Ruprecht-Karls-Universität Heidelberg
Institut für Informatik
Lehrstuhl für Parallele und Verteilte Systeme

Masterarbeit

Automated Checking of Clean Code Guidelines for Python

Name: Enrico Kaack
Matrikelnummer: 3534472
Betreuer: Prof. Dr. Artur Andrzejak
Datum der Abgabe: 31.10.2020

Ich versichere, dass ich diese Master-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Heidelberg, 31.10.2020

Zusammenfassung

Die automatische Erkennung von Verstößen gegen die Clean Code Prinzpien ermöglicht es Entwicklern, die Verständlichkeit und Lesbarkeit von Quellcode leicht zu verbessern. Wir führen eine Systematik ein, bei der wir die Clean Code Regeln basierend auf der Komplexität automatischer Erkennungsverfahren einordnen. Darüber hinaus entwerfen und implementieren wir eine Plattform, die das Orchestrieren verschiedener, automatischer Erkennungsverfahren ermöglicht. Als nächstes trainieren wir verschiedene Verfahren des maschinellen Lernens zur Erkennung von Code Mustern, die Clean Code Regeln verletzen: Einen Support-Vector-Klassifikator (1), der die Code Muster nicht zufriedenstellend erkennen kann. Ein auf LSTMs basierendes neuronales Netzwerk (2), welches alle anderen Modelle übertrifft, aber empfindlich gegenüber weniger Trainingsdaten ist. Einen Random-Forest Klassifikator (3), der moderate Ungleichheit in der Klassenhäufigkeit in den Trainingsdaten kompensieren kann und einen Gradient-Boosting Klassifikator (4), der nicht empfindlich gegenüber Ungleichheit in der Klassenhäufigkeit ist. Durch das Hinzufügen von unbekannten Problemvariationen untersuchen wir, ob die Modelle die Struktur des zugrunde liegenden Problems gelernt haben. Unsere Ergebnisse zeigen, dass die Modelle die unbekannten Variationen schlecht erkennen können und nur das Muster statt der Struktur des Problems gelernt haben. Zur Verbesserung schlagen wir eine Veränderung der Kodierung des Codes für die Modelle vor, die einen größeren Fokus auf die Struktur des Codes legt. Außerdem empfehlen wir das Hinzufügen von weiteren Variationen durch händisch gesammelte Daten.

Abstract

The automatic detection of clean code violations enables developers to improve understandability and readability of source code more easily. We propose a novel taxonomy to categorise clean code rules based on the assumed level of complexity for the implementation of automated checkers. Furthermore, we design and implement a platform to orchestrate automated rule checkers and find that it offers a simple way of extension compared to other platforms. Next, we train different machine learning models to detect code patterns that violate clean code rules: A support vector classifier (1) that is not able to detect violations with acceptable performance. An LSTM-based neural network (2) that outperforms all other models, but is sensitive towards less training data. A random forest classifier (3) that can handle a moderate class imbalance in the training data, and a gradient boosting classifier (4) that is insensitive towards class imbalance. By introducing unseen variations, we further investigate if the models have learnt the structure of the problem. We find that the models do not perform well on unseen problem variations and only learn the problem pattern. We suggest improvements by changing the encoding to emphasise code structure and by adding hand-labelled data with more variations.

Acknowledgement

I want to thank my supervisor Professor Dr. Artur Andrzejak for his ongoing guidance, support, and feedback during this master thesis. Furthermore, I want to show my gratitude to numerous people that gave me helpful feedback on this work.

Thank you.

Contents

1 Introduction

The rise of information technology with computers over the last decades allowed humanity to process data and solve problems with unprecedented performance. Computers require instructions in order to fulfil their purpose. The common languages between humans and computers are programming languages. Software developers use these languages to write source code that instructs computers to process data and solve problems. While a machine only follows the instructions described by the source code, the developers have to understand the purpose of the instructions.

Nowadays, software has become a vast market with a revenue of over 450 billion U.S. dollars in 2019 [1, 2]. Creating new code often requires reading and understanding preexisting code. As a consequence, it is essential to make code as readable and understandable as possible.

The objectives of the clean code principles are a high level of readability and understandability of the source code [3]. Thus, the clean code principles comprise a set of rules and recommendations about writing code that is easy to read and understand. For computers, it is irrelevant whether or not a human can understand the code since they only execute the instructions. However, any developer who reads code that is written in accordance to the clean code principles profits from it.

In Figure 1.1, the first listing shows an implementation of an algorithm without paying attention to clean code rules. In the second listing, the same algorithm is implemented following the clean code rules such as having one purpose per function or method calls in if-conditions. Executing the code will yield the same result. However, it is far simpler to understand the second than the first code. Reading the clean `get_students_eligible_for_exam` function lets the developer immediately understand the two eligibility criteria for the exam. Comments like in the first example are not even necessary, since the code is self-describing. Consequently, using or modifying the second code is much easier and less time-consuming.

An automated tool for checking violations of the clean code rules can preserve the readability and understandability of the code for future modifications. Integrated into

```python
class Student():
    def __init__(self, name):
        self.name = name
        self.exercises = []

def get_students_eligible_for_exam(students):
    # filter for students with enough points
    enough_points = []
    for student in students:
      points = 0
      for e in student.exercises:
        points += e
      if points >= 60:
        enough_points.append(student)

    # only students with at least 8 completed exercise
    eligible = []
    for student in enough_points:
      if len(student.exercises) >= 8:
        eligible.append(student)
    return eligible
```

```python
class Student():
    def __init__(self, name):
        self.name = name
        self.exercises = []

class ExamEligibilityChecker():
    MIN_NUMBER_OF_POINTS = 60
    NUMBER_OF_SOLVED_EXERCISES = 8

    def has_student_enough_points(student):
        total_points = 0
        for exercise in student.exercises:
            total_points += exercise
        return total_points >= ExamEligibilityChecker.MIN_NUMBER_OF_POINTS

    def has_student_enough_completed_exercises(student):
        return len(student.exercises) >= ExamEligibilityChecker.NUMBER_OF_SOLVED_EXERCISES

def get_students_eligible_for_exam(students):
    eligible_students = []
        for student in students:
        if ExamEligibilityChecker.has_student_enough_points(student)
            and ExamEligibilityChecker.has_student_enough_completed_exercises(student):
            eligible_students.append(student)
    return eligible_students
```

Figure 1.1: Motivating example code that checks if a student is eligible for an exam. The first listing shows a smaller, more chaotic implementation. The second example applies some clean code rules like a single purpose per function or method calls in if-conditions. Further clean code improvements are possible.

a build pipeline, it could reject code changes if necessary and act as a quality gate. Furthermore, it can teach students about clean code rules. In practical works, an automated checker could assess the students' code and recommend improvements. The students can change the code and get immediate feedback if they improved the code. Contrary to manual code review, the automated checkers allow a consistent detection quality.

1.1 Objectives and Contributions

This thesis focuses on the automated detection of clean code violations in Python source code. We have the following main objectives:

- Design and implement a platform that can be extended with different automated checkers.
- Compare machine learning models on detecting code patterns that violate the clean code rules.
- Evaluate the generalisation capabilities to unseen pattern variations.

To fulfil the objectives, we made the following contributions:

- Overview over code quality, clean code rules, quantitative metrics and existing tools.
- Proposal for a taxonomy of the implementation complexity of automated checkers for clean code violations.
- Design and implementation of the clean code analysis platform (CCAP) with focus on extensibility, useability and integration capabilities.
- Extension of CCAP with checkers for returning `None` and using direct comparisons in conditionals.
- Comparison of our CCAP with preexisting tools.
- Creation of a dataset and evaluation of a support-vector classifier, random forest classifier, gradient boosting classifier and an LSTM-based neural network on detecting problematic code patterns for that we may not find a deterministic checker.
- Introduction of an automated way to manipulate the code to contain unseen problem variations.
- Simulation of the impact of having a hand-collected dataset for the machine learning models if the samples do not contain all possible variations of a problem pattern. We give ideas on how to handle the situation in a time-optimising way.

Based on our main objectives, we will evaluate the following research questions:

RQ1: What is the utility of the CCAP besides existing tools?

RQ2: How do different machine learning models compare on the task of detecting non-clean code?

RQ3: Do machine learning-based models cover a larger variety of cases than rule-based checker?

1.2 Structure

This thesis is organised as follows: In Chapter 2 we introduce code quality and clean code principles. We then classify the clean code principles depending on the implementation complexity of an automated checker. We introduce several quantitative metrics to measure code quality and provide an overview of existing tools that can determine the code quality and clean code violations to an extent. In Chapter 3, we describe our approach for the CCAP and the machine learning-based clean code classification. We evaluate the CCAP and machine learning models and answer the research questions in Chapter 4. Finally, we conclude our work with a summary and outlook in Chapter 5.

2 Background and Related Work

In this chapter, we describe relevant background information for this work. We define code quality, describe the clean code principles and explain the relation between clean code and code quality. We then introduce different techniques for static code analysis and define a taxonomy based on different complexity levels of automated checkers for clean code violations. Afterwards, we introduce quantitative metrics to measure the code quality and their useability to measure clean code violations. We conclude with an overview of related work in the form of tools for code quality analysis and in academic research.

2.1 Code Quality

Before defining code quality, we define software quality. According to ISO/IEC 25010:2011 standard, software quality describes the degree "to which a product or system can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, freedom from risk and satisfaction in specific contexts of use" [4]. Software is written with source code, so the quality of the source code reflects the software quality on the implementation level.

The standard specifies eight characteristics for software quality [4]. The main influence of source code and its quality lies in those four characteristics:

1. Reliability
2. Performance efficiency
3. Security
4. Maintainability

Reliability is the ability of a system to run without critical exceptions that would cause the system to crash, become unavailable or result in an inconsistent state. A code with

high quality runs without such critical exceptions. Performance efficiency describes an optimal use of the resources like CPU-time, memory, or network bandwidth. Well written code with a high quality uses the resources efficiently. The security aspect focuses on minimising possible attack vectors. Software is vulnerable if the source code implements behaviour in an unsafe way. A code with high quality would implement its functionality with a focus on the security aspect.

Software maintainability consists mainly of those activities that involve changing source code:

1. Adding code for new features.
2. Implementing changed requirements by modifying code.
3. Fixing bugs to ensure correct functionality of the software.

The importance of good maintainability of source code is based on the associated costs. For project planning, different sources suggest, that project manager should estimate 66% or even 75% of the total software costs for maintenance [5, 6]. A recent study from 2018, conducted by Stripe and Harris Poll highlights that developers spent 42% of their work time maintaining code [7]. Increasing the effectiveness of maintaining code is consequently cost-relevant. The easiness of code changes due to code quality becomes a business-critical requirement and can positively impact the maintainability in the following ways [8]:

1. Well-written code makes it easy to determine the location and the way source code has to be changed.
2. A developer can implement changes more efficient in good code.
3. Easy to understand code can prevent unexpected side-effects and bugs when applying a change.
4. Changes can be validated easier.

The application of agile software development methodologies like *Scrum* is another important contributor for code to be changeable with the least possible effort. *Scrum* is an agile software development practice of building software in increments while having a running software at the end of each increment. One of the advantages is the adaptability of this model to changing requirements, since, with the end of each increment, a requirement change can be implemented [9].

To sum up, code quality is part of the overall software quality on the implementation level. It contributes to several characteristics of software quality, with the most important one being the maintainability. A good code quality allows efficient changes and therefore, high maintainability.

2.2 Clean Code

In Section 2.1, we defined the terminology of software and code quality. The *clean code concept* describes rules and is coined by Robert C. Martin in his book "Clean Code: A Handbook of Agile Software Craftsmanship" [3]. These rules should ensure readability and understandability for developers. With understandability and readability, the changeability and in turn the maintainability of code is impacted.

The author describes clean code principles in the form of practical rules. Those practical rules should result in clean code that is easy to understand and read. Chaotic code, by contrast, is harder to read and understand. The author argues, developers produce chaotic code in a conflict between deadline pressure and the necessary effort to make code more intuitive and follow clean code principles [3]. The former pressure is created by deadlines that focus on the visible output like the functionality of the program, whereas the latter is not directly visible in the final product. If the success criteria are solely based on the visible output, the code quality may suffer as a consequence of fast-to-write chaotic code. This behaviour is shortsighted since an accumulation of chaotic code reduces productivity over time. A larger system with chaotic code will slow down later modifications or additions of code. With the costs associated with maintainability as described in Section 2.1, reducing the maintenance effort is a logical step. With the clean code principles, developers can read and understand code in a more intuitive way, and the described costs of chaotic code will decrease [3].

The following sections explain the clean code guidelines, as described by Robert C. Martin [3]. First, we describe rules for naming and functions. Then we recap rules regarding comments, data structures and objects. Finally, we conclude with suggestions for classes and error handling.

2.2.1 Naming

The naming section covers several rules for naming variables, functions, types and classes. Good naming can make it easier to read and understand code. "Good" nam-

ing is an opinionated topic; For Robert C. Martin, the key factor of good naming is the *descriptiveness* of names [3]. Descriptive names provide the reader with enough information to understand what a variable stores or what a function computes. Abbreviations or symbolic names like *a1, a2* are not descriptive and do not provide information about the meaning. Using symbolic names for mathematical expressions could be an exception to the rule if the naming mirrors the expression.

To come up with a descriptive name, it is helpful to include a verb or verb expression for function names, because functions express an action. For class names, a noun emphasises the object character of the class.

Another valuable property of a name is an *easy pronunciation* since it is easier to read and to talk about the code with other developers. For reading, the pronunciation improves subvocalization[1]. Non-dictionary words or unusual abbreviations are harder to pronounce. Consequently, it is useful to make longer but descriptive and pronounceable names, especially since the autocomplete feature of IDEs will free the developer from typing the long name. Additionally, searching for long names works better than for short names, because long names are more likely to be unambiguous compared to shorter names. Acceptable exceptions to this rule are short variables in a small scope (e.g. variable `i` in a loop), but using `i` in a large scope could be ambiguous and troublesome for searching and understanding.

2.2.2 Functions

This chapter describes several rules regarding functions, that Robert C. Martin characterises [3].

First, functions should be *small in length.* Exceeding 20 lines should not be necessary in most cases. If a function is small, it can be read more easily and without scrolling.

Second, inside a function, *if-, else- and while-statements should contain a function call for the body.* Additionally, the rule of descriptive naming can apply to the condition of the if statement: If the condition becomes hard to understand, a function call can simplify the understanding of the condition. With a descriptive naming, a function call documents the condition and body in a concise and readable way. In Listing 2.1, reading a condition with function calls does not require the reader to decrypt the logical expression. This can also save additional comments that explain the meaning of the condition. Furthermore, since the body also contains a function call, it is easy to understand the

[1] the internal speech when reading

```
if person.is_authorized():
    modify_data()
else:
    redirect_to_login()
```

Listing 2.1: Sample for using functions in if-statements with the increased documentary value of the cause-effect relationship.

cause-effect relation of the if expression.

Third, functions should fulfil *one purpose*. Since functions may have to call several functions subsequently to perform the necessary computation, Robert C. Martin expands the rule that functions should only operate on one abstraction level [3]. This would result in the following structure: Functions with a low abstraction level handle data access and manipulation, e.g. string manipulation. Functions on a middle abstraction layer orchestrate the low-level operations. On the next higher level, the mid-level functions are orchestrated. Following this structure, a function has one purpose on the low level and one orchestration purpose on a higher abstraction layer.

Fourth, *the number of function arguments should be three or lower*. With many function arguments, it becomes harder to call a function. Simultaneously, testing becomes harder, especially if all argument combinations should be tested. Functions with none or one argument simplify testing. Since testing can reveal bugs, testable code is crucial for reducing the number of bugs. If a function requires more arguments, it may be advisable to bundle those in a config object to pass multiple arguments as one. With a config object, arguments can be logically grouped and thus help the reader to understand the arguments.

Fifth, *side-effects in a function body should be avoided*. Side-effects happen if a function modifies a variable outside its scope without explicitly mentioning this in the function name. This leads to dependencies between the functions that are not obvious to a developer who checks the signature with its name, input arguments and return type when using the function. To spot a side-effect, the developer would have to read through the function, although reading the signature should be enough to use the function. With side-effects, the function behaves unexpected, and time-consuming mistakes are the consequence. Especially side-effects that initialise other objects result in a time-dependency that is hard to identify and could be overlooked easily.

The next rule forbids the *returning of error codes* and suggests raising an exception if the programming language supports it. Raising an exception allows better separation between application logic and error handling code since the error code checking inter-

```python
def query_all_admins():
    try:
        all_users = database.read_all_users()
        admins = [user for user in all_users if user.is_admin]
        return admins
    except DatabaseError as error:
        print("Database error", error)

def query_all_admins():
    all_users = database.read_all_users()
    if all_users == -1: # -1 is database error
        print("Database error code -1")
    else:
        admins = [user for user in all_users if user.is_admin]
        return admins
```

Listing 2.2: Sample listing for error handling with try and except statements vs error codes.

rupts the reading flow of code. Additionally, catching a raised exception is separated from the logic for a non-error execution and can be separated into an additional function for an even cleaner structure. Listing 2.2 shows an implementation of a simple function with exception raising and error code return in the `database.read_all_users` method. The former error handling shows the separation between the application logic and the error handling logic, whereas handling the error code in the latter interrupts the reading flow of the code.

Last, the important principle for software engineering applies directly to functions: *Do not repeat yourself* [10]. Repeated code is dangerous and chaotic since it requires changes in multiple locations if it has to be modified. This makes duplicated code very prone to copy and paste errors and small mistakes. Those may not be obvious during development but can lead to a fatal crash at runtime. Code clone detection is an active research field, especially for finding semantic clones without syntactic similarity. For instance, Büch and Andrzejak suggest an AST-based recursive neural network [11] and Saini et al. present an approach that can even detect duplications with just minor syntactic similarities [12].

2.2.3 Comments

Comments are part of most programming languages and can play an important role in code quality. Subjectively good comments clarify the meaning of the code and help to understand the code. However, comments can become outdated and wrong or provide

useless information [3]. In a perfect world, the programming language and the programmer would be expressive enough that commenting is not required for clarity. Following the clean code guidelines for naming and functions makes many descriptive comments obsolete, since the function description is encoded in the function name.

Comments will not help to turn chaotic code into clean code. If a developer explains a line of code by a comment, it is often more helpful to call a function with a descriptive name to replace the comment. Since comments are not part of the program logic, missing a comment update along a code update would give misinformation to the reader as described before.

In brief, before writing a comment, the developer should think about expressing the same meaning in code. Robert C. Martin gives some exceptions for situations, where comments can be helpful or necessary [3]:

Legal notes: Legal notes like copyright or license information and author mentions may be necessary. Although they should be short and link to an external licensing document in full extent.

Explanatory comments: Some explaining comments can be helpful and are not easy to encode in regular code. For instance, an explanation of a complex regular expression may be too complicated for a function name, and a comment provides the space for a sufficient explanation. This increases the readability since the developer does not need to interpret the regular expression manually.

Intention: Explaining an intention which does not become evident by reading the source code might also be a valid use-case for a comment.

Warning for consequences: Some parts of the code can have extraordinary consequences like not being thread-safe or using many system resources. A warning can spare a developer from having trouble when using the function.

Emphasis: A comment to emphasise the importance of a seemingly unimportant part of the code prevents breaking modifications of the code.

2.2.4 Data Structures and Objects

This chapter describes the rules of data structures and objects. Both store data but the way this data gets exposed is different. Objects hide data behind abstractions and

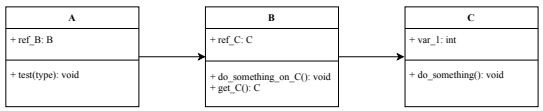
have functions that operate with their data. Data structures expose the data and do not provide functionality.

For objects, I. Holland defined the *Law of Demeter* (LoD) that is part of the clean code rules [13]. In object-oriented programming, a method m of an object O may only call methods of the following components:

- The object O itself.
- The parameters of the method.
- Objects that are created within m's scope.
- Instance variables of O.

In other words, the object O only calls methods on direct "neighbours" or on itself. The object does not call a method of its neighbour to subsequently call a method on the returned object from the neighbour. By restricting the allowed method calls outside the object to direct "neighbours", the dependencies between objects are reduced, and the modularity increased. As a consequence, changing methods only affects the direct neighbours and not objects in potentially different parts of the software. Figure 2.1 shows an example for the LoD. In the method `test`, a method call on `B` returns a reference to `C`. Since `C` is not a direct neighboor of `A`, the function call violates the LoD. The modularity of objects suffers, since a change of the method `do_something()` would require changes in object `A`. This close coupling of `A` and `C` is circumvented with a method `do_something_on_C` that wraps the functionality of `C`. A change in `C` would only require a change in `B`. With that common layer, `A` and `C` are decoupled. Object A in this case does not know how a certain functionality is performed and it does not matter. Bad code practice like the violation of the LoD in Figure 2.1 is called a train wreck since subsequent method calls look like multiple train carriages. The described wrapping function can improve such train wrecks.

Data structures are often used as data transfer objects (DTO) to communicate with other processes or services. These objects do not contain any functionality; instead, they only have accessible member variables via accessor methods or directly. Specialisations of DTOs are active records that contain additional methods for data storage. They are used to represent a data source like a database. For DTOs and active records, it is bad practice to insert business logic into these objects [3]. With business logic, the data structure becomes a hybrid between an object and a data structure. It becomes unclear if the purpose of a hybrid is to store data or to expose methods to modify the stored




```
def test():
    # violation of LoD
    self.B.get_C().do_something()

    # no violation of LoD
    self.B.do_something_on_C()
```

Figure 2.1: A sample illustration of the *Law of Demeter*. The figure on the left shows a class diagram for the classes A, B and C. The listing on the right shows the implementation of the *test* method of class A.

data in a controlled way. Without the clear purpose of the object or data structure, it becomes harder to understand.

2.2.5 Classes

For classes, the clean code principles describe multiple rules that improve the readability and understandability of classes.

The first rule specifies the recommended size of a class. Unlike counting lines as for functions, the size of a class is the number of responsibilities. A responsibility of a class can be manipulating data or writing data to a file. A single responsibility per class is described as *Single-Responsibility-Principle* and offers several advantages [3]: First, the naming can reflect this responsibility, so the class functionality is easy to understand. Second, changing one functionality should only require changes in the one class that is responsible for this specific function. And last, we can reuse the class in different projects if we need that functionality. If the class fulfilled two responsibilities, we would have to remove the unnecessary functionality before reusing the class in a different project. Reaching back to the examples of one responsibility being data manipulation and one being file writing, reusing the latter functionality would require us to strip out the data manipulation. The *Single-Responsibility-Principle* leads to a system of many small classes with one, clearly defined responsibility.

The second rule recommends classes to have *high cohesion*. The cohesion is high if a method manipulates many instance variables. If all methods of a class manipulate all instance variables, the class has its peak cohesion. With a high cohesion, the methods and the class are a logical unit. A low cohesion indicates the need to split the class into multiple classes since the methods are not a logical unit. As a result of splitting classes into smaller logical units, they will more likely comply to the *Single-Responsibility-Principle*.

