 2019.

[56] Roger B Grosse and David K Duvenaud. Testing mcmc code. *arXiv preprint arXiv:1412.5218*, 2014.

[57] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[58] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[59] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18:594–623, 2013.

[60] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[61] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024. URL `https://www.fuzzingbook.org/`. Retrieved 2024-01-18 17:28:37+01:00.

[62] TypeError: unsupported operand type(s) for |: 'NoneType' and 'NoneType' · Issue #53 · se2p/pynguin — github.com. `https://github.com/se2p/pynguin/issues/53`, 2023. [Accessed 22-05-2024].

[63] Berg. Add support for parsing module aliases by BergLucas · Pull Request #58 · se2p/pynguin — github.com. `https://github.com/se2p/pynguin/pull/58`, 2024. [Accessed 22-05-2024].

[64] Cannot generate testcase with import statement e.g. 'numpy' · Issue #35 · se2p/pynguin — github.com. `https://github.com/se2p/pynguin/issues/35`, 2022. [Accessed 22-05-2024].

[65] Berg. Fix signature and private module errors that comes from C extension modules by BergLucas · Pull Request #60 · se2p/pynguin — github.com. `https://github.com/se2p/pynguin/pull/60`, 2024. [Accessed 22-05-2024].

[66] Berg. GitHub - BergLucas/pynguin at main-fixed — github.com. `https://github.com/BergLucas/pynguin/tree/main-fixed`, 2024. [Accessed 22-05-2024].

[67] M McKerns and M Aivazis. Pathos: a framework for heterogeneous computing. *See http://trac. mystic. cacr. caltech. edu/project/pathos*, 2010.

[68] Michael M McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael AG Aivazis. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056*, 2012.