The last rule provides a guideline on how to handle internal or third-party class dependencies. If a dependency changes, the class depending on it should not change. To accomplish this *decoupling from the implementation of a dependency*, classes should not depend on the dependency directly but instead on an abstract class (or interface). That abstract class describes the concept of the dependency that is unlikely to change. For example, for a database dependency, the abstract class defines the concepts of storing and retrieving an entry. For the specific database, a specific class implements the abstract class or interface and provides the functionality. Changing the database only changes the implementation in the specific class but not the concept. Therefore, there is no need to modify a class that depends on the concepts from the abstract class. This isolation of dependencies is especially useful if the dependencies are not under the developer's control and may be changed at any time by a third party. The abstraction is also valuable for testing since the dependency can be replaced by a so-called mock class that only simulates the behaviour. This allows the testing to be independent of the dependency and to locate potential errors in the own system or the dependency.

2.2.6 Exception Handling

This chapter covers clean code rules for error handling. The following ways are methods for handling errors: First, a function can return an error code if some exception occurred. The function caller has to check for the error code and act accordingly. Second, some programming languages support the concept of throwing and catching an exception. Third, a function can return a null value to indicate the execution failure.

In Section 2.2.2, we described the clean code rule that prefers throwing an exception over returning an error code. The supporting argument of Robert C. Martin is the better separation between application logic and error handling [3]. Error codes have to be checked immediately, so the error checking interrupts the code for the application logic. By throwing and catching errors, the error handling can be completely extracted into a separate function and can be removed from the main application logic. The `try` block contains all logic for the regular program execution. In case of an exception, the execution jumps to the `catch` block to handle the error[2]. Furthermore, the `try-catch` block enforces transactional behaviour. At the end of either the try or the catch block, the application should be in a consistent state. Error codes hide this explicit transactional behaviour.

[2]This blocks are named differently in e.g. Python (`try`, `except`, and `raise` an exception)

Despite the clean code rules, Go as more recent programming language[3] returns explicit error types instead of throwing an exception. This was designed to "encourage you to explicitly check for errors where they occur", instead of "throwing exceptions and sometimes catching them" [15]. So the practical application of the clean code rule for throwing exception depends on the design of the programming language.

The third method of error handling is returning a null value. Instead of checking for a specific error code, the caller would check for a null return value. If a developer misses the check, the program will terminate with an unhandled NullPointerException at runtime. In other programming languages, the null value can be named differently like `None` in Python and the exception may differ. Despite the different naming, the problem remains. Tony Hoare introduced the null reference in 1965 and later called it his "billion-dollar mistake" [16], since it leads to many bugs and security vulnerabilities throughout the decades. Languages like Kotlin are designed with null safety enforced. Kotlin distinguishes between nullable and non-nullable references, with a compiler enforcing explicit null checking [17]. If a language does not enforce null safety with a compiler, a special case like an empty collection or an optional type can be returned. Returning null introduce the same problems as error code handling such as mixing application and error handling logic. Furthermore, it is not explicitly marked as an error code. From a function signature, it may not be evident that the return value may be null if the language does not support compile-time checking or optional types. Robert C. Martin describes returning null as a violation of the clean code rules [3].

Independent of the error handling method is the next rule for error handling with a third-party library. In section Section 2.2.5, we described *wrapping dependencies* like third-party libraries to decrease the coupling to the dependency. The wrapping concept also applies to error handling, since the wrapper can unify the exceptions, so the application does not check for dependency-specific error types. Changing the dependency requires only a change in the wrapper implementation but not everywhere this wrapper is used.

2.2.7 Additional Rules

Robert C. Martin describes more rules for system design, multithreading, testing and third-party code [3]. Additionally, the author describes the concept of code smells. In theory, following these rules seems to lead to clean code. In practice, however, developers

[3] designed in 2007 as a response to problems with C++, Java and Python [14]

tend to violate rules in some situations. A code smell is a code characteristic that could indicate such a violation.

This work will focus on the described rules and corresponding automated checkers.

2.3 Source Code Analysis

In order to define and measure code quality, methods for analyzing a program and its source code are necessary.

To analyse source code, different techniques are bundled as static code analysis. A static code analysis gathers information about the structure of the source code and program [18].

The following provides a non-comprehensive overview of the most important methods for code analysis and representation. Source code itself is stored as encoded text files. This representation offers no structural information.

A first processing step is a lexical analysis, also known as tokenization. The tokenization transforms the character stream from the text file into a stream of tokens. This is possible due to the syntactic definition of a programming language, that allows splitting the raw text into typed tokens [19]. A token type can be, for instance, a keyword or an operator.

With the token stream, a parser can build an Abstract Syntax Tree (AST) that represents the abstract structure of the source code. Since the AST represents the code structure, it is possible to traverse the tree-like data structure to analyse the code structure.

Another method is called control flow analysis. To perform a control flow analysis, the program has to be represented as a control flow graph. The control flow graph represents possible execution paths in a program and consists of nodes and edges. A node represents a basic block, i.e. a code sequence without branching. A directed edge represent jumps in the control flow; e.g. an if-statement would have two outgoing edges, one for a true condition and one for a condition evaluated as false [20].

Besides the control flow, it is possible to analyse the data flow in a data flow graph. With the data-flow analysis, it is possible to collect information about the read and write operations to variables. In combination with the control flow graph, the data flow for different control flows can be calculated [19].

Last, a call graph can be computed. A call graph maps all method or function calls starting from the primary method. It allows spotting dependencies between mod-

ules [18].

2.4 Clean Code Complexity Levels

We described the clean code rules in Section 2.2. This work focuses on the automated checking of clean code rules. Some rules can be checked straightforward with an algorithm, whereas others may be too abstract to be checked in an automated way. In this section, we want to clssify the clean code rules into different complexity levels. The complexity levels are based on the complexity of implementing an automated rule checker.

We define the taxonomy of complexity in three levels, based on the different source code analysis techniques and effort needed:

Basic Level: The level of basic complexity covers all rules that can be checked by a single analysis of the text, token stream, or AST of the program.

Advanced Level: Rules with medium complexity require a control flow, data flow, call graph analysis or more advanced, deterministic analysis techniques. Furthermore, a combination of different analysis methods from the basic complexity level will be categorised as advanced complexity.

High Level: The high level of complexity covers the clean code rules we assume can not be checked with traditional, deterministic analysis techniques such as those described in Section 2.3. For this level, it may be indispensable to use statistical methods like machine learning approaches to create an automated checker. It is important to note that we do not prove that no deterministic algorithm exists.

Based on these levels, we classify the clean code rules introduced in Section 2.2. Table 2.1 gives an overview of all rules and our assigned complexity level. In the following, we explain the reasoning behind the classification for all rules:

For the naming category, all rules have in common that an algorithm would have to extract the names of various entities like variables, classes, or functions from the source code first. The AST provides the names and entity types they describe. Extracting all names is a matter of traversing the AST. We see this in the basic complexity level.

The first rule in naming is the descriptiveness of names. Assessing the descriptiveness of a name is a problem with a high level of complexity. An algorithm would have to understand the purpose of the source code to determine if the name describes the source

		Level		
Category	Rule	Basic	Advanced	High
Naming	Descriptiveness			X
Naming	Pronunciation	X	X	
Naming	Names in Small Scope		X	
Functions	Length	X		
Functions	Conditional Function Calls	X	X	
Functions	Purpose			X
Functions	Levels of Abstraction			X
Functions	Number of Arguments	X		
Functions	Side-Effects			X
Functions	Error Code Detection			X
Functions	Duplicated Code			X
Comments	Detecting Necessary Comments			X
Objects	Law Of Demeter		X	
Objects	Mix with Data Structures		X	
Classes	Single Responsibility		X	X
Classes	Decouple Dependencies		X	
Errors	Return None	X	X	

Table 2.1: Classification of clean code rules based on our taxonomy for the complexity of automated checkers into different levels.

code. It is an ongoing research effort to use machine learning on code for different tasks [21]. The tasks of code summarisation explore the use of machine learning to generate a natural language description of source code. In a paper from 2018, Alon et al. describe a neural network to represent code as vectors and "predict a method's name from the vector representation of its body" [22]. Based on that and the following research, it may be possible to compare the summarisation result with the chosen name to detect non-descriptive names. Nevertheless, it may be harder to apply such concepts to variable names, since a variable name has less contextual information compared to the method name with its method body.

The next naming rule covers the easy pronunciation of names. A rule violation would be the use of abbreviations or non-dictionary words in variable names that are hard to pronounce. An approach with a basic level of complexity could be a minimum number of characters for every name. A more advanced approach could utilise naming conventions such as camel case or underscore-separated words to extract the single words of a name and compare it with a dictionary. While it would help to prevent spelling mistakes and

should ensure pronounceability, a dictionary may not contain all words. Especially with domain-specific words, this could lead to false alarms.

Variable names in a small, local scope are excepted from the rule of pronounceability. To determine the scope of a variable, the data flow analysis could be used. With the analysis, the previous rule could include this exception. Due to the data flow analysis, we categorise this rule into the advanced level of complexity.

The first rule in the function category restricts the length of functions to be less than 20 lines of code. A checker for this rule could be grouped at the basic level of complexity since the AST contains all the necessary information about the function body. If the metadata (e.g.line number) is retained during AST construction, a checker can calculate the lines of code for the function body.

The next rule covers function calls in conditions and in body parts of a conditional such as an if-statement. An if-statement is represented as a node with a condition, body and an else descendant. An algorithm can scan the AST to check the condition and body nodes of the conditional statement. Such a checker would have a basic level of complexity. Since a function call mainly increases the understandability of more complex conditions, some threshold such as a maximum number of logical sub-expressions could be applied. Due to the additonal consideration of the threshold, we would group an automated checker in the advanced level of complexity.

Regarding the purpose of a function, the clean code rules suggest a function to have one purpose and to operate on one abstraction level. Counting the number of purposes requires an understanding of the semantics of the code. Similar understanding would be necessary for the level of abstraction. We argue such a checker would be in the high level of complexity since the understanding of code could only be achieved with machine learning. The aforementioned research field of code summarisation could be beneficial for this task too. A natural language description of a function may require the model to "understand" the purpose and task in order to articulate it in natural language. It may be possible to use work in this field to detect functions with multiple purposes or abstraction levels.

The rule that limits the number of function arguments is on a basic level of complexity. The AST contains the information about the code structure and therefore, all function arguments. Counting those requires the algorithm to search for function nodes in the AST and count the number of arguments.

The next rule forbids side-effects of a function. Detecting side-effects can be broken down into a two-step process. First, tracing the data manipulation of functions is at the

advanced level of complexity. With a data flow analysis and call graph, it is possible to check what variables a function manipulates. Second, the classification whether an effect of a function is a side-effect requires an understanding of the intended effect of a function. The intended effect is described in the name. We argue that the understanding of the intended effect is at the high level of complexity.

The clean code rules discourage the use of error codes to indicate exceptions. If the error codes are not part of the programming language specification (as they are not for Python), it could be challenging for an automated checker to distinguish between returned error codes and non-error values. There may be indicators such as an early return of a constant value or comparing a returned value with a constant value. This indicators as a heuristic may be good enough for practical use. We would classify such a checker at a high level of complexity.

The automated checking for repeated code is an active research topic, as described in Section 2.2.2 [11, 12]. Especially the detection of duplicated semantic is hard with just a few syntactic similarities. Therefore, we classify a code duplication checker as a high level of complexity.

The clean code rules suggest to not use comments except for legal notes, explaining comments, comments stating intention, warnings for consequences and emphasising purpose. An automated checker would need to flag all comments, except the allowed ones as stated by the rules. Finding all code comments is possible on the token stream. While this has a basic level of complexity, detecting the exceptions is harder. The first comment at the start of a file could be seen as a legal note, and a warning for consequences may include the term “warning”. However, distinguishing necessary explanations or intentions from useless comments is a task with a high level of complexity.

For objects, the clean code principles require compliance with the *Law of Demeter* that restrict methods calls. An automated checker may find violations by analyzing the ASTs of all classes. It can then assess if a method call is violating the LoD. Due to the analysis of multiple AST to determine allowed “neighbours” of an object, we see this checker on the advanced level of complexity.

The next rule discourages the mixing of data structures with objects by introducing application-logic. An algorithm could analyse the AST and detect data structures by searching for public variables and accessor methods. For objects, the algorithm would expect no public variables and no accessor methods. To determine, if a method is a an accessor method, a data flow and control-flow analysis could be sufficient. If a method only reads or writes to a single member variable, it would be an accessor method, whereas

methods with more complicated control and data flow would not be seen as accessor method. This determination process would be an advanced level of complexity.

For classes, the clean code rules enforce a single responsibility per class. As discussed for detecting a single purpose of a function, detecting a single responsibility of a class requires an understanding of the code. Similarly, a checker would fall into the high level of complexity. On the contrary, it may be possible to use the cohesion metric as an indicator for too many responsibilities of a class. A high cohesion will result in smaller classes that form a logical unit. Smaller logical units can be expected to have fewer responsibilities. Cohesion can be measured as *Lack of Cohesion in Methods* (LCOM) as described by Hitz and Montazeri [23]. An analysis of the AST and call graph should provide all information to calculate the LCOM number. Therefore, we see an automated checker for cohesion on the advanced level. Using the cohesion as a heuristic for the responsibilities of a class, we could detect potential violations against the *Single-Responsibility-Principle* with an advanced level of complexity.

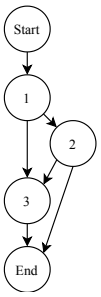
An automated check for the decoupling of dependencies can be performed with an advanced level of complexity. All required information is in the code. Finding all class dependencies is possible by combining several ASTs. Following the dependencies, it is possible to determine whether they inherit from an abstract class or implement an interface. In that case, the checker would assume a dependency decoupling. Based on our classification schema, this checker would be in the advanced level of complexity.

Last, the clean code rule prevents the return of null values as error codes or as an indication for no result. A checker could prevent this with a basic level of complexity. It could search for all `return` statements in the AST and check for the `null` constant. If a variable gets returned, the checker would have to check, if the variable could be a null value. Therefore, the data flow and control flow analysis are necessary. The latter approach would have an advanced level of complexity, whereas the former has a basic level of complexity.

2.5 Quantitative Metrics for Code Quality

Quantitative metrics represent the code quality as quantitative unit. We see the following advantages when using quantitative metrics in software projects:

- A metric sums up the quality of a project in a single unit.
- A quantitative approach tracks the changes in code quality over time. Therefore,




```
def function(a,b):
    if a < b: # (1)
        if b < 10: # (2)
            a = a + b
        else:
            return b # (end)
    a = a * 2 # (3)
    return a # (end)
```

Figure 2.2: Control-flow graph visualisation for a code sample. The graph has six edges, five nodes and one connected component. The *Cyclomatic Complexity* is 3.

it is evident if code quality improves or not. In case the quality undercuts a threshold, special measures like mandatory improvements can be undertaken.

- A developers performance can be evaluated based on the code quality. Since the maintainability and reliability of the software depends on the code quality, this is a good incentive to enforce high-quality work.

- A certain level of code quality may be required by a contract. As a result, a customer may have fewer bugs and a smaller maintenance effort.

The following sections describe common software metrics that express code quality.

2.5.1 Cyclomatic Complexity

Cyclomatic Complexity is a metric for the complexity of a code section, which was introduced by Thomas McCabe [24]. It measures the complexity by counting the number of linearly independent execution paths [24].

An execution path is the subsequent execution of instructions. With control flow structures such as if statements, two execution paths are possible, depending on the evaluation of the if condition. An execution path is linearly independent if it includes one subpath that is not part of any other path. A control flow graph represents all possible control flows as described in Section 2.3. Figure 2.2 shows a control flow graph for a sample function.

The *Cyclomatic Complexity* on a control-flow graph as the number of linearly independent execution paths is defined as [24]:

$$M = E - N + 2P \tag{2.1}$$

E is the number of edges, N the number of nodes and P the number of connected components. A connected graph is a subgraph, in which all nodes are connected to each other by a path. Following this definition, the *Cyclomatic Complexity* can be calculated. Figure 2.2 shows a visualisation of a sample python function. The control-flow graph has six edges, five nodes and is one connected component. Using Equation (2.1), the *Cyclomatic Complexity* is 3.

MacCabe recommends limiting the *Cyclomatic Complexity* to 10 [24]. A lower *Cyclomatic Complexity* improves testability since the complexity represents the number of execution paths that need to be tested. Therefore, M is the upper bound for the number of test cases for full branch coverage. Software that has to comply with safety standards like ISO 26262 (for electronics in automobiles) or IEC 62304 (for medical devices) are mandated to have a low *Cyclomatic Complexity* [25, 26].

Although *Cyclomatic Complexity* is used throughout the industry, several shortcomings are under critique. First, complexity from data flow is ignored. When working with a larger number of variables and operations, the code becomes more complicated, but the *Cyclomatic Complexity* does not take this into account. Second, nested code structures are not considered by the metric, although it adds additional difficulty for understanding [27].

2.5.2 Halstead Complexity Measures

Maurice Halstead introduced the *Halstead Complexity Measures* (HCM) in 1977 [28]. Halstead approached the complexity measure with an empirical approach by defining observable and measurable properties and put them into relations.

The measurable properties are operators and operands. Operators are symbols in expressions like an addition, subtraction or multiplication symbol. Operands are the values and variables that are manipulated by the operators.

The sum and number of distinct operators and operands are counted as:

- η_1 as the number of distinct operators.
- η_2 as the number of distinct operands.
- N_1 is the sum of all operators.
- N_2 is the sum of all operands.

From these base properties, Halstead derived additional properties [28]:

- Program vocabulary size:

$$\eta = \eta_1 + \eta_2$$

- Program length as the sum of all operators and operands:

$$N = N_1 + N_2$$

- The volume of a program in terms of program length and program vocabulary is defined as:

$$V = N * \log_2 \eta$$

Based on those properties, code metrics can be calculated. First, the difficulty of understanding a program is calculated as:

$$D = \frac{\eta_1}{2} * \frac{N_2}{\eta 2}$$

The major contributors to difficulty are the number of distinct operators η_1 and the sum of all operands N_2.

Second, the combination of difficulty and volume is the required effort for understanding or changing code and is defined as:

$$E = D * V$$

Third, the effort for changes translates into real time following the relation:

$$T = \frac{E}{18} s.$$

Last, the number of bugs correlates with the following relation [27]:

$$B = \frac{V}{3000}$$

HCM focus on the complexity introduced by data manipulation but ignores complexity from the control-flow. Since the *Cyclomatic Complexity* focuses on the control-flow complexity but not on the data-flow complexity, it is practical to use both together to supplement each drawback [27]. Additionally, the correlation with the number of bugs is based on the programmer's skill estimation with a fixed value of 3000. Since this varies between projects, such an "experiential and fixed value [...] is doubtable" [27].

2.5.3 Software Maintainability Index

The *software maintainability index* was developed by Dan Coleman and Paul Oman in 1994. 16 HP engineers evaluated 16 software systems and scored them in a range from 0 to 100, with 100 representing best maintainability [29]. Following a regression analysis, they identified the following equation to match the maintainability of the evaluated systems:

$$MI = 171 - 5.2 * \ln \overline{V} - 0.23 * \overline{M} - 16.2 * \ln \overline{LOC} + 50 * \sin \sqrt{2.4 * C}$$

$\overline{V}$ is the average Halstead Volume, $\overline{M}$ the average cyclic complexity, LOC the lines of code and C is the fraction of comments.

The *software maintainability index* was defined many years ago with a limited sample size of developers and projects. Additionally, programming languages have changed significantly over time. A study by Sjøberg et al. suggests no correlation between the *software maintainability index* and actual maintenance effort in a controlled environment [30]. Although the study only analyes four software projects and lacks generalization, they found a strong anti-correlation with different maintainability metrics. Only the code size seems to correlate with the actual maintainability. The former seems to be consistent with a systematic review on software maintainability predictions and metrics by Riaz et al. [31].

2.5.4 Application on Clean Code

The described metrics are used to quantify the code quality. Since the clean code principles have a positive impact on code quality, lower code quality metrics may indicate more rule violations. The *Halstead Metric* derives the understandability of different parts of the code but does not explain the reason for it. Similarly, a high *Cyclomatic Complexity* of a function can indicate a violation against the recommended single purpose per function. In other words, the metrics indicate parts of code with low quality without pinpointing a specific violation of clean code rules.

2.6 Tools for Code Quality Analysis

With quantitative metrics for software quality, different tools can analyse source code and can inform about the increase or decrease in code quality. It is good practice to

use static code analysis tools to improve code quality. The result of static analysis can be based on approximations, so there might be false positive results. An expert has to examine the result if necessary [18].

Besides the use of static code analysis for compiler and optimizations, it is also helpful for analysing the code quality. The tools in this chapter analyse different categories of code quality-related principles:

Code Guidelines: A static code analysis can ensure compliance to structural, naming and formatting code guidelines.

Standard Compliance: A requirement for a software may be compliance to an industry-standard like IEC 61508 or ISO 26262 [32, 25]. Static code analysis can ensure or at least assist with the compliance.

Code Smells: Code smells are characteristics of source code that can indicate a underlying problem [33]. Some code smells follow a known pattern and a static code analysis can detect those smells. Since code smells can be an indication for chaotic code, it is best if these smells are detected and removed early on.

Bug Detection: Although bug detection with static analysis can not detect all bugs, every detected bug before a software release is essential. Examples for detected bugs are unhandled exceptions or concurrency issues [34].

Security Vulnerability Detection: Static code analysis tools can detect some security vulnerabilities like SQL Injection Flaws, buffer overflows and missing input validation with high confidence. Since these are some common, easy to exploit vulnerabilities, an analysis for security vulnerabilities can increase the security of the overall software [35].

The following sections provide an overview of several static code analysis tools with a focus on code quality and maintainability. We selected them based on popularity and if the projects are still maintained.

2.6.1 PyLint

PyLint is a code analysis tool for the Python programming language. It is open-source and licensed under the GNU General Public License v2.0 and available for all common platforms. PyLint can be executed as a standalone program or can be integrated into

common IDEs like Eclipse. A continuous integration pipeline may include PyLint as well to ensure an analysis on every build [36].

With PyLint, the developer can make sure that the code complies to the PEP 8 style guide for python coding [37]. This includes name formatting, line length and more. It does not calculate a metric for the code but instead warns about violated principles. Additionally, PyLint can detect common errors like missing import statements that may cause the program to crash at the start or later at runtime. To support refactoring, PyLint can detect duplicated code and will suggest code changes.

PyLint can be configured to ignore some checks and to disable specific rules. To expand the ruleset, a developer can write a "checker", an algorithm to check for a specific rule. The algorithm can analyse the raw file content, a token stream of the source code or an AST representation. The checker can raise a rule violation by providing the location information and the problem type to the PyLint framework, and it can be included in the PyLint analysis report [38].

2.6.2 PMD Source Code Analyzer Project

PMD is an "extensible cross-language static code analyzer" [39] for Java, JavaScript and more. As an open-source project, it is licensed under a BSD-style license and is available for macOS, Linux and Windows-based systems. It integrates into build systems like Maven and Gradle as well as into common IDEs like Eclipse and IntelliJ. PMD can run as part of a continuous integration pipeline and is included in the automated code review tool Codacy [40].

Depending on the target language, PMD supports different rules. For Java, PMD has rules in the following categories [41].

Best practices: Best practices include rules like one declaration per line or using a logger instead of a `System.out.print()` in production systems.

Coding Style and Design Several rules to improve the readability of the code like naming conventions, ordering of declarations and design problems like a violation of the *Law of Demeter*, *Cyclomatic Complexity* calculation and detection of large classes.

Multithreading: Rules to mainly prevent the use of not thread-safe objects or methods. Due to the nature of multithreading and unpredictable scheduling of threads, problems like unpredictable values of variables or deadlocks may occur. Since they may

not occur in every execution, they are hard to spot and to debug. Consequently, warnings of using non-thread-safe code may save hours of debugging.

Performance: These rules flag known operations that have hidden performance implications. For example, string concatenation with the plus operator in Java causes the Java Virtual Machine to create an internal string buffer. This can slow down the program execution if numerous string concatenations are performed.

Security: Security-wise, PMD only checks for hardcoded values for cryptographic operations like keys and initialization vectors.

Error Prone Code: PMD checks for several known code structures that will or may cause a bug at execution. Some rules are part of the clean code principles described in Section 2.2, and some rules are specific to the target language.

A user can expand the PMD ruleset in two ways: A XPath expression can be specified and will be validated against the AST by the PMD framework. For more control, it is possible to write a Java-based plugin that implements a custom AST traverser. The latter allows for more sophisticated rules and checks [41].

2.6.3 Codacy

Codacy is a software to "check your code quality" [42] for more than 30 programming languages. It is available as a cloud-based subscription service with a free tier for open-source projects and a self-host option for enterprise customers. Codacy runs as cloud software, and a user can connect a GitHub, GitLab or Bitbucket repository that will be scanned automatically on every push or trigger a scan of local files with a command-line program. Additionally, a badge can be added to the readme page to show off the analysis results.

Codacy can be seen as a platform that runs multiple different *analysis engines*. These analysis engines combine multiple tools depending on the language. For Python, Codacy uses PyLint, Bandit (a scanner for security issues) and a metric plugin. Due to the licensing of those analysis engines, the engines are open-sourced, whereas the Codacy platform is proprietary.

Depending on the language, Codacy supports scanning for common security and maintainability issues. The later is faced with scanning for code standardization, test coverage and code smells to reduce technical debt.

Codacy allows customisation by disabling specific rules and changing rule parameters like the pattern to fit the rules to the project [42].

2.6.4 Sonarqube

Sonarqube is an analysis tool to maintain high code quality and security. It supports 15 programming languages in the open-source version licensed under the GNU Lesser General Public License, Version 3.0. In the Developer Edition, Sonarqube supports 22 languages and 27 languages in the enterprise edition. Sonarqube is a self-hostable server application with a SonarScanner client module that can be integrated into build pipelines like Gradle and Maven as well as a command-line tool for other build pipelines. Additionally, an IDE plugin allows code scanning and reviewing inside the editor. The SonarScanner reports the result to the server, that serves a website to review the results. With the Developer Edition, branch and pull request analysis are possible, and a pull request will be annotated with the analysis results [43].

For Python, Sonarqube offers more than 170 rules. These rules are in the following categories [44]:

Code Smells: Code Smells like duplicated string literals are flagged with four different levels of severity (from higher to lower severity): Blocker, Critical, Major, Minor. Explanations and examples are accessible for the developer to understand and fix the issue. The analysis is more in-depth than AST-based solutions since it additionally uses control and data flow analysis.

Bug: Multiple rules cover bugs that would definitively result in a runtime exception and program termination. As an example, calling a function with the wrong number of arguments will be flagged with the highest severity, since it will raise an error during runtime. Although programmers may notice issues like this during coding and testing, the issue can remain hidden if it is only triggered by a particular execution path of the system.

Security Hotspot: The security hotspot analysis unfolds pieces of code that may be a real vulnerability and requires a human review. The scanning has a hotspot detection for seven out of the OWASP Top Ten Web Application security risks [44]. A flagged hotspot contains a detailed explanation of the reason for being flagged and a guide on how to review this hotspot. The recommendation to double-check

a piece of code can help to prevent a security vulnerability from being deployed to production.

Security Vulnerabilities: A security vulnerability analysis reveals a code that is at risk of being exploited and has to be fixed immediately. For example, misconfigurations of cryptographic libraries can be revealed quickly, and possible damage can be prevented.

Additionally, Sonarqube offers several metrics like test coverage and a custom derivate of *Cyclomatic Complexity* named *Cognitive Complexity* [45]. The authors see several shortcomings in the original *Cyclomatic Complexity* model:

- Parts of code with the same *Cyclomatic Complexity* do not necessarily represent an equal difficulty for maintenance.
- *Cyclomatic Complexity* does not include modern language features like exception handling and lambdas. Therefore, the score can not be accurate anymore.
- *Cyclomatic Complexity* lacks useability on a class or module level. Small classes with complex methods would have the same *Cyclomatic Complexity* as large classes with low complexity per method (such as getter or setter methods). The informative value of the metric is low.

Cognitive Complexity addresses these points, especially the incorporation of modern language features and meaningfulness on class and module level.

The calculation is based on three rules [45]:

Ignore Shorthand Structures: Method calls condense multiple statements into one easy-to-understand statement. Therefore, method calls as shorthand are ignored for the score. Similarly, shorthand structures like the null-coalescing operator reduce the *Cognitive Complexity* compared to an extensive null check and are therefore ignored for score calculation.

Break in the linear control flow: A break in the expected linear control flow by loop structures and conditionals adds to *Cognitive Complexity* as well as to *Cyclomatic Complexity*. Additionally, catches, switches, sequences of logical operators, recursion, and jumps also adds to *Cognitive Complexity*, since these structures break the linear control flow.

Nested flow-breaking structures: Nested flow-breaking structures additionally increase the *Cognitive Complexity*, since they harder to understand than non-nested flow-breaking structures.

The method-level *Cognitive Complexity* score represents a relative complexity difference between methods that would have the same *Cyclomatic Complexity*. Furthermore, the aggregated *Cognitive Complexity* on a class or module level differentiates between classes with many, simple methods and classes with a few, but complex methods. As a result, data transfer objects (containing mainly getters/setters) have a lower score compared to classes with complex code behaviour [45].

Developers can extend Sonarqube similarly to PMD. They can write a Java plugin that can access the AST but also a semantic model of the code. Additional, extensions can be created that provide functionality to other extensions. The Java plugin is compiled into a `.jar` file and placed into the plugin directory of the Sonarqube installation. For more straightforward rules, XPath expressions are possible through the web interface and allow a quick extension. For instance, if developers observe bad code style during a pull request review, they can quickly write a rule to enforce this rule in all subsequent pull requests automatically.

2.7 Automated Detection of Clean Code Violations

We described the background and related work to different aspects of code quality, such as quality metrics (in Section 2.5.1) and tools that detect problems in code (in Section 2.6). Those tools can help to improve the code quality and detect code smells. Code smells can be an indicator of clean code violations. Several studies have been performed to improve the code smell detection: Fernandes et al. compared 84 tools to detect code smell [46] and Fontana et al. applied 16 machine learning models on code smells by using quantitative metrics as model input [47]. A recent literature review by Azeem et al. compares studies for machine learning for code smell detection and that most models use quantitative metrics as features [48]. Their results indicate an "absence of studies investigating metrics different from the structural ones" [48].

We do not want to detect code smells but directly identify clean code violations on a source code level. Code smells can indicate clean code violation, but they do not necessarily pinpoint the root cause of the problem. A code smell that indicates a long function violates the clean code rule of small functions but does point to problems such

as mixing different abstraction levels. Additionally, we want to utilise the source code as a feature set directly without extracting higher-level metrics.

3 Approach

In this chapter, we first describe our approach in designing and implementing the Clean Code Analysis Platform. Afterwards, we depict our approach to use machine learning models to detect violations of clean code rules.

3.1 Clean Code Analysis Platform

The goal of the design and implementation of the Clean Code Analysis Platform (CCAP) is a tool for software developers to improve the code quality of existing and new code. The tool accepts a directory containing source code files as input and analyses the input for snippets of improvable code quality. If the analysis classifies a code snippet as problematic, it should help the developer to improve the snippet with information about the problem. Ultimately, this should train the developer to spot problematic code by his own and to write clean code by default. Consequently, the number of alerts should decrease over time. At the same time, the overall software quality of a project increases immediately at rewriting a marked snippet and in the long term at training the developer to write code with higher quality.

In order to use the tool effectively, the design and implementation should cover the following requirements:

Useability: The CCAP should be an easy-to-use tool. Developers shall be able to install and run the tool without difficulties. The extra effort of using this tool should be small, and the developers should use the tool in their day-to-day workflow without additional friction. The developers can interpret the results and localise the issue immediately.

Extensibility: The extension of detectable clean code violations should be easy. A clear defined interface for extensions is required. An extension developer would not need specific knowledge about the internal architecture of the tool. The extensibility allows the desired workflow of a developer finding problematic code in an, e.g.

peer-review, formalising it into an extension and sharing this extension with the team. With each iteration, the code quality of all team members would increase.

Integration: The tool should be easy to integrate into different systems. These systems include local workflows like git pre-commits or build systems and remote continuous integration/delivery/deployment pipelines.

A more specific requirement is Python as an input and extension language. After JavaScript, Python is the second most popular programming language 2019 according to the Github statistics [49] and the third most popular language according to the TIOBE index [50]. Besides the general popularity, Python is heavily used in the scientific community for machine learning and at universities for teaching programming. These groups are part of the potential target audience.

3.1.1 Architecture

The CCAP architecture is divided into the main part and two extension possibilities: An extension with analysis plugins adds automated checkers for more rules that are validated by the system. Adding an output plugin allows specifying the output format to fit custom workflow needs. The main part consists of four components: A core component to act as an orchestration unit, a component for handling the source code input, and a component for handling the analysis plugins as well as output plugins. The design follows the requirements and goals for the platform. Figure 3.1 provides a high-level overview of the components.

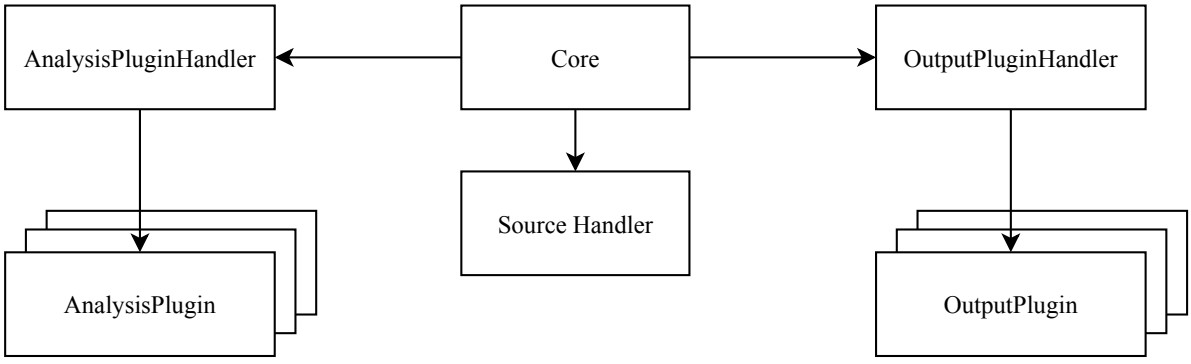


Figure 3.1: High-level schematic overview of the CCAP. The core component is the orchestration unit, the source handler parses the source code files, and the two plugin handlers search for and execute the analysis and output plugins.

Core Component The core component contains the main function and handles argument validation and parsing. Furthermore, it orchestrates other components by initialising and executing those. This process is divided into the argument parsing, initialisation and execution phase:

In the *argument parsing phase*, the command line arguments are parsed and validated. It validates the existence of the required input directory argument and the optional plugin path configuration for the analysis and output plugins. Additionally, the logging level and the output format can be defined. The latter determines, which output plugin will be used, although the existence of the specified plugin is not validated in this phase. A parsing or validation error will cause a program termination without further processing.

The *initialisation phase* instantiates all components, and the analysis plugin handler scans the specified directories for plugins and keeps an index of all found plugins in memory. The output plugin manager scans for an output plugin, that satisfies the specified argument. If no plugin matches the output argument, the program will terminate and display a failure message to indicate the problem.

In the *execution phase*, the input handling component scans the input directory for files ending with `.py` and parses the source code into an AST per file. In the next step, the core passes the parsed data to the analysis plugin handler. The latter executes all plugins for all files and collects the results. Afterwards, the core component calls the output plugin component to output the results. If no exceptions occur during the execution phase, the program will be terminated successfully.

Input Component The input component scans the given input directory for all Python source code files and parses the source code into an AST. For scanning the input directory, an algorithm walks recursively over all folders and files. The detection of Python source code files is based on the file ending `.py`. The algorithm returns a list of file paths and the corresponding file content.

Next, the AST parser is called and will add a parsed AST object to the list besides the file path and content. This list is passed to the analysis plugins by the analysis plugin handler. An alternative approach would be to not read and parse the code in the input component, but instead, let the plugins read and parse the file content if needed. With many files to scan, the latter approach would have a lower memory footprint since the file content and the AST will not be held in memory. However, every analysis plugin has to perform an expensive read operation from disk and the performance scales with

the number of files and the number of analysis plugins.

If the input component reads all files, the information is held in the main memory, and the performance only scales with the number of files. Since the files are text-based, we expect the number of files needed to exceed the main memory to be high enough to fit most projects.

Analysis Plugin Handler The analysis plugin handler manages all analysis plugins. This component finds all plugins, executes the plugins and collects the reported results.

During the *initialisation phase* of the core component, the analysis plugin handler scans the plugin directory for all Python files. It imports all Python files and scans those for classes, that inherit from an abstract `AbstractAnalysisPlugin` class. The abstract class defines all methods that need to be implemented in the specific plugin subclass. The analysis plugin handler instantiates all found classes.

During the core *execution phase*, the handler receives a list with the file name, file content and the parsed AST. All plugins are called on a specific entry point method that is defined in `AbstractAnalysisPlugin` with the aforementioned list. The plugin returns an instance of `AnalysisReport` with the plugins metadata and a collection of problems. The report is collected for every plugin into a `FullReport`. Additionally, the report contains information about the run arguments. After all plugins have been executed for all files, the analysis plugin handler returns the `FullReport` to the core component.

Output Plugin Handler The last component in the execution chain is the output plugin handler that passes the `FullReport` to the specified output plugin. It implements the same algorithm as the analysis plugin handler to find all plugins that inherit from `AbstractOutputPlugin`. Instead of keeping track of all plugins, only the plugin that corresponds to the output format argument is instantiated. The output plugin has an entry point as defined in `AbstractOutputPlugin` that is called with all the collected results.

3.1.2 Analysis Plugins

Analysis plugins provide the easy extensibility of the platform to developers. All users of the tool can extend the set of problems it can detect by implementing a plugin in Python and placing it into the plugin directory. In order to be compatible with the core components, a plugin has to inherit from the `AbstractAnalysisPlugin` class. First, the

```
class ReturnNoneProblem(AbstractAnalysisProblem):
    def __init__(self, file_path, line_number):
        self.name = "Returned None"
        self.description = "Returning None is dangerous since the caller has to check for None
. Otherwise, a runtime exception may occur."
        super().__init__(file_path, line_number)
```

Listing 3.1: Example for overwriting the `AbstractAnalysisProblem` with a specific implementation. The problem name and description are overwritten.

abstract class introduces a class member variable `metadata` of the class `PluginMetaData`. A specific plugin class sets this class member in the constructor to provide a plugin name, author and optional website. This metadata is used in the output to show more information about the plugin that reported a problem. Second, `AbstractAnalysisPlugin` specifies a `do_analysis` method as an entry point that the platform will call. It accepts a list of `ParsedSourceFile` objects that contains the file path, the file content and the corresponding AST for all input files. With this information, the plugin can implement any logic to detect problems. For example, the plugin can traverse the AST to look for specific node types, it can use a tokenizer on the content of the files and analyse the token stream, or it can run sophisticated machine learning algorithms on the source code.

The plugin can even import third-party libraries, although the user has to install those on the system. After detecting all problems, the plugin returns an `AnalysisReport`. The report contains the plugins metadata and a list of found problems. These problems are instances of a problem class inheriting from `AbstractAnalysisProblem`. The abstract class expects a file path and line number as constructor arguments and requires the plugin developer to override the problem name and description. Listing 3.1 shows an example implementation. The problem name and description will be shown in the final output and should adhere to the following guidelines:

- Have a proper name that allows the experienced developer to recognise the problem quickly.
- Explain what code construct is problematic.
- Give reasons why this code is seen as problematic.
- Show guidance and examples on how to fix the problem and improve the code.

Although it is possible to use one plugin for multiple, different problem types, having one plugin for one problem type helps to reuse and share the plugin. Additionally, it can be disabled easily by removing the plugin from the plugin folder and complies to the *Single-Responsibility-Principle*.

Steps to create an analysis plugin In order to extend the CCAP with an analysis plugin, the following steps are required for a developer:

1. Create a `.py` file with a class inheriting from `AbstractAnalysisPlugin`.
2. Instanziate `PluginMetaData` and assign it to the metadata member.
3. Define a problem class inheriting from `AbstractAnalysisProblem` and set the problem name and description following the guidelines above.
4. Implement the `do_analysis` method with a `ParsedSourceFile` parameter. Return an `AnalysisReport` instance with all found problems.
5. Place the `.py` file into the analysis plugin directory of the tool.

While these are the minimum required steps to implement the plugin, the developer is free to add additional methods, classes or import libraries as necessary. Furthermore, it is advisable to implement several tests to ensure the plugins correctness. CCAP uses the pytest[1] library for testing. Implementing a test is as easy as writing a function with a `test_` prefix and running the `pytest` command. The plugin can be tested by simulating a call from the CCAP core with a mocked `ParsedSourceFile`.

Return None Plugin

A rather simple analysis plugin demonstrates the capabilities of the CCAP: The Return None Plugin scans the source code for functions that return the `None` value. Returning `None` may result in runtime exceptions if the function caller does not expect and handle a potential `None` return. Although `None` can be returned directly or as a value of the returning variable, this plugin only focuses on the explicit `return None` statement to showcase the options for the developer to analyse the source code. We will later refer to this problem type as `RETURN_NONE` (RN).

Detecting a `return None` statement is possible in multiple ways. The following shows the possibilities for a developer to implement such an analysis plugin:

[1] `https://docs.pytest.org/en/stable/`

Regular Expression: In the `do_analysis` method, a developer has access to the source code as a string. Therefore, it is straightforward to use a regular expression to detect a `return None` statement:

```
import re
matches = re.finditer(r"return None", source_file.content, re.MULTILINE | re.DOTALL)
```

Since the regex library only returns the start and end index of matches, these have to be converted to line numbers. Afterwards, a `ReturnNoneProblem` instance can be created for every match and added to the `AnalysisReport` for this plugin.

This approach uses regular expressions to match patterns in a string without utilising the structure of the code. It is a simple but powerful way, and most developers are familiar with regular expressions. On the flip-side, source code may have various syntactic ways to express a semantic. Consequently, the regular expressions have to be designed carefully to cover all variations. For instance, it is possible to encounter a doubled whitespace, that would break the aforementioned regular expression. Additionally, regular expressions do not operate on the structure of the code; therefore, they can not detect high-level patterns on the code structure.

Tokenization: The process of dividing a character stream into a sequence of tokens is called tokenization (also known as lexical analysis). With the Python tokenizer, a token can contain multiple characters and has a token type like name, operator or indentation. A token sequence provides more information about the code structure that can be used to detect problematic patterns. A token sequence for a `return None` statement would be the following:

```
[...( type: "NAME", value: "return"), (type: "NAME", value: "None")...]
```

A simple algorithm would scan the sequence for two subsequent name tokens with the values `return` and `None`. The abstraction level of a token sequence is higher than of a character sequence. Since the whitespace between `return` and `None` provides no semantic meaning, it is removed on this abstraction level. The regular expression-based approach would have to deal with problems like the doubled whitespace. In contrast, the token-based approach profits from the higher abstraction level and can access meaningful tokens directly.

Abstract Syntax Tree : After tokenization, a parser takes the token stream and parses it into a hierarchical data structure like an Abstract Syntax Tree (AST). With an

AST, the code is represented structurally, and it is possible to traverse the tree following the structure of the code. Analysing an AST allows a higher abstraction level then analysing the token string since the AST represents the code structure. Therefore, automated checkers for more abstract rules that analyse the code structure are possible.

To detect a `return None` statement in the AST, an algorithm would traverse all AST nodes looking for a return-typed node. If found, the `value` descendant contains the expression after the `return` keyword. A `None` value would be represented as a node of type `Constant` or `NameConstant`, depending on the parser version. The `value` descendant of the node may then be checked for equality to `None`. All AST nodes contain the corresponding line number that is necessary for creating a problem message. See Listing 3.2 for a Python implementation.

Analysing the AST works best to find problems in the code structure since the AST represents the code structure in a well-defined, traversable data structure. Although simpler patterns like the `return None` could be detected using regular expressions, a detection on AST level allows detecting the semantic meaning instead of the syntactic representation of a problem. For more complex problems, it is inevitable to use the AST most of the time and to combine it with other analysis techniques.

This implementation covers only the simple case, in which `return None` is written explicitly. Of course, several variations are possible, that would not be detected by this plugin and would require more sophisticated algorithms. Evaluating, if machine learning models trained on this simple rule can also detect such modifications is part of RQ3.

Since detection alone is not a great help for the user, a useful description of the problem is necessary. As described in Section 2.2.6, there are some alternatives to returning `None`. If returning `None` indicates an error, it is better to raise an exception and provoke an explicit error handling in order to prevent runtime errors. If the standard return type is

```
for node in ast.walk(a.ast):
    if isinstance(node, ast.Return):
        return_value = node.value
        if isinstance(return_value, ast.Constant) or isinstance(return_value, ast.NameConstant):
            if return_value.value is None:
                problems.append(ReturnNullProblem(a.file_path, return_value.lineno))
```

Listing 3.2: Detecting a `return None` by analsing the AST data structure.

a collection, an empty list is more appropriate, since the function caller will most likely write the logic for an unknown amount of list items. However, if the standard return type is a single object, it is not apparent what to return. With PEP484, an optional type was introduced as a type hint in Python 3.5 [51]. A static type checker for Python like mypy outputs a warning if a developer forgets to check an optional for not being `None` [52].

Condition with Comparison

The second plugin detects a direct comparison in a conditional statement. The clean code rules suggest using a function call in the body and the condition if the logic expression becomes more complicated. A function call allows a natural reading of the conditional statement without deciphering the meaning of the boolean logic or the effect in the body. As described in Section 2.4, checking for function calls in the body or condition is at a basic level of complexity. Designing a heuristic to decide when the expression in the condition becomes too complicated is at an advanced level of complexity. The latter is out of scope for this thesis.

For our implementation, we simplify the detection to the following two base cases: The first case is the detection of a direct comparison inside the condition. We will later refer to this pattern as `CONDITION_COMPARISON_SIMPLE` (CCS). A simple algorithm to detect a comparison in the condition would take the AST, search for `if` nodes, and check if the condition part is a compare node. However, the simple algorithm would not detect a negation and logic AND/OR conjunction. Consequently, a recursive algorithm has to follow logic operators and checks the expressions for comparison nodes. This simple approach does not provide value for the CCAP, but we use if for the machine learning evaluation.

We name the second case `CONDITION_COMPARISON` (CC). It is a superset of the CCS case and incorporates the considerations regarding negation and logic conjunction. It flags a condition as problematic if a sub-expression inside negations and logical conjunctions is not a function call. The more advanced algorithm for this problem would scan for `if` nodes in the AST. The conditional part as a boolean expression is evaluated in recursion. It can be a binary (like AND/OR) or a unary operator (like NOT). In these cases, the function will be called recursively with the respective sub-expressions. If the conditional part is neither a binary nor a unary operator, the function returns `true` if the single expression is not a function or method call. This last option would be the base case in the recursion. Listing 3.3 shows the Python implementation of this algorithm

```
def _check_if_direct_comparison(self, node):
    if isinstance(node, ast.BoolOp):
        violated = False
        # check all expressions of the binary operator
        for value in node.values:
            if self._check_if_direct_comparison(value):
                violated = True
        return violated
    elif isinstance(node, ast.UnaryOp):
        return self._check_if_direct_comparison(node.operand)

    return not isinstance(node, ast.Call)
```

Listing 3.3: Recursive function to analyse an if statement for direct comparisons (CC). Since a condition should contain a method call, the function returns False if this is not the case.

and the detected code patterns. Although this checker for the CC problems has too many false alarms due to the missing heuristic, it is helpful for the machine learning part of this thesis. As a superset of the CCS type, it allows us to train models on the CCS subset and evaluate their performance of unseen variations of the same problem on the CC set.

3.1.3 Output Plugins

With output plugins, CCAP adds additional flexibility towards the output format. Depending on environmental requirements, the output can be adapted with custom logic by defining an output plugin. For instance, an output plugin could write the results to the standard output in a human-readable way or create a formatted HTML file to be displayed in the browser. A machine-readable JSON output may be preferred when running inside an automated workflow.

Output plugins follow similar concepts as analysis plugins. A plugin class inherits from `AbstractOutputPlugin`. The abstract class introduces the `metadata` member of the `PluginMetaData` class to encapsulate plugin name and author information. Additionally, a second member variable `output_format` has to be defined with a short abbreviation for the output format. This field is used by the output plugin handler to select the output plugin by the run arguments. It should be globally unique, or the first found plugin will precede other plugins with equal naming. We suggest using a custom prefix to the format name to maintain global uniqueness.

As an entry point, the method `handle_report` has to be implemented. The method

provides an instance of `FullReport` as its argument. The `reports` field holds a collection of `AnalysisReport` for every analysis plugin that was executed. With metadata and problem information, the output plugin has access to all required information to produce the desired output.

Steps to create an output plugin To extend the output capabilities of the CCAP, the following steps are, similiar to analysis plugins, required:

1. Create a `.py` file with a class inheriting from `AbstractOutputPlugin`.
2. Instanziate `PluginMetaData` and assign it to the metadata member.
3. Set the `output_format` field with a unique abbreviation for this output format. Since it should be unique, a custom prefix prevents a name collision with preexisting plugins.
4. Implement the `handle_report` method with a `ParsedSourceFile` parameter.
5. Place the `.py` file into the output plugin directory of the tool.

Standard Output Plugin The Standard Output Plugin writes the formatted output to the `stdout` stream. It is enabled by default if no output plugin is explicitly specified.

The output is divided into a general part and the problem list. The former contains the input path, all executed plugins, and a summary field with the total number of problems. The latter displays the list with problems, grouped by analysis plugin. For each problem, the problem name, file path, line number and description is printed. A colon separates the file path and line number. Some terminals parse the path and line number so that a user can click the path and it opens the default editor with the cursor in the correct line. A sample output is printed in Listing 3.4.

HTML Output Plugin

For larger projects, the Standard Output Plugin may not be sufficient to get an overview of the project or to generate a report. Therefore, the HTML Output Plugin creates a `report.html` file that can be opened in modern browsers. The report provides a cleaner overview and the scrolling capabilities of a browser. It has similar general data at the top as the Standard Output Plugin, but it moves the problem description into a tooltip to offer a more compact overview. See Figure 3.2 for an example screenshot.

```
Analysis Report on /Users/enrico/MA_CleanCodeAnalyser/test_programs.
Analyse Plugins: Condition Method Call Plugin, Simple Condition Method Call Plugin, Return None
    (Null) Plugin, Sample Plugin.
Summary: Found 16 problem(s)
    ----------------------------------------
PROBLEMS:
    PLUGIN NAME: Condition Method Call Plugin by Enrico Kaack <e.kaack@live.de>
        Found: Explicit comparison in condition in /Users/enrico/MA_CleanCodeAnalyser/
    test_programs/return_none.py:18
            Explicit comparisons in conditions should be replaced by method call for better
    readability
...
PLUGIN NAME: Return None (Null) Plugin by Enrico Kaack <e.kaack@live.de>
        Found: Returned None in /Users/enrico/MA_CleanCodeAnalyser/test_programs/
    return_none.py:4
            Returning None is dangerous since the caller has to check for None. Otherwise, a
    runtime exception may occur.
```

Listing 3.4: Example output to `stdout` of the Standard Output Plugin. Besides a list of problems, it also outputs further information such as input directory and analysis plugins.

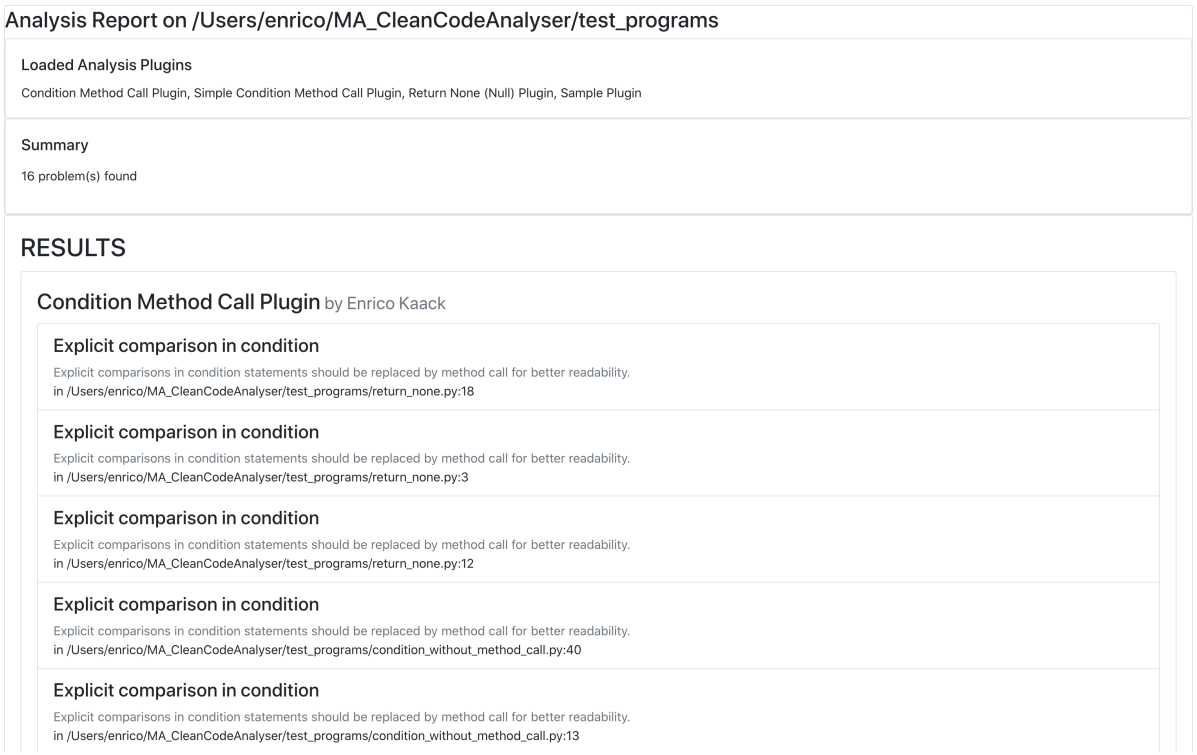


Figure 3.2: The output of the HTML Output Plugin, displayed in a browser.

3.1.4 Improvements

While the core functionality exists, extensions for the CCAP would aim for improved useability. One crucial improvement in useability would be an IDE integration for common IDEs like Visual Studio Code. A good starting point would be the Language Server Protocol[2]. This IDE integration requires a language client as a Visual Studio Code extension and a protocol-compliant language server. The latter would wrap the CCAP and modify it slightly to process a single, changed document and return the results. Since the CCAP architecture is modular, this should be possible without larger modifications.

Another feature would be a configuration to disable specific analysis plugins for a project. With the current architecture, this is be possible by moving the analysis plugin file outside the plugin directory. A configuration with command-line parameters or with a hidden config file per project would have a better user experience. The latter could also be committed to the version control system, so every team member has the same configuration.

Continuing on configuration possibilities, plugin configurations would increase the flexibility for a plugin developer. Introducing configurable warn levels like `warning` or `error` could help in continuous integration pipelines to decide if a build succeeds with warnings or if a build should fail because of error-level problems. Some rule violation could be seen as recommendations to improve (warning level), whereas other violations may be unacceptable (error level).

Lastly, a feature to disable problem reporting for a specific code location could bring a boost in user acceptance. While some clean code rules are objective and can be measured precisely, some rules may not apply to every occurrence of the situation. For instance, the clean code guidelines generally suggest not to have more than three function arguments; it may be necessary or even inevitable to have four arguments. The CCAP would report this as a problem, but the user should be able to decide if it is acceptable. In this case, the problem type on this specific location should not be reported in the future. In the current version, the user has no way to ignore a specific problem or mark it as false alarm. Consequently, the number of problems the user has to ignore will accumulate over time until the user abandons the tool since it does not provide an added value over the frustration of manually ignoring problems.

[2] `https://microsoft.github.io/language-server-protocol/`

3.2 Clean Code Classification

In the previous chapter, we presented a platform to check for clean code rules. This platform requires analysis plugins to detect specific problems. These handwritten rules work well for rules with a basic level of complexity. However, certain rules with an advanced level of complexity may require a complicated analysis of the AST, the data flow or the inter module dependencies. Furthermore, rules with a high level of complexity can be subjective, and we do not see a way of detecting those rule violations with a deterministic algorithm.

Therefore, this chapter introduces an approach to evaluate different supervised machine learning models to classify code into clean code or problematic code. Furthermore, we describe how we modify the code to test the generalisation capabilities of the models. The mentioned machine learning models include random forest classifiers, support vector classifiers, gradient boosting classifiers and recurrent neural network models based on Long Short Term Memory (LSTM) units.

The objectives are twofold: For RQ2, we train, evaluate and compare the different models on the detection of different code violations. Then we manipulate the source code to test in RQ3, how well the models generalise to non-seen variations of the rule violations.

To achieve our objectives, we will train our models on the detection of three problem types:

1. The `RETURN_NONE` (RN) problem type covers a `return None` statement in Python code (see Section 3.1.2). A detector would report such a statement as this problem type.

2. A detector for the `CONDITION_COMPARISON_SIMPLE` (CCS) problem type detects a comparison operator in the condition of an if-statement. It does not detect the comparison if it is nested with logical expressions. The detection algorithm is further described in Section 3.1.2.

3. With the `CONDITION_COMPARISON` (CC) type, we describe whether the condition inside an if-statement contains other expressions then boolean logic and function calls. The CC type is a superset of the CCS problems.

From the three problem types, we expect the RN type to be the easiest type to spot since the model only has to spot the same statement in every sample. The CC type is

more complex to spot, and the model would have to learn the connection between the if-statement and the possible logic expressions. The CCS type should be easier to detect than the CC type due to fewer variations. Additionally, we use the trained model on the CCS subset and test it with a manipulated CC superset to assess the generalisation to unseen variations in RQ3.

In the following, we describe the approach for the machine learning part in detail. First, we discuss the challenges and our solutions. Then we introduce our dataset and our split in train, validation and holdout sets. Afterwards, we describe our pipeline of processing the raw files into samples with a fixed size, our encoding approach and our code manipulation for RQ3. In the final section, we describe the machine learning models and parameters we use for all research questions.

3.2.1 Challenges

Lack of Datasets

The most crucial preconditions of supervised machine learning experiments are labelled datasets. For clean code detection, we did not find a suitable dataset with labelled clean code violations in our research. As a result, we evaluated different approaches to create such a dataset:

Solution 1: Related research fields like code completion often use the py150 dataset to train models [53]. This dataset contains 150.000 python files, sourced from GitHub. For our supervised classification tasks, this dataset is inadequate due to its missing labels.

Solution 2: Another possible data source could be git commit histories. We could scan the history for commits that fixed unclean code. The previous code could then be labelled "unclean" and the committed code would be "clean". The same logic could be used on issues and referenced pull requests on GitHub. We dismissed this approach for several reasons:

1. There is no standardised annotation in neither issue descriptions nor git commit messages that would indicate reliably if the code change is fixing a violation of the clean code rules.

2. Even if we would be able to find commits that improve chaotic code, we could not ensure automatically, that the commit message is correct and only the improvement is included in the commit.

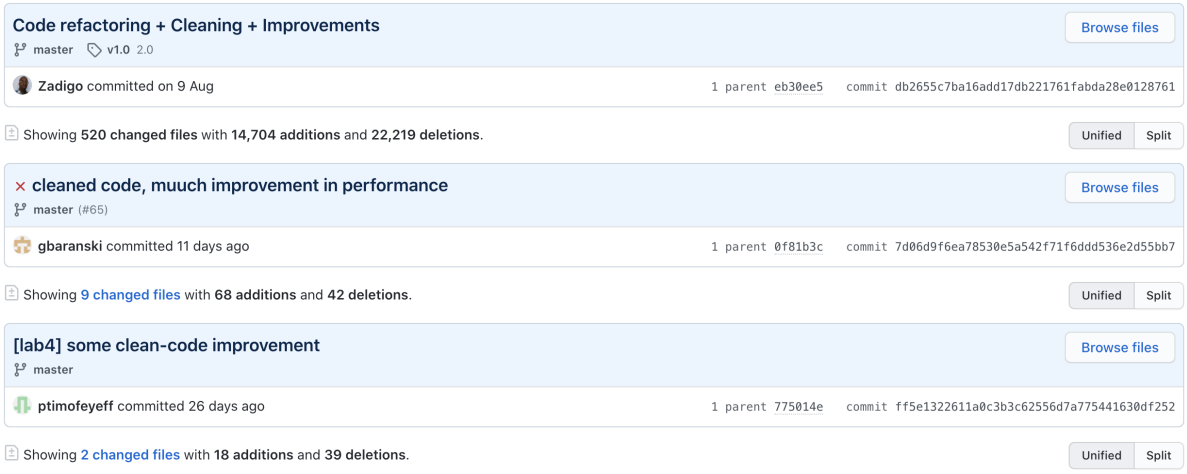


Figure 3.3: Example commit messages from different repositories. Those examples highlight (1) the missing naming convention, (2) inclusion of additional performance improvements, and (3) several edits in multiple files [54, 55, 56]. Those commits were found by searching GitHub for "clean code improvements" and filtering the results for type=commits and language=Python.

3. Searching for commits, we found several clean code improvements and refactorings of large portions of the codebase in one commit. Additionally, it is often not explicitly mentioned, which rule applied for the improvement. Consequently, training with such data could only lead to binary classification into clean code or unclean code. From a practical standpoint, it is advantageous to name the explicit rule violation and to explain how to improve.

Figure 3.3 offers an illustrative example of commit messages from different projects. First, there is no consistent annotation schema to display clean code optimisation. Second, a commit may include multiple changes like the second example. Finally, the first commit shows a large refactoring that most likely fixes several problems and not only clean code violations.

Solution 3: Since Python code is available in abundance in the form of open-source projects, it is possible to manually label source code that violates clean code rules. For the following reasons, we dismissed this approach:

1. Curating a hand-labelled dataset is a time-consuming task.

2. Manual labelling does not ensure correct labelling. A human can make mistakes or can subjectively misinterpret chaotic code as clean code. The quality of the dataset could suffer and limit the machine learning models performance.

Label	RN	CCS	CC
0	99.79%	97.36%	95.14%
1	0.21%	2.64%	4.86%

Table 3.1: Class distribution for the dataset. Label [0] represents clean code and label [1] marks problematic code of the corresponding category.

Solution 4: Given any amount of Python files, we could use the analysis plugins from Section 3.1.2 to reliably detect violations of the corresponding rule. Based on this labelled data, we could train the classifier to detect those rule violations. The analysis plugins only cover rule violations of basic level of complexity, so the trained classifier could only detect rule violations that could also be detected by a basic checker. However, if there would be a labelled dataset for more complex clean code rules, we could transfer the findings to these more complex rules.

We choose this solution to solve the dataset problem and train classifiers. We can automatically generate labelled data with correct and consistent labels. Furthermore, this solution allows for scaling the dataset size if necessary. We will use the analysis plugins to label the three problem types: `RETURN_NONE` (RN), `CONDITION_COMPARISON_SIMPLE` (CCS) and `CONDITION_COMPARISON` (CC). However, this implies that we train the classifier to detect rule violations that could be detected by simple analysis plugins. If a labelled dataset for rules with a high level of complexity becomes available, we hope to transfer our findings to those rule violations as well. In RQ3, we will then test those classifiers on unseen variations of the rule violations to test the generalisation of the models. This gives an outlook of the transfer to the training on rules with a high-level of complexity by simulating a hand-labelled training set that is limited in problem variations.

Imbalanced Dataset

A second challenge is a major imbalance between the samples labelled as problematic code and clean code. For the three different problem types, RN, CCS, and CC, the class distribution is shown in Table 3.1. The imbalance is especially severe for the RN type since only two out of 1000 samples contain the rule violation. For the CCS and CC type, the classes are slightly more balanced with 26 and 49 samples with rule violations out of 1000.

We combined the following solutions:

Solution 1: Due to the imbalance, we do not choose accuracy as a suitable evaluation metric, since classifying all samples as clean code would result in an accuracy of 99.79% for the RN problem type. Instead, we use precision and recall as metrics, with an F1 score as a single, weighted metric. More details about the metrics can be found in Section 4.1.2.

Solution 2: We will apply undersampling and oversampling techniques on the data and evaluate the differences in model performance. Undersampling removes random samples from the majority class and thus balances class labels. Oversampling replicates random samples from the minority class to balance class labels while also increasing the overall amount of data. This technique is only used for the training data since it would distort the evaluation results when applied to validation data.

Solution 3: Our approach of training different classifier is a solution to the data imbalance since different classifiers have a different sensitivity to data balance.

SVM Quadratic Complexity in Training

The implementation of the support vector machine has a quadratic time complexity on the number of training samples [57]. Consequently, the training takes a lot of time, and the oversampling strategy worsens the training time further since it introduces more samples.

Solution: We reduce the number of experiments with SVM and evaluate some simple cases for baseline performance.

3.2.2 Dataset

The sources of our dataset are open-source Python projects on GitHub. We queried the top starred Python repositories and handselected 18 repositories. Although most top stared repositories are data science frameworks, we explicitly choose projects from other domains such as web server, automation and containerisation. In Table 3.2, we list all projects with the corresponding domain.

A script downloaded all projects in their main branch with the current head. See Table A8 for the corresponding git hashes. Afterwards, we removed all non-python files. Next, we uniformly sampled 20% of all files and separated those into a holdout set for testing and the code manipulation in RQ3. Additionally, we perform our train/validation

Organisation	Project	Domain
scikit-learn	scikit-learn	Data Science
pytorch	pytorch	Data Science
explosion	spaCy	Data Science
bokeh	bokeh	Data Science
pandas-dev	pandas	Data Science
keras-team	keras	Data Science
django	django	Web Server
pallets	flask	Web Server
tiangolo	fastapi	Web Server
tornadoweb	tornado	Web Server
encode	django-rest-framework	Web Server
jakubroztocil	httpie	Web Requests
home-assistant	core	Home Automation
certbot	certbot	Automation
google	python-fire	Automation
apache	airflow	Automation
ansible	ansible	Automation
docker	compose	Containerization

Table 3.2: Open-source repositories we used in our dataset and their corresponding domain.

split on the file level and not on the sample level (see Figure 3.4). We describe the reason for the file level split later in Section 3.2.3.

We ensured to have similar data and label distributions on all data sets. Table 3.3 shows general and problem-specific metrics on file level for the train, validation and holdout set. The problem-specific metrics are collected after processing the code files as described in Section 3.2.3. We define the metrics as follows:

We calculate the *average lines of code per file* for n files with l_i as the lines of code for file i with the following equation:

$$\textit{average LOC per File} = \frac{\sum_{i=0}^{n} l_i}{n}$$

For the *proportion of lines of code containing a problem*, we assume one line contains the problem. Therefore, we define this metric for n files with l_i lines of code for the i-th file and p_i problems in the i-th file as follows:

$$\textit{LOCs containing problem} = \frac{\sum_{i=0}^{n} p_i}{\sum_{i=0}^{n} l_i}$$

	Metric	Train	Val.	Holdout
	Lines of Code	2,530,455	246,813	671,554
	Number of Files	13,330	1,481	3,702
	average LOC per File	189.83	166.65	181.40
Return None	LOCs containing problem	0.07%	0.09%	0.08%
	Problems per File	0.14	0.15	0.14
Condition Comparison Simple	LOCs containing problem	1.29%	1.21%	1.32%
	Problems per File	2.44	2.02	2.4
Condition Comparison	LOCs containing problem	2.34%	2.31%	2.44%
	Problems per File	4.44	3.85	4.42

Table 3.3: General and problem-specific metrics for the train, validation and holdout set.

Last, we calculate the number of problems per file for n files and p_i problems in the i-th file as:

$$\textit{Problems per File} = \frac{\sum_{i=0}^{n} p_i}{n}$$

The most important metrics for the data distribution is the proportion of lines of code containing the problem type and the average number of problems per file. For lines of code containing the problem type, we observe a maximal difference of 0.02 percentage points between the datasets for the RN problem type, 0.11 percentage points for CCS and 0.13 percentage points for the CC problem type. This directly translates into the class distribution as shown in Table 3.4 with a maximal difference of 0.06, 0.26 and 0.31 percentage points for the three problem types. The number of problems per file is lower in the validation set for the CCS and CC problem type. Nevertheless, the size of a file may vary, and thus the number of problems per file may vary too. Since the lines of code containing the problem and the class frequencies are comparable, our datasets have a comparable data distribution over the different dataset splits.

3.2.3 Processing

For the processing steps, we create a pipeline with the d6tflow framework[3]. Each processing step is a task that stores its results depending on the parameter configuration on the disk. This allows to define a pipeline once and run it with different parameter combinations. The scheduler automatically determines what tasks to run and what task

[3] `https://github.com/d6t/d6tflow`

Problem Type	Label	Train	Validation	Holdout
Return None	[0]	99.79%	99.73%	99.78%
	[1]	0.21%	0.27%	0.22%
Condition Comparison Simple	[0]	97.34%	97.49%	97.23%
	[1]	2.66%	2.51%	2.77%
Condition Comparison	[0]	95.13%	95.21%	94.9%
	[1]	4.87%	4.79%	5.1%

Table 3.4: Class frequencies for all problem types on the train, validation and holdout set. We observe a comparable class distribution among the datasets for each problem type.

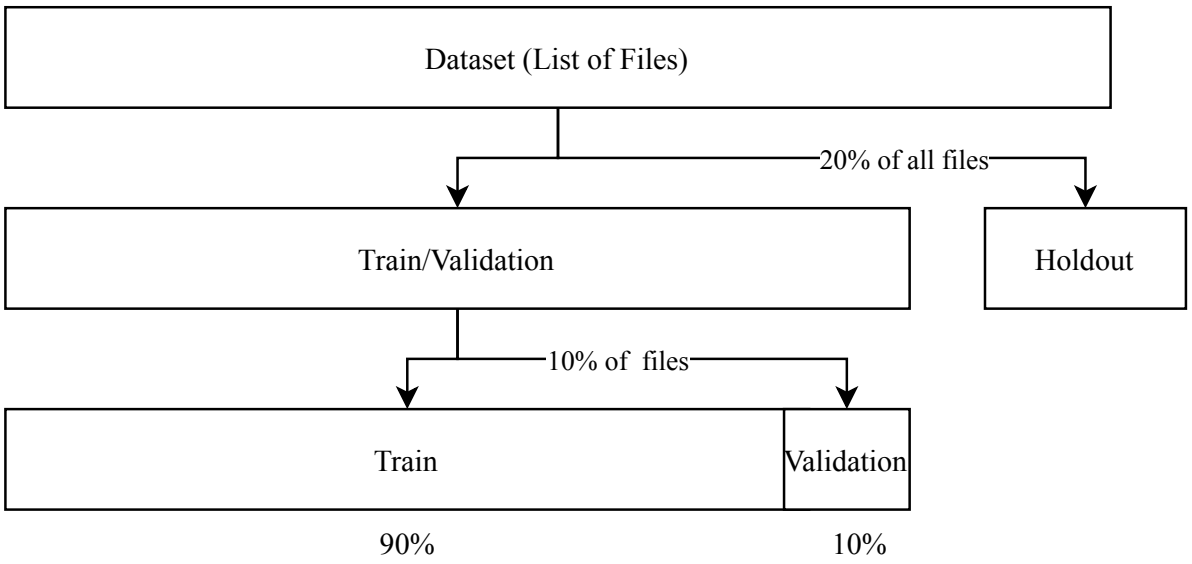


Figure 3.4: Visualisation of the dataset split. First, 20% of all files are separated into a holdout set for later testing. The remaining 80% of the files are split into 90% training and 10% validation data.

outputs are already stored. Consequently, running the pipeline is more efficient and repeatable.

The processing pipeline consists of tasks to read the files into a data structure, to process source code and find the problems, to create a vocab dictionary, and to convert the data into labelled samples. In Figure 3.5, we provide a schematic representation of the pipeline.

Files into Internal Data Structure

First, a scanner walks recursively over all subdirectories of the input folder to find all files (pipeline task `TaskSourceFileToDataStructure`). If the file has a `.py` extension, its file path and the file content will be stored in a dictionary. The dictionaries for all files are collected into a list. Reading all files into main memory increases the performance for downstream tasks since main memory access is faster than disk access. Although the size of the system's main memory limits the dataset size, a file contains text and is just several kilobytes in size. The training dataset with 13,330 files is 86.47MB in accumulated file size.

Problem Detection

As a next step, the analysis plugins from the CCAP (see Section 3.1.2) process every file and store the line number along with the problem type (RN, CCS, or CC). As a result, for every file, a list of problematic line numbers and the corresponding type is available for further processing). In our pipeline, the task `TaskRuleProcessor` handles this processing.

Data Encoding

The data encoding step transforms the internal data representation into the input vector x and output vector y. One (x, y) pair describes a sample with input and ground-truth label. The transformation happens in the following multi-stage process:

Fixed Length Sequence The length of the source code is dynamic, whereas the input size of our models is fixed. Therefore, the character stream of variable length has to be transformed into a token stream of fixed size. To extract meaningful tokens from the character stream, we use the Python tokenizer from its standard library. The tokenizer separates the character stream into tokens based on the Python syntax definition. All

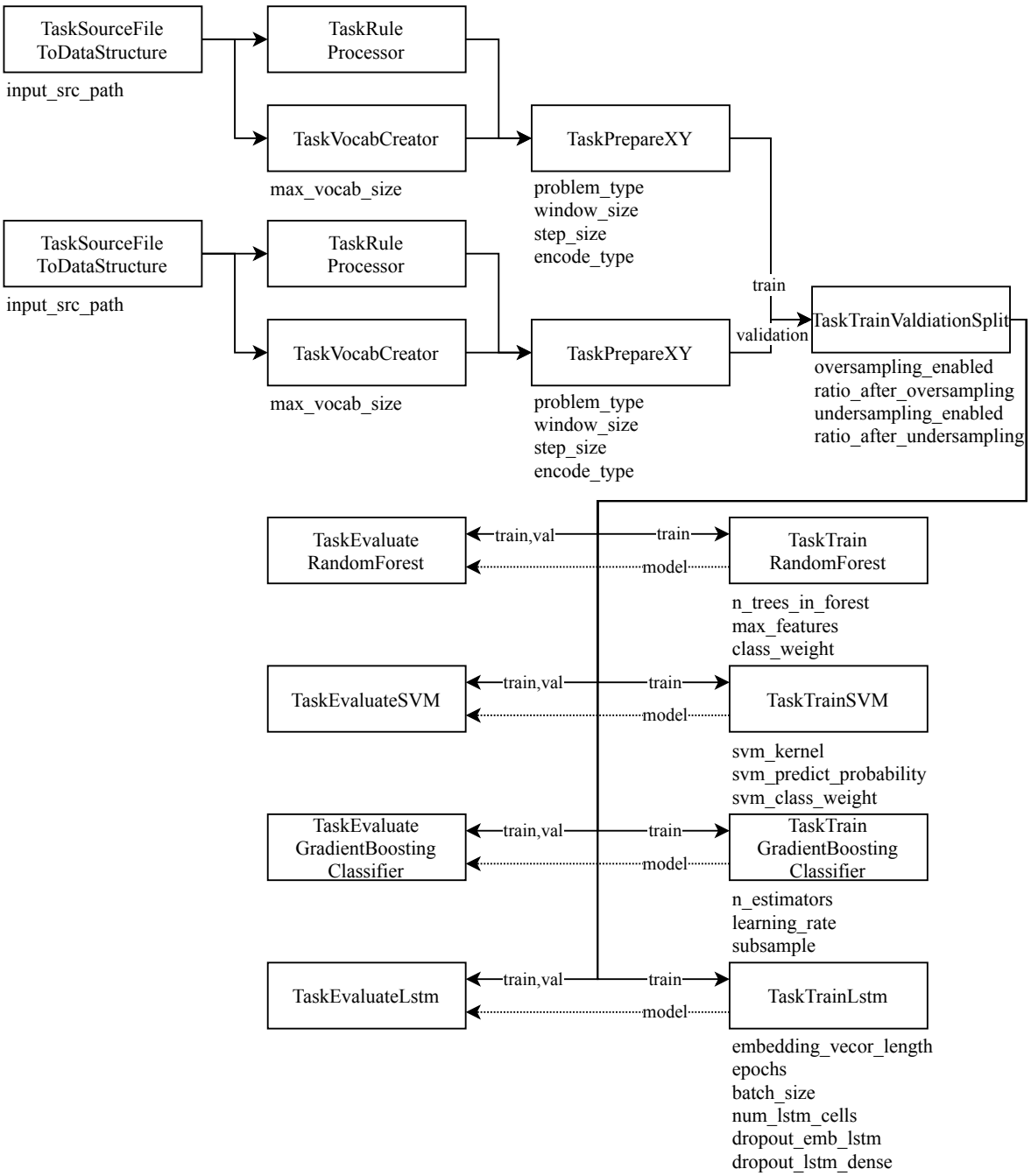


Figure 3.5: Schematic representation of the pipeline for RQ2 to train and validate models with configurable parameters. First, the pipeline reads the source code into memory (`TaskSourceFileToDataStructure`), extract the vocabulary (`TaskVocabCreator`) and identify all problem types (`TaskRuleProcessor`). It then converts the code of dynamic length into fixed-sized samples (`TaskPrepareXY`). This sub-pipeline runs separately for the train and validation set. In `TaskTrainValidationSplit`, both datasets are combined and over- or undersampling is applied to the train set. The train set is used to train different classifiers, whereas the train and validation set are used for evaluation.

tokens contain a token type (like a name or operator token), the corresponding characters in the source code and a start and end position (line and column number). Afterwards, the token stream is divided into a fixed size token sequence using a sliding window approach. The step and window size are configurable, and we use a window size of 20 and a step size of 3 in our experiments. The sliding window operation is part of the `TaskPrepareXY` step in the pipeline.

Vocabulary Creation In order to feed the textual token values into a model, we need to transform it into a numeric representation. Since our token values can be the entire value of a string literal, we can not encode every single token value into a number. We create a vocabulary with a limited size that contains the most common token values and a numerical representation for all other values. Therefore, the occurrence of each token value is counted and sorted based on their frequency. To ignore potential capitalisation mismatches, all token values are lower-case. The overall size of the vocabulary is configurable. If the size is smaller than the size of the distinct token value set, the least common token values will be replaced by an unknown token. The unknown token has a numeric representation one bigger than the vocabulary size. We found that using a vocabulary size of 100,000 results in an acceptably low 3.9% of token values being unknown. In our pipeline, the task `TaskVocabCreator` creates the vocabulary.

Index-Based Encoding The textual token values are encoded with an Index-Based encoding. The most common token values have a numerical representation based on their index in a vocabulary (created with all token values of the train dataset). Optional, the type of each token is encoded by its numerical representation in the standard library (see Figure A1). The type and value encodings are then combined alternating in a final encoding vector x_i for every sample i.

Label Extraction With the internal representation after the problem detection step, the ground truth is encoded as a line number. This representation has to be transformed in a label y_i for the given input vector x_i for sample i. Label `1` is assigned if the sample contains the given problem type. A sample contains a problem type if it contains all tokens from the problematic line. If the sample is missing a token of the problematic line at the beginning or end, the label `0` for non-problematic code is assigned. After the label extraction in the pipeline step `TaskPrepareXY`, we have a list of input vectors x and a list of ground truth labels y. With this, we can train and test our models.

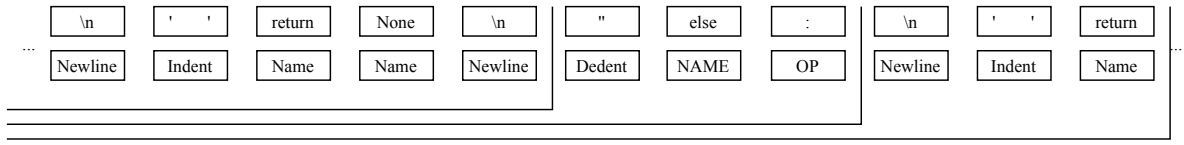


Figure 3.6: Sliding window approach on source code containing a problematic `return None`. Multiple windows will contain the problematic sample. Consequently, we can not split on a sample level, since the same problem would be in the train and validation set. Instead, we split on file level to have a clean separation between the datasets.

Train/Validation Split

In Section 3.2.2, we described how to create a train, validation and holdout dataset by file-level separation and not by splitting the samples. The reason for this approach is our sliding window concept. Since we use the sliding window to convert source files with dynamic length into fixed-length token samples, we may have one problematic code pattern in multiple samples. The difference would be in the location inside the sample and the surrounding tokens, whereas the pattern remains the same. If we would do a train/validation split on a sample level, we may introduce samples covering one problematic pattern in both datasets. Figure 3.6 illustrates multiple windows covering one sample.

This unclean separation would diminish the validity of metrics calculated from the validation and holdout dataset. Therefore, we split on file level and perform the previous preprocessing separately for training and validation dataset. Only the vocabulary generation happens on the training file for both processing steps. In `TaskPrepareXY` in the pipeline, we then merge those two pipelines for training and validation (see Figure 3.5).

Code Manipulation

In research question RQ3, we evaluate the models' generalisation to detect similar patterns the model was not trained on. Therefore, we manipulate the code so that it is still a rule violation, but the original analysis plugins would not spot them. The pipeline for code manipulation differs from the pipeline for RQ2 by an additional code manipulation task and removal of the training step (see Figure 3.7).

Return None For the RN problem type, we manipulate the code to have an inline if statement that only returns `None` in one branch. Therefore, a function may still return

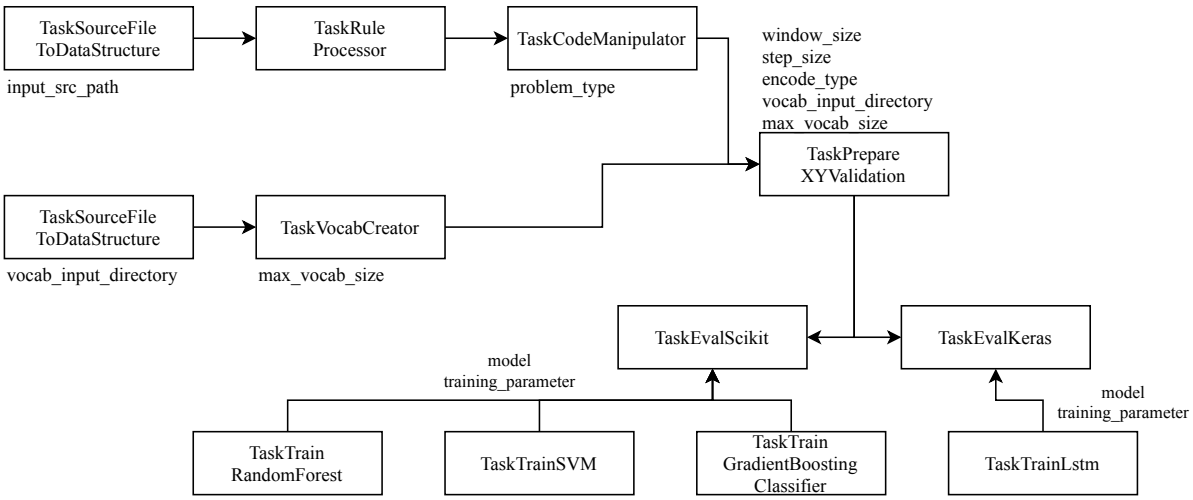


Figure 3.7: Pipeline for code manipulation and evaluation. The preprocessing part is like in Figure 3.5 but with an additional *TaskCodeManipulator* to manipulate the code as described in Section 3.2.3. The random forest classifier, support vector classifier and gradient boosting classifier require different evaluation code as the LSTM, so two *TaskEval** are used.

`None`, but the analysis plugin would not detect it. Listing 3.5 shows an example for this case. Although a modification of the analysis plugin could cover the variations as well, we want to see how well machine learning models could perform on this task.

To modify the original code, we use the preprocessed code with problems detected as shown in Section 3.2.3. For every source code line containing the RN problem, we use regular expressions to replace the `return None` with a variation as seen in Listing 3.5.

The subsequent data encoding step is similar to the one for model training, although the modified samples will only be used during evaluation.

```
def f(a,b):
    # detected by the analysis plugin
    return None

    # not detected by the analysis plugin
    return None if a < b else b

    # not detected by the analysis plugin
    return a if a < b else None
```

Listing 3.5: Samples for returning None. The analysis plugin would flag the first return; the second and third return are modified variations that would be ignored by the analysis plugin. The performance of the machine learning models on detecting the latter will be evaluated.

Condition Comparison We want to test the generalisation of the model trained on the CCS types by evaluating on the labelled data for the CC type. Since the dataset for the CC type includes the CCS subset, we manipulate the CC dataset to only contain samples the CCS was not trained on. The sample in Listing 3.6 shows the manipulation to ensure that all CC problems were not part of the CCS training set. The implementation uses a similar regular expression approach, as described in Section 3.2.3.

```
def f(a,b):
# detected by CONDITION COMPARISON SIMPLE
# detected by CONDITION COMPARISON
if a < b:
    pass

# manipulated
# not detected by CONDITION COMPARISON SIMPLE
# detected by CONDITION COMPARISON
if not a < b:
    pass

# manipulated
# not detected by CONDITION COMPARISON SIMPLE
# detected by CONDITION COMPARISON
if is_smaller(a,b) and b < c::
    pass
```

Listing 3.6: Sample statements for the difference between the two analysis plugins CC and CCS.

3.2.4 Models

Classifying code samples is a binary classification problem since we consider each problem type as a separate classification task. We train and evaluate a random forest classifier, a support vector classifier, a gradient boosting classifier and a neural network with LSTM cells.

Random Forest Classifier We use a random forest classifier with 100 decision trees to perform the binary classification. Additional to over- and undersampling, we experiment with an automatic class weighting inverse proportional to class frequency. The other hyperparameter follows the default implementation of the scikit-learn library: The quality measure of a split follows the Gini function, the tree depth is unlimited and the number of features to consider for each split is the square-root of the number of features overall. We use the implementation in the scikit-learn library in version 0.23.2 [58].

Support Vector Classifier We use the support vector classifier (SVC) from the scikit-learn library [58]. As a kernel function, we choose the radial basis function for a performance baseline. Additionally, we try the class weighting inverse proportional to the class frequency to combat the imbalanced dataset.

Gradient Boosting Classifier For the gradient boosting classifier, we mainly vary the number of boosting steps and the learning rate. Furthermore, we experiment with stochastic gradient boosting by setting the subsample parameter to 0.4 and 0.7. Other hyperparameter settings follow the default value of the scikit-learn library [58].

Neural Network with LSTM-Cells Our neural network consists of five layers (see Listing 3.7). First, we use an embedding layer with an embedding size of 32. The input are the index-based, encoded samples without type encoding. This layer is trained end-to-end with the complete network on the training data. Furthermore, this layer covers most trainable parameters of the model. As a hidden layer, we use 10 LSTM-cells with a prior and posterior dropout layer with a 0.2 dropout. To perform binary classification, our model ends with a dense mapping onto a single neuron. The binary result is determined based on a 0.5 threshold for the output of the last neuron.

We vary the hyperparameters for batch size, number of epochs, embedding size and number of LSTM cells. For the implementation, we use Keras in version 2.4.3 [59].

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 20, 32)            3200064
_________________________________________________________________
dropout (Dropout)            (None, 20, 32)            0
_________________________________________________________________
lstm (LSTM)                  (None, 10)                1720
_________________________________________________________________
dropout_1 (Dropout)          (None, 10)                0
_________________________________________________________________
dense (Dense)                (None, 1)                 11
=================================================================
Total params: 3,201,795
Trainable params: 3,201,795
Non-trainable params: 0
```

Listing 3.7: Summary of our LSTM network. We use an embedding layer of size 32, 10 LSTM cells, a dropout layer before and after the LSTM layer with a dropout ratio of 0.2 and a dense mapping to a single output neuron for binary classification with a threshold of 0.5.

4 Quantitative Evaluation

In this chapter, we will evaluate the approach described in Chapter 3. First, we will recap the research questions. Afterwards, we will describe the motivation, details of the approach, and the results per research question. We explain the results we observe and recommend possible improvements.

4.1 Research Questions

The main objectives of this work are twofold: First, we design and implement the CCAP as a platform for automated code checking. We compare our platform with preexisting tools and validate if the CCAP can supplement or replace an existing tool. Second, we use machine learning models to detect violations of a clean code rule. We train models to detect a specific problem pattern and evaluate the generalisation capabilities on unseen pattern variations. By testing on unseen pattern variations, we want to simulate a real-world scenario for checking for rules with a high level of complexity. For rules with a high level of complexity, we can not generate a labelled dataset automatically. Instead, we have to hand-collect samples, that may be limited in variety.

We formulate this objectives into the following research questions that will lead this evaluation:

RQ1: What is the utility of the CCAP besides existing tools?

RQ2: How do different machine learning models compare on the task of detecting non-clean code?

RQ3: Do machine learning-based models cover a larger variety of cases than rule-based checker?

4.1.1 RQ1: What Is the Utility of the CCAP Besides Existing Tools?

Motivation Several tools are established that aim to improve code quality and detect unclean code (see Section 2.6). Every new tool with similar claims has to prove its usefulness besides the preexisting tools. For this reason, we compare the CCAP with preexisting tools to evaluate its utility.

Approach To answer this research question, we compare the CCAP with existing tools based on the design goals extensibility, useability and integration. Additionally, we will mention and compare useful features unique to existing tools. Finally, we assess if it can supplement or replace existing tools. As described in Section 2.6, the compared tools are Sonarcube, PMD Source Code Analyser Project, Codacy and PyLint.

Results

Finding 1: Extensibility The extensibility of CCAP with analysis plugins is comparable to PyLint, that also offers plugins to analyse the raw string, the token stream or the AST. PMD allows plugins to analyse the AST or define XPath rules. Since XPath rules can be used in the CCAP as well (using a third-party library in the plugin), PMD offers fewer expansion possibilities. Sonarqube, on the contrary, provides the most possibilities for extension. Not only can developers analyse the AST or specify XPath rules, but they have access to an additional semantic model of the code that provides direct access to structures such as methods, parameters, and return values. This semantic model simplifies an analysis in comparison to the AST analysis. Additionally, Sonarqube allows plugins to expose an API for other plugins to use. With plugins that expose functionality to other plugins, the possibilities for rule checking plugins increases further and functionality can be reused. The least expandability offers Codacy that allows customising or disabling existing rules. However, it does not offer to add rules to the existing ruleset.

Regarding output plugins, no compared tool offers the output customisation of CCAP. PMD and PyLint have different predefined output formats like JSON, HTML, CSV, or text. Although PyLint allows customising the message format using a formatting string as a command-line argument, they do not offer extensions for the output. Sonarqube displays the analysis reports in its WebUI. The scanner tool, SonarScanner, sends the reports to the server component, that renders the results in the browser. Codacy's local

scanner produces text output; the cloud version also has a WebUI. With the extension possibility of CCAP, we offer the developer a simple mechanism to adapt the output to its own needs. While this feature is unique, the result could also be achieved by parsing the JSON output of e.g. PyLint in a subsequent step.

Finding 2: Integration The CCAP has significant shortcomings in its integrations into IDEs, build processes and CI pipelines. All contestant tools provide plugins for build systems like Gradle or Maven. Except for Codacy, all tools have IDE integrations into most common editors. Sonarqube and Codacy have integrations into source control systems. Same applies to PMD and PyLint when bundled with Codacy.

Finding 3: Useability ISO 9241 defines useability as the "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [60]. For the tools, we identify two main user groups: the plugin developers want to encode a rule to be checked automatically, and the developers want to analyse their code. We compare the useability for the tools with the focus on the efficiency, measured as the required time to achieve a useful output.

For a plugin developer, the useability of the plugin system is simple for CCAP and PyLint. Both tools have the same plugin mechanism and a simple plugin interface. A simple interface reduces the required time to understand and use the interface. PMD has a similar, straightforward plugin interface, but its Java plugin requires an extra bundling step before adding to the classpath. The latter applies to Sonarqube as well; additionally, the powerful and rich plugin API of Sonarqube is contrary to the simple plugin interface of CCAP and PyLint. While this opens more options to the extension developer, it requires more time to understand the possibilities. Especially, if a developer wants to quickly encode a problematic pattern that does not fit an XPath rule, the more complicated plugin interface of Sonarquabe can decrease the efficiency.

For the developer, installing CCAP is like installing every other Python package. Running it from the command-line is fast and can be automated, e.g. using git pre-commit hooks. The same workflow could be achieved with PyLint and PMD. Sonarqube is more complex to install due to its multiple components. However, after the setup, the WebUI and integrations into most developers workflows have a comparable or better useability than CCAP or PyLint. Codacy, as a cloud service, has the simplest setup by granting access to the source code repository to start scanning the code. The setup of

the local scanner requires a configuration with a project token to retrieve information from the cloud service.

Finding 4: Multi-Language Scanning Sonarqube, PMD and Codacy offer multi-language scanning. This multi-language ability is advantageous for code repositories with multiple languages and for reusing the same tools and infrastructure for multiple projects with different programming languages.

Supporting multi-language scanning is not possible with CCAP. It would require a modification of the architecture: The source handler would have to scan the files concerning their programming language and transform those into a language-independent representation. Other parts of the architecture could remain similar. Although multi-language support was not a design goal, it would be a significant advantage for adoption.

Finding 5: Maturity and Community Support The maturity and community-support of tools impact the integration of the tools into the developer workflow. Some integrations are community-made and shared, so everybody has an advantage. CCAP has no community support since it has not been released. The community support would be necessary for CCAP to write and share additional analysis plugins and to integrate into different workflows.

Summary In summary, CCAP offers a good, but not best in class extensibility with analysis plugins. It has a high useability for plugin developer and users. The lack of integrations for different workflows reflects the lack of community support and maturity. We see the CCAP as an addition to the powerful Sonarqube. The simple, yet robust expandability of the CCAP could supplement the shortcomings of Sonarqube in its powerful, but complex expandability. For instance, a code reviewer might identify an unclean code snippet, writes the corresponding new plugin to identify this problem in the future, distributes it to the team, and would hopefully not reencounter the same problem. With Sonarqube, the effort to understand the plugin interface may surpass the time the reviewer is willing to invest. Additionally, the CCAP could be used to teach about clean code and to enforce specific coding rules for students' exercises, since the local setup is straightforward. Clean code rules could be turned into analysis plugins by students or teachers without having to understand a complicated interface or requiring a server setup.

With an increase in adoption and further development, the drawbacks of integration and community support would diminish. It has the potential to become an equal

contestant to PyLint.

4.1.2 RQ2: How Do Different Machine Learning Models Compare on the Task of Detecting Non-Clean Code?

Motivation As we have shown with the CCAP and the analysis plugins, it is possible to write an algorithm that can detect specific code patterns like non-clean code. If we can write a well-tested detection algorithm, we do not need error-prone machine learning models to detect code patterns. Nonetheless, if we can not design an algorithm that detects the desired code pattern reliably, we may perform better using a machine learning model. Additionally, if we encode a subjective pattern, it may not be possible to design an algorithm to detect such a pattern objectively. As a solution, we could label code parts as our pattern and use machine learning approaches to extract rules to detect such a pattern implicitly.

Approach For answering the research question, we will use the approach detailed in Section 3.2. The dataset, consisting of 18 projects from GitHub, is prepared as described in chapter Section 3.2.2. First, 20% of all files are moved to a separate holdout dataset. We split the remaining files into 90% train and 10% validation data. We use the training dataset to train Random Forest Classifiers, Gradient Boosting Classifiers, Support Vector Classifiers and LSTM-based neural networks. Due to the class imbalance, we will additionally train with an oversampled dataset (oversampling rate of 0.5) and an undersampled dataset (undersampling rate of 0.5). Since our datasets contain real-world code samples, we expect them to represent a real-world label distribution, and we, therefore, accept the potential effects of the data imbalance.

Our evaluation metrics are recall, precision and the combined F1 score. As described in Section 3.2.1, accuracy is not a meaningful metric due to the data imbalance. We choose the F1 score as a combined metric since we see recall and precision as equally important. This opinion is ensured by the F1 score that is the harmonic mean of recall and precision. Our reasoning to weight recall and precision equally is as follows: A high recall means a high detection rate of non-clean code. The costs of false negative predictions are potential costs in unclean code (maintainability, understandability). With high precision, a predicted non-clean code sample is likely to be a non-clean code sample, and the system reports less false positives. The immediate cost of false positive predictions (resulting in a lower precision) is the additional time of developers to identify it

as a false alarm. Additionally, the Cry Wolf Effect comes into play [61]. In our case, a developer who has seen a false alarm will take subsequent alarms less serious. A study has shown that a higher false alarm rate in the advisory warning system in cars results in a lower, subjective evaluation of the system [62]. A low subjective evaluation of developers would lead to a decrease in adoption rate, which consequently leads to more ,unfixed problematic code. Since both recall and precision are essential to the success of our models, we decide to use the equally weighted F1 score as a single-value evaluation metric. Additionally, to compensate for the cascading effect of a low precision due to the Cry Wolf Effect, we require a precision of 0.8 as a satisficing metric. On the other contrary, we define the recall as our optimising metric.

We evaluate all models for the three different problem types `RETURN_NONE` (RN), `CONDITION_COMPARISON_SIMPLE` (CCS) and `CONDITION_COMPARISON` (CC). We expect the CC type to be the most difficult to learn since it is the most complex rule. CCS and RN should be easier to learn due to less possible variations in the code structure.

Results We list all hyperparameter configurations and the corresponding training, validation and holdout performance for all problem types in the appendix in Tables A1 to A4. We report a detailed analysis in a separate finding for each model. Additionally, we describe our findings when comparing the models for the given classification tasks.

Finding 1: Support Vector Classifier *The support vector classifier in our configurations is not able to detect any problem type.*

Due to the quadratic time complexity during training described in Section 3.2.1, we were only able to test a few configurations. Since the scope of this paper is the comparison between different models, we did not have time to optimise the support vector classifier. We only tested configurations with a general subsampling and undersampling to reduce the input size and to achieve a manageable training time. Therefore, we do not have data for non-resampled datasets or oversampled datasets to draw reasonable conclusions.

We can only observe some effects: First, the class balancing increases the recall by up to 0.79 for RN, 0.5 for CCS and 0.55 for CC. Second, undersampling with a ratio of 0.1 results in an F1 performance of 0.0 for the tested RN and CCS problem types. Third, the F1 performance for the CC type is almost double the performance for the CCS type, i. e. 0.11 vs 0.06. The F1 performance for the CCS type is ten times the performance of the RN type, i.e. 0.062 vs 0.0066. The zero performance with an 0.1 undersampling

ratio may be an indicator of a strong sensitivity for class imbalance. Although we did not test this hypothesis any further, due to the time constraints detailed above, this is in line with the work of Tian et al. [63]. They observe a poor performance of support vector classifier on imbalanced datasets and evaluate further mitigation strategies.

In conclusion, the support vector classifier does not fulfil our acceptance criteria of a precision higher than 0.8. Especially the proneness to class imbalance decreases the useability for our problem domain with severely imbalanced datasets. Furthermore, training time limits our experiments. It is important to note that we did no further investigation into improving the training time or performance. It could be possible to increase classification and runtime performance by choosing a different kernel function such as a linear kernel function or by following strategies addressing this problem as proposes by Tian et al. [63]. We unsuccessfully try to combat class imbalance by our undersampling approach. Oversampling would increase the training time and is not feasible. We see further improvements as out of scope for this thesis since we want to compare different machine learning models and not optimising every single classifier.

Finding 2: Random Forests *Random forest classifier performs well and fulfils our satisficing condition on precision.* It achieves F1 scores of over 0.8 for the RN type and over 0.9 for the CCS and CC type. The best performing random forest classifier uses a 0.5 oversampling ration, 100 trees, type encoding and no additional class weighting.

Our observation indicates that the random forest classifier profits from more data, even with data duplication. We base this thesis on our observations with over- and undersampling the training data. All oversampled configurations perform better on the validation set than their not resampled counterparts. This performance improvement is especially large for the RN type, with an F1 score increase of 0.03-0.05. For the CC and CCS problem type, the performance growth is smaller, with only around 0.01 in F1 score. We explain the difference between the problem types in the difference in class frequency as we described in Section 3.2.1. For the RN problem type, only 0.21% of the samples are labelled positive, whereas 2.66% of the samples for CCS and 4.87% of the samples for CC are labelled positive. Oversampling the data with a ratio of 0.5 results in more duplicated samples from the minority class for the RN problem type than for CCS and CC. Table 4.1 contains detailed measures of the training set size after resampling and the performance metrics.

For undersampling, we observe a decrease in performance for all problem types. For the RN problem type, the F1 performance drops by up to 0.51, whereas the performance

	Validation F1 Score			Dataset		
	RN	CCS	CC	RN	CCS	CC
No resampling	0.7699	0.9208	0.9211	4,950,329	4,950,329	4,950,329
0.5 oversampling	0.8379	0.9329	0.929	7,409,925	7,228,141	7,064,008
1.0 oversampling	0.8358	0.9287	0.9273	9,879,900	9,637,522	9,418,678
0.1 undersampling	0.6857	0.9268	0.929	114,169	1,447,248	2,650,890
0.5 undersampling	0.2583	0.806	0.8632	31,137	394,704	722,970

Table 4.1: Dataset sizes for different resampling strategies with F1 performance on the validation set. The F1 performance for the specific resampling is for the *random forest classifier* with 100 trees, no class weighting and type encoding.

decrease for CCS and CC is lower with up to 0.12 and 0.07 for an undersampling ratio of 0.5. We explain this difference again with the differences in class frequency and the resulting changes in training size. With an undersampling ratio of just 0.1, fewer samples from the majority class get deleted, and the performance for RN only deteriorates by 0.08. The different performance decrease between the CCS and CC type can also result from the difference in class frequency. The undersampling with a ratio of 0.5 reduces the training size for CCS to 394,704 whereas for CC just to 722,970. With an undersampling ratio of just 0.1, the effect is smaller and only visible for the RN type (see Table 4.1).

Our experiments show that the random forest classifier is not very prone to class imbalance. Only an extreme class imbalance as for the RN type with a class distribution 0.21% vs 99.79% limits the classifier. We conclude this insight from two main observations. First, all performance metrics for the CCS and CC problem type are comparable for all configurations without over- or undersampling. If the model would be sensitive to class imbalance, the higher class imbalance of the CCS type (97.34% vs 2.66%) should be reflected in a worse performance than the CC type. We do not observe such performance differences. By contrast, the class frequency of positive samples for the RN type is by a factor twelve smaller than for the CCS type. The resulting severe class imbalance seems to affect the performance negatively. Second, the class weighting has no positive influence on the performance of the classifier. With balanced class weighting, the weighting for each class is adjusted inversely proportional to the class frequency. If the random forest classifier were sensitive to class imbalance, we would expect a performance improvement. Instead, we see a slight deterioration in F1 performance up to 0.04.

Type encoding increases the performance of the random forest classifier slightly for

the CCS and CC type of up to 0.02. For the RN problem type, we observe a similar improvement for oversampling with type encoding. For undersampling with type encoding, we observe a larger decrease of around 0.09 in F1 performance. We expected type encoding to have a more significant impact. The types should work as an additional feature the classifier can utilise. With the index-based encoding of the token values, 100,001 possible values can be encoded. The number of different token types is significantly smaller with only 61 different types (see the complete list of possible token types in Figure A1). Especially for function and method names, it would allow the model to treat two different function calls as one. However, the improvement is lower than expected. At least for the CCS type, the `NAME` type of a method call should be easier to learn on than the changing value of the method name.

Overall, the random forest classifier handles class imbalance fairly well and can profit from more data, even if it contains duplicates from oversampling. Undersampling decrease the performance of the classifier. Type encoding has a slightly positive impact on performance.

Finding 3: Gradient Boosting Classifier *The gradient boosting classifier performs well on all problem types and fulfils our satisficing metric.* We achieve an F1 score of over 0.9 for the RN and CC type and 0.84 for the CCS type. The best performing configuration is trained with 300 boosting stages, a learning rate of 0.2 and type encoding.

The gradient boosting classifier is sensitive to less training samples or samples with less variety. Two main observations allow this conclusion. First, we observe a decrease in performance for under- and oversampling (see Table 4.2). For undersampling the RN type with a ratio of 0.5, the F1 score drops from 0.8306 by 0.3761 (roughly 45%) and by 0.3759 (approximate 41%) to 0.4847 for oversampling with a ratio of 0.5. For the CCS type, the F1 score drops from 0.773 by approximately 0.08 to 0.6948 and 0.6836 for over- and undersampling. The smallest drop is for the CC type from 0.8567 by around 0.06 to 0.79 for over- and undersampling. With undersampling, we reduce the number of examples and with oversampling, we increase the minority class without increasing the variety of samples. Since the performance characteristics for over- and undersampling are comparable, we conclude that more samples with the same variety further decrease the model's performance. The decrease in F1 score is caused by a drop in precision, i.e. an increase of false positives. With less data by undersampling or less variety by oversampling, the classifier tends to be biased towards the positive class. Second, the

	Validation F1 Score			Dataset		
	RN	CCS	CC	RN	CCS	CC
No resampling	0.8306	0.773	0.8567	4,950,329	4,950,329	4,950,329
0.5 oversampling	0.4847	0.6948	0.7933	7,409,925	7,228,141	7,064,008
0.5 undersampling	0.4545	0.6836	0.7925	31,137	394,704	722,970

Table 4.2: Dataset sizes for different resampling strategies with F1 performance on the validation set. The F1 performance for the specific resampling is for the *gradient boosting classifier* with 200 stages, a learning rate of 0.2 and type encoding.

larger performance drop for the RN problem type and the smaller decrease for the CC type correlates with the number of training samples. A larger decrease in training data results in a larger drop in performance.

The gradient boosting classifier is insensitive towards class imbalance. One supporting point is the aforementioned performance decrease with over- or undersampling. Additionally, the performance of the RN problem type is comparable to the CC problem type despite their difference in class imbalance (0.21% class frequency for RN vs 4.87% for CC).

The gradient boosting classifier has problems learning the CCS type. Contradictory to our assumptions, the gradient boosting classifier learns the CC type better than the CCS type. The precision is lower by up to 0.05, whereas the recall is lower by up to 0.11 (without resampling and $\geq$ 200 boosting stages). We assume, the gradient boosting classifier has trouble learning the CCS subset and instead learns the more general superset of CC problems. To recap, the CCS problem type only labels a direct comparison in the if-condition as a problem, whereas the CC type considers nested boolean expressions. With the CCS subset, we restrict the classifier to only detect a direct comparison.

We analysed the classification of the holdout samples, especially the false-positive ones. If the classifier learns the structure of the CC superset, the classifier will label samples as positive that are only positive for the CC type. We randomly chose 20 false-positive samples and found 11 samples like in Listing 4.1. Those 11 samples contain a direct comparison nested inside boolean operators. Such nesting is only labelled as a CC problem, not as CCS problem. We think our restriction in problem types collide with the classifier's ability to detect the more general pattern of a comparison somewhere inside the if-condition.

```
    if self._supported_features & support_color:
    self._hs_color = (
<UNKNOWN> ( self.
DEDENT,NAME,NAME,OP,NAME,OP,NAME,OP,NEWLINE,INDENT,NAME,OP,NAME,OP,OP
    ,NL,NAME,OP,NAME,OP
##########################

if self.state is not none and self._sensor_type [ 0 ] == "battery" :

NEWLINE,NAME,NAME,OP,NAME,NAME,NAME,NAME,NAME,NAME,OP,NAME,OP,
    NUMBER,OP,OP,STRING,OP,NEWLINE,INDENT
##########################
if isinstance ( <UNKNOWN> , datetimearray ) and <UNKNOWN>.tz is not none :
    pytest.
INDENT,NAME,NAME,OP,NAME,OP,NAME,OP,NAME,NAME,OP,NAME,NAME,NAME,NAME
    ,OP,NEWLINE,INDENT,NAME,OP
```

Listing 4.1: False-positive samples for the best performing *gradient boosting classifier* with 300 boosting stages, a learning rate of 0.2, type encoding and without resampling for the CCS type. We removed spaces between tokens if necessary, but leave the indentation as in the sample.

Subsampling the train data for the individual base learner during training has no impact. The performance for all subsampling ratios are identical, so the individual base learners are trained with the same data as before.

The gradient boosting classifier is robust to overfitting the training data. We observe a comparable train and test performance for all non-resampled configurations. Only with under- or oversampling, the classifier seems to overfit the train data, since the test performance is worse than the train performance. Another indicator for the robustness is the performance increase when using more boosting stages. Increasing the boosting stages increases the recall and precision for all problem types. Our best model with 300 boosting stages and a 0.2 learning rate can probably be more fine-tuned by increasing the boosting stages and adapting the learning rate. Again, being restraint by the increase in training time, we did not fine-tune the classifier further.

In conclusion, the gradient boosting classifier is robust to overfitting and somewhat improves its performance with more boosting steps. We can also observe robustness for class imbalance and on the other hand sensitivity towards less training data. Subsampling has no effect, and we observe an unexpected high difficulty of learning the CCS type for this classifier.

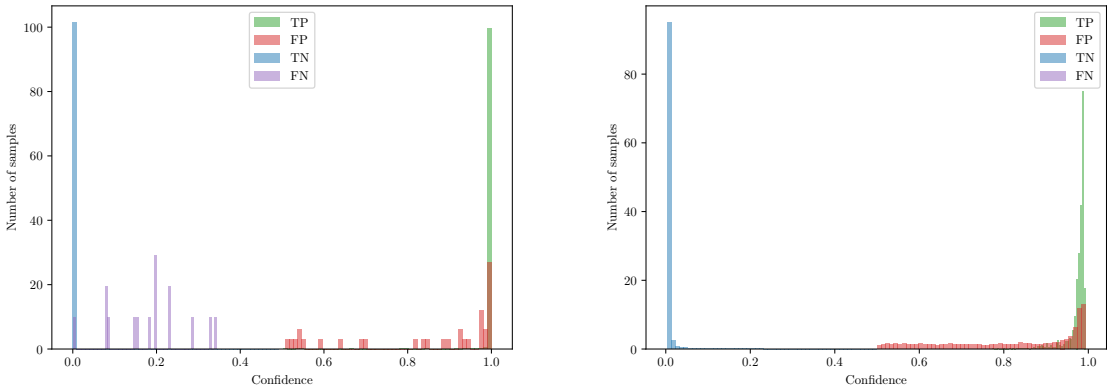
Finding 4: LSTM-Based Neural Network *Our implementation of an LSTM-based neural network performs well and fulfils our satisficing condition.* We achieve an F1 score of over 0.97 for all problem types without resampling, 10 LSTM-cells, an embedding size of 32 and training with two epochs and 256 batch size.

Since we have a remarkably high performance, we do not vary all parameters. Instead, we use a 0.5 under- and oversampling as for other models, vary the batch size and number of epochs and try reducing the size of the embedding layer since this layer contains the majority of the trainable parameter (see Listing 3.7).

Oversampling does not have a measurable impact on our LSTM-based neural network. The F1 performance with a 0.5 oversampling is not significantly different from the performance without oversampling. We observe overfitting on the training data with a near-perfect recall and precision. The test performance remains higher than 0.97 F1 score. Although the model overfits to the train data, it still generalises well enough with comparable performance to the models trained on the original class distribution.

The LSTM-based neural network is sensitive to fewer training data. We draw this conclusion based on the performance deterioration for undersampled training sets (see Table 4.3). For the RN type, undersampling results in a significant drop in performance from 0.987 by roughly 36%. For CCS and CC, the performance decrease is smaller at less than 3%. As described before, the train set of the RN type is significantly smaller after undersampling than for the CCS and CC type, due to the lower class frequency of the minority class of only 0.21% before undersampling. Therefore, more samples from the majority class are deleted during undersampling and the variety decreases. In combination with many trainable parameters in our model, the model fails to fit the training data correctly. Responsible for the performance drop is a decrease in precision, whereas the recall remains high. In other words, the false-positive rate increases by factor 100 to 0.3%. By removing negative samples and variety in training, we think it becomes more challenging for the model to detect negative samples correctly. Furthermore, the recall remains high, which is a sign that the model tends to classify more records into the positive class. In Figure 4.1b, we see this trend towards the positive class but with increased inconfidence. Without undersampling, the model is very confident about the true negatives and false positives but is inconfident with some samples.

Our model configuration has too many trainable parameters for the problem complexity. We base this assumption on two observations. First, we halved the size of the embedding layer to 16 and consequently reducing the 3,201,795 trainable parameters of the model to roughly half the size. If the model needed all parameters to fit the train-



(a) Confidence Histogram after training without resampling.

(b) Confidence Histogram after training with a 0.5 undersampling ratio.

Figure 4.1: Confidence Histogram for predicting the holdout set for an *LSTM* trained with an embedding size of 32, three epochs, a batch size of 256 and 10 LSTM-cells.

	Validation F1 Score			Dataset		
	RN	CCS	CC	RN	CCS	CC
No resampling	0.987	0.9808	0.9782	4,950,329	4,950,329	4,950,329
0.5 oversampling	0.9859	0.9781	0.9745	7,409,925	7,228,141	7,064,008
0.5 undersampling	0.6294	0.9564	0.9561	31,137	394,704	722,970

Table 4.3: Dataset sizes for different resampling strategies with F1 performance on the validation set. The F1 performance for the specific resampling is for the *LSTM* with an embedding size of 32, 10 LSTM-cells, a training on three epochs with a batch size of 256.

ing data accurately, we would expect a performance drop. Instead, we do not observe any significant change in performance. Due to this observation, we assume a further reduction of the embedding size may be possible without performance loss. However, since this configuration trains in a reasonable time of under an hour without a GPU and has an acceptable performance on the holdout set, we do not investigate this too powerful model further. Second, a variation of the epochs and batch size for the training yields no significant difference. We think, the problem detection for our problem types is too simple for the model and it quickly reaches its optimum. Further training does not increase the model's performance further.

To sum up, we achieve near-perfect performance with our neural network model. We acknowledge, that our model has too many trainable parameters that we do not need for the same performance as shown with the reduction of the embedding layer size. The model does not benefit from resampling. Oversampling has no impact on the test performance, and undersampling impairs the training of the model's weight since it reduces the overall number of samples.

Finding 5 We found a weakness in the way we labelled samples with the rule checkers. We decided to label a sample as a rule violation if all tokens of the problematic line are present. With the sliding window approach, it is possible to have all tokens for the detection in the sample, but missing the first or last newline or indentation tokens. The sample would therefore be labelled as clean code, although the problematic pattern is there. In the analysis of the false-positives for the gradient boosting classifier, we found samples for both cases in the false-positives, as seen in Listing 4.2. In the first sample, not all preceding `DEDENT` types are within the window, whereas the trailing `NEWLINE` token is missing in the second sample.

To improve this weakness, we suggest to strip out all `DEDENT` and possibly `NEWLINE` tokens from the data before applying the sliding window. Due to time constraints, we are not able to retrain all models with an improved approach. However, the problematic line will occur in at least another sample, if the line is shorter than 16 tokens. It will most likely not impact the variety of problem patterns, but it will slightly reduce our performance score due to false-positives.

Summary Table 4.4 shows the performance score of the best performing model of each classifier for all problem types. The SVC performs poorly and does not meet our mandatory satisficing condition of precision higher than 0.8. Therefore, we will not evaluate it further. On the other hand, the LSTM-based neural network model outperforms all other models on this classification task for all problem types.

Both random forest and gradient boosting classifiers, fulfil the satisficing condition for all problem types. Interestingly, they supplement each other: For the RN problem type, the gradient boosting classifier performs better with an F1 score of 0.9328, whereas the random forest classifier surpasses the gradient boosting classifier for the CCS and CC problem type with 0.9352 and 0.9351 F1. As discussed, we explain the former result with the better insensitivity to a class imbalance of the gradient boosting classifier. In contrast, the latter seems to be harder to learn for the gradient boosting classifier. Especially the difficulties for the CCS type are unexpected and do not affect the random

```
if 'add_host' in <UNKNOWN> :
<UNKNOWN>
                        <UNKNOWN> = <UNKNOWN> . get ( 'add_host' ,
DEDENT,DEDENT,DEDENT,NAME,STRING,NAME,
NAME,OP,NEWLINE,COMMENT,NL,INDENT,NAME,
OP,NAME,OP,NAME,OP,STRING,OP
##########################
if vhost.ssl:
points += 3

if points > <UNKNOWN> :
NL,DEDENT,NAME,NAME,OP,NAME,OP,NEWLINE,
INDENT,NAME,OP,NUMBER,NEWLINE,NL,DEDENT,
NAME,NAME,OP,NAME,OP
```

Listing 4.2: False-positive samples of the best *gradient boosting classifier* with 300 boosting stages, a learning rate of 0.2, type encoding and without resampling for the CCS type. In the first sample, not all preceding `DEDENT` tokens are in this window. For the second example, the trailing `NEWLINE` token is missing in the window. Without this encoding issue, both samples would have been most likely detected correctly.

forest classifier or LSTM.

4.1.3 RQ3: Do Machine Learning-Based Models Cover a Larger Variety of Cases Than Rule-Based Checker?

Motivation For the previous research question, we trained models on a dataset. We created the dataset by downloading open-source repositories and applying our analysis plugins from Section 3.1.2 to label samples as clean or non-clean code. With this automated approach, it is simple to generate a dataset and to scale the dataset to any size needed for training. Such algorithms (checkers) that detect non-clean code are the prerequisite for the dataset and successful training of machine learning models. This research question wants to put the machine learning approach into a scenario, in which this prerequisite is not met, and we lack a checker to generate a labelled dataset. This scenario transfers to clean code rules with a high level of complexity (as defined in Section 2.4). For these rules, we may not be able to analyse the source code or another representation with a checker to generate a representative dataset. We could (1) hand-label samples and train a machine learning model on those samples. Alternatively, we may (2) only find a deterministic checker that detects a subset of the rule violations. We hope that a model trained on either one of those datasets generalises well to all

		RN	CCS	CC
Random Forest				
	F1	0.8486	0.9352	0.9351
	Recall	0.7404	0.8936	0.8976
	Precision	0.9939	0.9809	0.9759
SVC				
	F1	0.0066	0.0629	0.1124
	Recall	0.0134	0.501	0.5497
	Precision	0.0044	0.0336	0.0626
Gradient Boosting Classifier				
	F1	0.9328	0.8489	0.9088
	Recall	0.9005	0.7953	0.8766
	Precision	0.9675	0.9102	0.9434
LSTM				
	F1	0.9933	0.9812	0.9783
	Recall	0.9933	0.9808	0.9868
	Precision	0.9933	0.9815	0.97

Table 4.4: Best performing classifier for RQ2 based on F1 score per problem type on the holdout set. For the support vector classifier (SVC), the validation performance of the configuration with available data is used.

patterns of the rule violation. Both scenarios boil down to a lack of samples for all or many structural variations of the rule violation and a discrepancy of the dataset and the real world with more variety. We will evaluate both scenarios for our models from the previous experiment in this research question.

Approach To answer this research question, we use all models except the support vector classifier from the previous research question for all three problem types: We use the models trained on the RN problem type to simulate scenarios (1). First, we manipulate the holdout dataset as described in Section 3.2.3. The code still contains returns of `None`, although it is not explicitly written as two consecutive code tokens. We want to explore if a model generalised well enough to detect unseen variations of the problem type. With the manipulation, we ensure that all samples contain the unseen variations and our performance metrics only refer to those samples.

Second, we simulate scenario (2) of having trained on a subset of all variations with the CCS models. The CCS model was trained on a subset of the CC problem patterns. We manipulate all occurrences of the CC problem type to contain only patterns that are

not in the subset for the CCS training. It is important to note that the class frequency of positive samples changes from the trained 2.51% to the 5.1% positive samples for the manipulated CC type in the holdout set.

Last, we evaluate the models trained on the CC type with our manipulated code to provide a baseline for the CCS trained models. We expect the CC model to show similar performance than in the previous experiments since it was trained on the problem structure of the manipulated CC dataset.

We expect the best performing CC model from the previous experiments to outperform all CCS models and to show similar performance like in the previous research question.

Results We evaluate the models' performance for both scenarios: The RN problem type simulates scenario (1) whereas the CCS problem type represents scenario (2). We list all evaluation results in the appendix in Tables A5 to A7.

Finding 1 *The detection of the manipulated RN problem type works poorly with all models.* The random forest classifier has an F1 score of less than 0.1, the gradient boosting classifier of 0.24 and the LSTM-based neural network of 0.39. The precision of all models is below 0.26 and would therefore not comply with the satisficing condition from the previous experiment of 0.8. Our simulation for scenario (1) fails.

We think the model did not learn the problem structure during training. Since all models perform significantly worse than in the previous experiment, it is feasible to search for the reason in the problem type. Our reasoning goes as follows: The problem type appears in the training set only in one variation: A `return` keyword is followed by a subsequent `None` value. The surrounding tokens varied, but only the subsequent occurrence of these two tokens had to be learnt to perform to detect the problem type. For the previous experiment, this behaviour resulted in the best performance. To perform well on this experiment, the model should have learned a more sophisticated approach of a `None` value after the `return` keyword before the end of the line. Since this was not part of the training data, it is obvious that the model learnt the simple subsequent detection of this pattern, since it yields the best performance in the previous experiment.

Finding 2 *The classification of the modified CC samples with the CCS models yields better results than for the RN type.* For the random forest classifier, the performance on the modified CC dataset is at 0.72 F1 score. The best gradient boosting classifier in this experiment achieves 0.83, and the best LSTM-based neural network reaches 0.6568 F1

score. Compared to the performance for the RN types, the performance values are at least doubled, so the model has learnt the pattern from the subset better. We attribute this to the wider variety in problem patterns in the source code: Since the CCS checker looks for comparisons in the condition, it will label different comparison operators such ==, < or > as rule violations, whereas the only variation for the RN type is `return None` as subsequent tokens.

Finding 3 *The more fine-tuning during training, the less the model generalises for the modified problems.*

All models on the modified problem types behave contrary to the unmodified problem types in the previous experiments: the hyperparameter and resampling combinations with a worse performance in the previous experiment score better in this evaluation. By contrast, the best model configurations from the previous experiment are among the worst-performing models for this evaluation. In Figure 4.2, we plotted the F1 performance of the baseline, manipulated RN and CCS. For each classifier, we sorted the models with different resampling and hyperparameter configurations descending on the baseline F1 score. The baseline performance is the performance of the CC model from the previous experiment on the manipulated CC data. It correlates with the performance of the CC model on the original holdout data in the previous experiment. The overall picture shows that the models with the best baseline performance have a low performance on the manipulated datasets. With decreasing baseline performance, the performance of the manipulated CCS increases for the random forest and gradient boosting classifier. The manipulated RN type follows this pattern for the random forest classifier and the LSTM.

We observe this behaviour on different hyperparameters and resampling combinations in the previous experiment. It applies to both problem types. For the random forest classifier, undersampling the training set lead inevitably to worse performance in the previous experiment than without undersampling. In this experiment, the random forest classifier, which was trained on the undersampled dataset leads to the overall best scores. For gradient boosting classifier, the models without resampling scored best in the previous experiment and are performing worst in this experiment. Additionally, increases in boosting stages that lead to an improvement in the previous experiment now lead to a deterioration in performance. The LSTM-based neural network results draw a similar picture. The undersampling harmed the performance, whereas, in this experiment, a 0.5 undersampling achieves the only F1 score over zero for the RN type

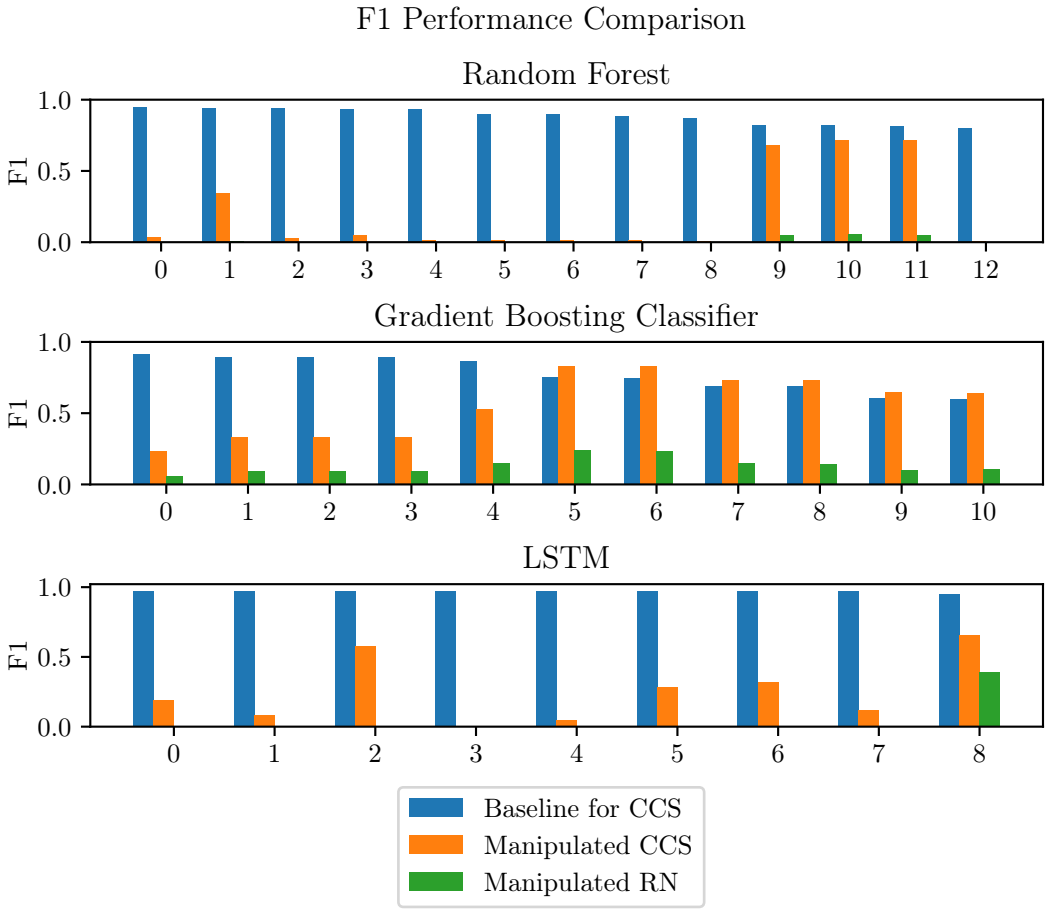


Figure 4.2: F1 performance of the models, sorted by the baseline F1 score. The baseline performance correlates with the performance of the CC type in the previous experiment. With a worse performance in the previous experiment, the performance for the manipulated types increases.

and the best for the manipulated CC type.

We would explain this observation separate for the RN and CCS type: For the RN type, the models probably only learnt the direct token sequence (as described in Finding 1). The better it learnt this sequence, the better its performance on the previous experiment. For this experiment, learning the direct token sequence would result in a low performance. Instead, the model would have to learn the structure of the problem. We analysed the false-positive samples of the random forest classifier that performed best in this experiment and its false-positives in the previous experiment. Contrary to our assumptions, it failed in the previous experiment since it used the `return` token as an indicator to label samples positive in 19 out of 20 samples. The `None` token was

not important, since it was only in 4 out of the 20 samples. This resulted in a low precision of 0.158. We see the same result with the manipulated RN data with 19 out of 20 samples containing a `return` token and 3 out of 20 samples containing a `None` token. For the best gradient boosting classifier in this experiment, 16 out of 20 false-positive samples contain a `return` token and 14 out of 20 for the best LSTM in this experiment.

The CCS problem type, simulating scenario (2), offered more variety to learn from, as discussed. If the model learnt a general problem structure from this variety, fine-tuning the hyperparameters for the previous experiment would restrict this generality to only CCS patterns. We assumed in Section 4.1.2 that the gradient boosting classifier get restricted by the CCS type and is more likely to learn the CC pattern. In this experiment, the best gradient boosting classifier configuration surpasses all other models. Additionally, it surpasses the baseline performance measure of the model trained on the unmodified CC type that we assumed to perform better. This solidifies our assumptions about the learning of the CCS type. We assume this explanation applies to other models as well.

Improvements As identified in the findings, we see the root cause for the bad performance in the learning of specific code patterns and not the general structure of a problem type. We identified the lack of variety as the main reason, manifesting as the restriction of the learning on the CCS type. For the RN type, the general structure on a token level would be something like a `None` value somewhere after a `return` but before a line break. A slightly more advanced structure could be extracted from the CCS type: A comparison inside a condition (starting with an `if` statement and closing with the colon) should be marked. With the chosen approach, the model did not train on the general problem structure of the clean code violation but explicit code patterns.

We see several ways to improve: First, we did not provide the model with the full variety of problem patterns. Therefore, we suggest hand-label or hand-generated patterns with explicit variations. In the motivation for this research question, we state that it may not be practically possible to collect many samples with the full variety of problem patterns for an adequate training corpus. Instead, it may be possible to collect samples with enough variety as an incentive to learn the pattern. For the improvement, we need a base collection of samples. This base collection can be obtained by an algorithm that can only detect a subset of the problem patterns (like in our approach with the CCS type) or a similar enough code pattern (like we tried with the RN type). On top of that, we explicitly create samples with more variety by either hand-labelling existing

code or writing new code. With the hand-crafted samples, we fine-tune the model to increase the variety of code patterns that the model can generalise on. Simultaneously, we loosen the indirect restriction of the dataset covering only a subset of the data. By combining a scaleable, automatically generated dataset with hand-labelled samples, we may be able to improve the models' performance on the samples that we could not label automatically. Additionally, we have to use some hand-labelled samples for more variety in the validation set to tune the hyperparameters to more possible problem variations in real datasets.

Second, we may compensate for less variety in training samples with a different encoding. Using the encoding to focus on the code structure may require fewer variations to extract the problematic code structure. We use an indexed-based encoding of the source code's token stream. With this approach, we do not exploit the code structure in our encoding. Other machine learning tasks on code profit from other encoding types that exploit more structure from the code. Alon et al. introduced code2vec to represent code as paths in the AST [22], capturing the structure of the code. Similarly, Zhang et al. split the AST into smaller statement trees and apply a recurrent neural network approach to generate a code representation [64]. Since we identified the lack of learning the problem structure from the samples, an encoding approach that explicitly takes advantage of the structure of code may improve the performance of this experiment. It may even be possible to train the model on a subset of possible problem variations like we tried with the manipulated CCS type without the restricting effect, since the problem structure may be inferred from the subset alone. Additionally, this may be combined with the fine-tuning on hand-labelled samples to increase the performance to a practical level. The satisficing condition of precision higher than 0.8, we defined in the previous experiment, would also apply to this experiment. If the precision were too low, the Cry Wolf Effect would come into effect, and the practical use would therefore be small.

Summary In summary, the detection of the modified RN type performs poorly with all models and configurations. We see the reason in the low variety of different problem patterns. The experiment with the RN type should simulate the scenario in which we could only obtain a dataset with hand-labelled samples that does not cover all possible patterns. In hindsight, having only one sample variation for the problem pattern simulates the worst possible scenario. Based on the analysis of the results, we admit that a hand-labelled sample set would probably contain a greater variety of samples.

With more variety, the result would most likely be more comparable to the performance of the CCS type with more variation. If we define the subset of the CCS type as a hand-labelled set, we can interpret the results for this problem type in the context of scenario (1).

The performance on the modified CC type is more promising than for the RN type. We explain this by more variety in the training samples, although the models seem to be restricted to this subset during training (especially the gradient boosting classifier). If we interpret the performance with the model trained on the CCS type with the evaluation on the modified CC type data as scenario (1), the variations of our hand-labelled samples will restrict the models' performance. Furthermore, it would lead to a misleading parameter tuning, since our validation set would not contain all real-world variations.

For improvement, we suggest to hand-craft additional samples with a different variation of the problem pattern. This can be used for increasing the variety of the training set with fine-tuning or to better approximate the real world in the validation set to tune the models' parameter towards real-life datasets. Additionally, encoding the data by utilising the source code structure, less variety may be necessary to learn the variations. If this is the case, it would have positive practical implications, since it would require less manual code labelling or creating.

5 Conclusion and Outlook

Automatically detecting violations of the clean code rules can improve the understandability and readability of source code.

Our first contribution was the proposal of a taxonomy for clean code rules based on the assumed level of complexity for automated checkers. We used this taxonomy to classify common clean code rules and identify areas of interest for our research questions.

Next, we designed and implemented a tool that can run multiple automated checkers on source code (CCAP) with a focus on extensibility, useability and integration. In RQ1, we compared the CCAP with existing tools for code checking. We found a high value for users and extension developers in its useability and extensibility, whereas it lacks the implementation for integrations in different workflows or IDEs unlike compared tools. We see the CCAP as a useful addition to Sonarqube to supplement its more complex extensibility.

In our taxonomy, we identified clean code rules that, we assumed, could not be checked without the help of machine learning algorithms. In RQ2, we trained and compared multiple machine learning models to detect code patterns that violate clean code rules. A support vector classifier was not able to detect the pattern with acceptable performance. The LSTM-based neural network outperformed all other models on the task, but it is sensitive towards less training data. A random forest classifier fulfiled our acceptance criteria. It can handle a moderate class imbalance in the training data and profits from oversampling the training samples. The gradient boosting classifier also fulfiled our acceptance criteria and performed better than the random forest classifier on two out of three problematic code patterns. Additionally, it is insensitive towards class imbalance which is a useful advantage for the imbalanced classification of problematic code patterns.

Last, we evaluated in RQ3 if our models can cover a larger variety of cases than in the collected training samples. On unseen variations of code patterns, all models performed below an acceptable level. Our analysis indicates that our models did learn detecting the pattern but not the structure behind it.

For future work, we suggest analysing a different encoding approach that incorporates the structural information of the source code and to increase the number of variations in the training and validation dataset. Furthermore, evaluating more machine learning models like a more recent transformer model might be beneficial. We can even think of changing the premise of the task from classification to anomaly detection and evaluate if those techniques could be superior.

For the CCAP, future work can be done in its integrations into different IDEs and additional analysis/output plugins. Additional functionality such as different warning levels and the ability to ignore a problem can further increase the useability and usefulness for developers.

Bibliography

[1] Statista. Software market revenue in the World 2016-2021. https://www.statista.com/forecasts/963597/software-revenue-in-the-world.

[2] Katie Costello. Gartner Says Global IT Spending to Grow 0.6% in 2019. https://www.gartner.com/en/newsroom/press-releases/2019-10-07-gartner-says-global-it-spending-to-grow-06-in-2019, July 2019.

[3] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[4] ISO Central Secretary. ISO/IEC 25010:2011: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard ISO/IEC 25000:2014, International Organization for Standardization, Geneva, CH, 2011.

[5] S.W.L. Yip and T. Lam. A software maintenance survey. In *Proceedings of 1st Asia-Pacific Software Engineering Conference*, pages 70–79, Tokyo, Japan, 1994. IEEE Comput. Soc. Press.

[6] Galorath. Accurately Estimate Your Software Maintenance Costs, January 2019.

[7] Stripe and Harris Poll. The Developer Coefficient - Software engineering efficiency and its $3 trillion impact on global GDP. https://stripe.com/de/reports/developer-coefficient-2018, September 2018.

[8] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[9] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Series in Agile Software Development. Prentice Hall, Upper Saddle River, NJ, 2002.

[10] Wang Haoyu and Zhou Haili. Basic Design Principles in Software Engineering. In *2012 Fourth International Conference on Computational and Information Sciences*, pages 1251–1254, Chongqing, China, August 2012. IEEE.

[11] Lutz Buch and Artur Andrzejak. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–104, Hangzhou, China, February 2019. IEEE.

[12] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 354–365, Lake Buena Vista, FL, USA, 2018. ACM Press.

[13] K.J. Lieberherr and I.M. Holland. Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48, September 1989.

[14] Go at Google: Language Design in the Service of Software Engineering. `https://talks.golang.org/2012/splash.article`.

[15] Andrew Gerrand. Error handling and Go. `https://blog.golang.org/error-handling-and-go`, December 2011.

[16] Tony Hoare. Null References: The Billion Dollar Mistake, March 2009.

[17] Null Safety - Kotlin Programming Language. `https://kotlinlang.org/docs/reference/null-safety.html`.

[18] Herbert Prahofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application. *IEEE Transactions on Industrial Informatics*, 13(1):37–47, February 2017.

[19] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer International Publishing : Imprint: Springer, Cham, 2nd ed. 2017 edition, 2017.

[20] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[21] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4):1–37, September 2018.

[22] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning Distributed Representations of Code. *arXiv:1803.09473 [cs, stat]*, October 2018.

[23] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

[24] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[25] ISO/TC 22/SC 32. ISO 26262:2018 Road vehicles — Functional safety. Technical report, International Organization for Standardization, December 2018.

[26] ISO/TC 210. IEC 62304:2006 Medical device software — Software life cycle processes. Standard, International Organization for Standardization, Geneva, CH, May 2006.

[27] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 352–356, Chengdu, China, 2010. IEEE.

[28] Maurice Howard Halstead et al. *Elements of Software Science*, volume 7. Elsevier New York, 1977.

[29] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, August 1994.

[30] Dag I K Sjøberg, Dag Sjoberg, Bente Anda, and Audris Mockus. Questioning Software Maintenance Metrics: A Comparative Case Study. page 4.

[31] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, Lake Buena Vista, FL, USA, October 2009. IEEE.

[32] TC 65/SC 65A. IEC 61508:2010 Parts 1-7. Standard, International Electrotechnical Commission, April 2010.

[33] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[34] Aurelien Delaitre, Bertrand Stivalet, Elizabeth Fong, and Vadim Okun. Evaluating Bug Finders – Test and Measurement of Static Code Analyzers. In *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pages 14–20, Florence, Italy, May 2015. IEEE.

[35] Dave Wichers and Eitan Worcel. Source Code Analysis Tools | OWASP. `https://owasp.org/www-community/Source_Code_Analysis_Tools`.

[36] Pylint - code analysis for Python | www.pylint.org. `https://pylint.org/`.

[37] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. PEP 8: Style guide for Python code, 2013.

[38] How to Write a Checker — Pylint 2.6.1-dev1 documentation. `https://pylint.pycqa.org/en/latest/how_tos/custom_checkers.html`.

[39] PMD Source Code Analyzer. `https://pmd.github.io/`.

[40] Codacy docs. `https://docs.codacy.com/`.

[41] Documentation Index | PMD Source Code Analyzer. `https://pmd.github.io/pmd-6.29.0/`.

[42] Codacy | The fastest static analysis tool from setup to first analysis. `https://www.codacy.com/`.

[43] Code Quality and Security | SonarQube. `https://www.sonarqube.org/`.

[44] Python static code analysis | SonarQube. `https://www.sonarqube.org/features/multi-languages/python/`.

[45] G Ann Campbell. Cognitive Complexity-A new way of measuring understandability. *SonarSource SA*, 2018.

[46] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*, pages 1–12, Limerick, Ireland, 2016. ACM Press.

[47] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, June 2016.

[48] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, April 2019.

[49] GitHub Inc. The State of the Octoverse. `https://octoverse.github.com/`, 2019.

[50] TIOBE - The Software Quality Company. TIOBE Index for October 2020. `https://www.tiobe.com/tiobe-index/`, 2020.

[51] Guido Van Rossum, Jukka Lehtosalo, and Lukasz Langa. PEP 484–type hints. `https://www.python.org/dev/peps/pep-0484/`, 2014.

[52] Jukka Lehtosalo et al. Mypy-Optional Static Typing for Python. http://mypy-lang.org.

[53] Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747, 2016.

[54] John Pendenque. Code refactoring + Cleaning + Improvements · Zadigo/ecommerce_template@db2655c. `https://github.com/Zadigo/ecommerce_template/commit/db2655c`, August 2020.

[55] Grzegorz Baranski. Cleaned code, muuch improvement in performance · gbaranski/houseflow@7d06d9f. `https://github.com/gbaranski/houseflow/commit/7d06d9f`, September 2020.

[56] Pavel Timofeev. [lab4] some clean-code improvement · ptimofeyeff/distributed-computing@ff5e132. `https://github.com/ptimofeyeff/distributed-computing/commit/ff5e132`, August 2020.

[57] Abdiansah Abdiansah and Retantyo Wardoyo. Time Complexity Analysis of Support Vector Machines (SVM) in LibSVM. *International Journal of Computer Applications*, 128(3):28–34, October 2015.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[59] François Chollet et al. Keras. 2015.

[60] ISO/TC 159/SC 4. ISO 9241-11:2018: Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts. Standard ISO/IEC 25000:2014, International Organization for Standardization, Geneva, CH, March 2018.

[61] Shlomo Breznitz. *Cry Wolf: The Psychology of False Alarms*. Lawrence Erlbaum Associates, Hillsdale, N.J, 1984.

[62] Frederik Naujoks, Andrea Kiesel, and Alexandra Neukum. Cooperative warning systems: The impact of false and unnecessary alarms on drivers' compliance. *Accident Analysis & Prevention*, 97:162–175, December 2016.

[63] Jiang Tian, Hong Gu, and Wenqi Liu. Imbalanced classification using support vector machine ensemble. *Neural Computing and Applications*, 20(2):203–209, March 2011.

[64] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, Montreal, QC, Canada, May 2019. IEEE.

Appendices

RQ2: Model Performances

Parameter						RN			CCS			CC		
over-samp.	under-samp.	#trees	class weight	encode type		Train	Valid.	Hold.	Train	Valid.	Hold.	Train	Valid.	Hold.
0.5	-	100.0	None	True										
					F1	1.0	0.8379	0.8486	1.0	0.9329	0.9352	1.0	0.929	0.9351
					Rec	1.0	0.7221	0.7404	1.0	0.8898	0.8936	1.0	0.8888	0.8976
					Prec	1.0	0.9978	0.9939	1.0	0.9803	0.9809	1.0	0.9731	0.9759
1.0	-	100.0	None	True										
					F1	1.0	0.8358	0.8367	1.0	0.9287	0.9322	1.0	0.9273	0.9324
					Rec	1.0	0.719	0.7225	1.0	0.8824	0.8876	1.0	0.8853	0.8925
					Prec	1.0	0.9978	0.9937	1.0	0.9801	0.9815	1.0	0.9736	0.9761
0.5	-	100.0	balanced	True										
					F1	1.0	0.8352	0.8385	1.0	0.9287	0.9321	1.0	0.9267	0.9329
					Rec	1.0	0.7182	0.7249	1.0	0.8826	0.8882	1.0	0.885	0.893
					Prec	1.0	0.9978	0.9942	1.0	0.9799	0.9806	1.0	0.9727	0.9765
0.5	-	100.0	None	False										
					F1	1.0	0.8287	0.8519	1.0	0.9191	0.9217	1.0	0.921	0.9248
					Rec	1.0	0.708	0.7453	1.0	0.869	0.8723	1.0	0.8736	0.8785
					Prec	1.0	0.9989	0.9939	1.0	0.9752	0.977	1.0	0.9739	0.9764
1.0	-	100.0	None	False										
					F1	1.0	0.8162	0.8268	1.0	0.9189	0.9186	1.0	0.9161	0.9214
					Rec	1.0	0.69	0.7084	1.0	0.8692	0.8674	1.0	0.8656	0.872
					Prec	1.0	0.9989	0.9926	1.0	0.9747	0.9763	1.0	0.9729	0.9767
-	-	100.0	None	False										
					F1	0.9999	0.7814	0.7987	0.9998	0.9057	0.9106	0.9998	0.9116	0.9173
					Rec	0.9998	0.6413	0.6658	0.9999	0.8406	0.8489	0.9998	0.8495	0.8571
					Prec	1.0	1.0	0.9979	0.9998	0.9818	0.982	0.9998	0.9835	0.9866
-	-	100.0	None	True										
					F1	0.9999	0.7699	0.7666	0.9998	0.9208	0.9243	0.9998	0.9211	0.9269
					Rec	0.9998	0.6264	0.6226	0.9999	0.8637	0.8694	0.9998	0.8675	0.8759
					Prec	1.0	0.9987	0.9972	0.9998	0.9859	0.9865	0.9998	0.9817	0.9841
-	-	100.0	balanced	True										
					F1	0.9999	0.7506	0.7503	0.9998	0.9064	0.9087	0.9998	0.9071	0.9121
					Rec	1.0	0.6013	0.6018	1.0	0.8385	0.8404	1.0	0.8414	0.8484
					Prec	0.9997	0.9987	0.9959	0.9996	0.9863	0.9891	0.9997	0.9839	0.9863
-	-	100.0	balanced	False										
					F1	0.9999	0.7475	0.7405	0.9998	0.8894	0.8924	0.9998	0.8913	0.898
					Rec	1.0	0.5973	0.5888	1.0	0.8112	0.8149	1.0	0.8136	0.8227
					Prec	0.9997	0.9987	0.9976	0.9996	0.9843	0.9863	0.9997	0.9853	0.9885
-	0.1	100.0	None	True										
					F1	1.0	0.6857	0.654	0.9999	0.9268	0.928	0.9999	0.929	0.9345
					Rec	1.0	0.9027	0.9275	1.0	0.92	0.9243	0.9999	0.8981	0.9064
					Prec	1.0	0.5529	0.5051	0.9999	0.9336	0.9317	0.9999	0.9621	0.9644
-	0.5	100.0	None	False										
					F1	1.0	0.3469	0.3024	1.0	0.8087	0.8157	1.0	0.8581	0.8624
					Rec	1.0	0.9364	0.9712	1.0	0.966	0.9684	1.0	0.9551	0.9566
					Prec	1.0	0.2128	0.1791	1.0	0.6954	0.7046	1.0	0.779	0.785
-	0.5	100.0	balanced	True										
					F1	1.0	0.2709	0.2357	1.0	0.8111	0.8181	1.0	0.8671	0.871
					Rec	1.0	0.9513	0.9775	1.0	0.967	0.9672	1.0	0.9537	0.9562
					Prec	1.0	0.158	0.134	1.0	0.6985	0.7088	0.9999	0.795	0.7998
-	0.5	100.0	None	True										
					F1	1.0	0.2583	0.2227	1.0	0.806	0.8151	1.0	0.8632	0.8693
					Rec	1.0	0.9568	0.9782	1.0	0.9692	0.9709	1.0	0.9562	0.9602
					Prec	1.0	0.1493	0.1257	1.0	0.6899	0.7024	1.0	0.7867	0.7941

Table A1: F1, recall and precision scores for all parameter combinations of the *random forest classifier* on the training, validation and holdout set.

Parameter							RN			CCS			CC		
over-samp.	under-samp.	stages	learn-rate	sub-sample	encode type		Train	Valid.	Hold.	Train	Valid.	Hold.	Train	Valid.	Hold.
-	-	300	0.2	1.0	True										
						F1	0.9364	0.9196	0.9328	0.8477	0.8419	0.8489	0.9087	0.9008	0.9088
						Rec	0.9015	0.8666	0.9005	0.7926	0.7902	0.7953	0.8767	0.8673	0.8766
						Prec	0.9742	0.9796	0.9675	0.9111	0.9009	0.9102	0.9431	0.937	0.9434
-	-	200	0.2	1.0	True										
						F1	0.8554	0.8306	0.8499	0.7801	0.773	0.7837	0.8644	0.8567	0.8667
						Rec	0.7695	0.7276	0.765	0.6934	0.6886	0.6992	0.8057	0.7951	0.8081
						Prec	0.9629	0.9676	0.956	0.8916	0.881	0.8915	0.9324	0.9286	0.9344
-	-	200	0.2	0.4	True										
						F1	0.8554	0.8306	0.8499	0.7801	0.773	0.7837	0.8644	0.8567	0.8667
						Rec	0.7695	0.7276	0.765	0.6934	0.6886	0.6992	0.8057	0.7951	0.8081
						Prec	0.9629	0.9676	0.956	0.8916	0.881	0.8915	0.9324	0.9286	0.9344
-	-	200	0.2	0.7	True										
						F1	0.8554	0.8306	0.8499	0.7801	0.773	0.7837	0.8644	0.8567	0.8667
						Rec	0.7695	0.7276	0.765	0.6934	0.6886	0.6992	0.8057	0.7951	0.8081
						Prec	0.9629	0.9676	0.956	0.8916	0.881	0.8915	0.9324	0.9286	0.9344
-	-	100	0.2	1.0	True										
						F1	0.6438	0.6177	0.6441	0.6055	0.5974	0.6021	0.7417	0.7314	0.7437
						Rec	0.4908	0.4584	0.4896	0.4757	0.4699	0.472	0.6241	0.6114	0.6252
						Prec	0.9354	0.9465	0.9412	0.8329	0.8198	0.8314	0.9138	0.9099	0.9175
0.5	-	200	0.2	1.0	True										
						F1	0.9936	0.4847	0.4165	0.9704	0.6948	0.7007	0.9624	0.7933	0.791
						Rec	0.9984	0.9882	0.9989	0.9843	0.9826	0.9842	0.9723	0.9693	0.9711
						Prec	0.9889	0.3211	0.2631	0.9568	0.5374	0.544	0.9528	0.6714	0.6672
-	0.5	200	0.2	1.0	True										
						F1	0.9944	0.4545	0.3817	0.9684	0.6836	0.6917	0.9632	0.7925	0.7905
						Rec	0.9977	0.9874	0.9986	0.9816	0.9793	0.9817	0.9733	0.9714	0.972
						Prec	0.9912	0.2952	0.2359	0.9556	0.525	0.534	0.9533	0.6693	0.6661
0.5	-	100	0.2	1.0	True										
						F1	0.9831	0.2906	0.244	0.945	0.5727	0.5866	0.9363	0.7233	0.7266
						Rec	0.9914	0.9765	0.9954	0.9602	0.9588	0.962	0.938	0.9338	0.9383
						Prec	0.975	0.1707	0.1391	0.9302	0.4083	0.422	0.9347	0.5902	0.5928
-	0.5	100	0.2	1.0	True										
						F1	0.9843	0.2767	0.2278	0.943	0.5641	0.579	0.9381	0.7234	0.727
						Rec	0.9935	0.9827	0.9958	0.957	0.9525	0.9588	0.9416	0.9373	0.9409
						Prec	0.9751	0.161	0.1286	0.9294	0.4007	0.4147	0.9346	0.589	0.5924
0.5	-	100	0.1	1.0	True										
						F1	0.9662	0.2047	0.1716	0.9111	0.4684	0.4863	0.8831	0.6059	0.613
						Rec	0.9714	0.9529	0.9775	0.9217	0.9186	0.9242	0.868	0.8636	0.869
						Prec	0.9611	0.1146	0.0941	0.9007	0.3143	0.33	0.8988	0.4667	0.4735
-	0.5	100	0.1	1.0	True										
						F1	0.9662	0.1901	0.1586	0.9115	0.478	0.4944	0.8853	0.6127	0.6216
						Rec	0.9737	0.9584	0.9814	0.9194	0.9151	0.9215	0.8694	0.8636	0.872
						Prec	0.9587	0.1055	0.0863	0.9038	0.3235	0.3379	0.9019	0.4747	0.4829

Table A2: F1, recall and precision scores for all parameter combinations of the *gradient boosting classifier* on the training, validation and holdout set.

Parameter								RN			CCS			CC		
over-samp.	under-samp.	emb-size	epochs	batch-size	#lstm-cells	dropout		Train	Valid.	Hold.	Train	Valid.	Hold.	Train	Valid.	Hold.
-	-	32	2	256	10.0	0.2|0.2										
							F1	0.9936	0.9894	0.9933	0.9905	0.9782	0.981	0.9876	0.9774	0.9778
							Rec	0.9912	0.9851	0.9933	0.9952	0.9864	0.9865	0.9941	0.9857	0.9852
							Prec	0.996	0.9937	0.9933	0.9858	0.97	0.9755	0.9812	0.9692	0.9706
-	-	16	2	256	10.0	0.2|0.2										
							F1	0.9925	0.989	0.9933	0.9864	0.9783	0.9791	0.9841	0.9724	0.9734
							Rec	0.9891	0.9843	0.9933	0.9959	0.9906	0.9886	0.9763	0.9594	0.9599
							Prec	0.9958	0.9937	0.9933	0.9771	0.9664	0.9698	0.992	0.9858	0.9872
-	-	32	3	64	10.0	0.2|0.2										
							F1	0.9919	0.9881	0.991	0.9897	0.9806	0.9808	0.9877	0.9783	0.9781
							Rec	0.985	0.9804	0.987	0.9964	0.9903	0.9889	0.9919	0.9837	0.9834
							Prec	0.9988	0.996	0.995	0.9831	0.9712	0.9728	0.9835	0.9728	0.9728
-	-	32	2	512	10.0	0.2|0.2										
							F1	0.9933	0.9878	0.9916	0.9905	0.9806	0.9797	0.9868	0.976	0.977
							Rec	0.9916	0.9835	0.9919	0.9939	0.9842	0.9818	0.996	0.9894	0.99
							Prec	0.9951	0.9921	0.9912	0.9872	0.977	0.9776	0.9777	0.963	0.9644
-	-	32	3	512	10.0	0.2|0.2										
							F1	0.9935	0.9873	0.9915	0.992	0.9813	0.9809	0.9886	0.9769	0.9774
							Rec	0.9889	0.9796	0.9887	0.9972	0.9886	0.988	0.9967	0.9887	0.9885
							Prec	0.9982	0.9952	0.9943	0.9869	0.9742	0.974	0.9807	0.9655	0.9665
-	-	32	3	256	10.0	0.2|0.2										
							F1	0.9945	0.987	0.9914	0.9928	0.9808	0.9805	0.9906	0.9782	0.9783
							Rec	0.9936	0.9859	0.9947	0.992	0.9795	0.9779	0.9956	0.9868	0.9868
							Prec	0.9953	0.9882	0.9881	0.9936	0.982	0.9831	0.9857	0.9698	0.97
0.5	-	32	3	256	10.0	0.2|0.2										
							F1	1.0	0.9859	0.9864	0.9995	0.9781	0.9795	0.9986	0.9745	0.9746
							Rec	1.0	0.989	0.9951	1.0	0.9814	0.9826	0.9999	0.9853	0.9865
							Prec	0.9999	0.9828	0.9779	0.9991	0.9747	0.9764	0.9973	0.964	0.9631
0.5	-	32	3	256	100.0	0.2|0.2										
							F1	1.0	0.9858	0.9905	0.9997	0.9785	0.9777	0.9993	0.9761	0.9756
							Rec	1.0	0.9796	0.9905	1.0	0.9755	0.9732	1.0	0.9756	0.9766
							Prec	1.0	0.9921	0.9905	0.9995	0.9816	0.9823	0.9986	0.9765	0.9747
-	-	16	3	256	10.0	0.2|0.2										
							F1	0.9899	0.9844	0.9838	0.9908	0.9797	0.9812	0.9876	0.9759	0.976
							Rec	0.998	0.9906	0.9961	0.9923	0.9807	0.9808	0.9879	0.9751	0.975
							Prec	0.982	0.9783	0.9719	0.9893	0.9787	0.9815	0.9874	0.9767	0.9771
-	0.5	32	3	256	100.0	0.2|0.2										
							F1	0.9994	0.8594	0.818	0.9986	0.9555	0.9548	0.998	0.9707	0.9722
							Rec	1.0	1.0	1.0	0.9998	0.9985	0.9981	0.9994	0.9961	0.9965
							Prec	0.9988	0.7534	0.6921	0.9974	0.916	0.9152	0.9966	0.9465	0.9491
-	0.5	32	3	256	10.0	0.2|0.2										
							F1	0.9978	0.6294	0.5796	0.9985	0.9564	0.957	0.9974	0.9561	0.9588
							Rec	1.0	1.0	1.0	0.999	0.9958	0.9965	0.9992	0.9963	0.9957
							Prec	0.9956	0.4593	0.4081	0.9979	0.9201	0.9206	0.9955	0.919	0.9245

Table A3: F1, recall and precision scores for all parameter combinations of the *LSTM-based neural network* on the training, validation and holdout set.

Parameter							RN			CCS			CC		
over-samp.	under-samp.	kernel	sub-sample	class weight	encode type		Train	Valid.	Hold.	Train	Valid.	Hold.	Train	Valid.	Hold.
-	0.5	rbf	0.5	None	True										
						F1	0.0364	0.0066	nan	nan	nan	nan	nan	nan	nan
						Rec	0.0187	0.0134	nan	nan	nan	nan	nan	nan	nan
						Prec	0.6552	0.0044	nan	nan	nan	nan	nan	nan	nan
-	0.5	rbf	0.5	balanced	True										
						F1	0.5257	0.0052	nan	nan	nan	nan	nan	nan	nan
						Rec	0.8069	0.766	nan	nan	nan	nan	nan	nan	nan
						Prec	0.3898	0.0026	nan	nan	nan	nan	nan	nan	nan
-	0.3	rbf	0.5	balanced	True										
						F1	0.4141	0.0052	nan	nan	nan	nan	nan	nan	nan
						Rec	0.8353	0.7877	nan	nan	nan	nan	nan	nan	nan
						Prec	0.2753	0.0026	nan	nan	nan	nan	nan	nan	nan
-	0.1	rbf	0.5	None	True										
						F1	0.0	0.0	nan	nan	nan	nan	nan	nan	nan
						Rec	0.0	0.0	nan	nan	nan	nan	nan	nan	nan
						Prec	0.0	0.0	nan	nan	nan	nan	nan	nan	nan
-	0.5	rbf	0.3	balanced	True										
						F1	nan	nan	nan	0.4476	0.0622	nan	0.4685	0.1115	nan
						Rec	nan	nan	nan	0.5129	0.5004	nan	0.5647	0.5535	nan
						Prec	nan	nan	nan	0.3971	0.0332	nan	0.4004	0.062	nan
-	0.3	rbf	0.3	balanced	True										
						F1	nan	nan	nan	0.3676	0.0629	nan	0.3807	0.1124	nan
						Rec	nan	nan	nan	0.5151	0.501	nan	0.5622	0.5497	nan
						Prec	nan	nan	nan	0.2857	0.0336	nan	0.2878	0.0626	nan
-	0.5	rbf	0.3	None	True										
						F1	nan	nan	nan	0.0318	0.0208	nan	0.0267	0.0189	nan
						Rec	nan	nan	nan	0.0164	0.0134	nan	0.0137	0.0106	nan
						Prec	nan	nan	nan	0.5553	0.0461	nan	0.5741	0.0829	nan
-	0.1	rbf	0.3	None	True										
						F1	nan	nan	nan	0.0	0.0	nan	nan	nan	nan
						Rec	nan	nan	nan	0.0	0.0	nan	nan	nan	nan
						Prec	nan	nan	nan	1.0	0.0	nan	nan	nan	nan

Table A4: F1, recall and precision scores for all parameter combinations of the *support vector classifier* on the training, validation and holdout set. NaN values indicate configurations that we did not test due to time constraints.

Parameter						RN	CCS	Base CC
over-samp.	under-samp.	#trees	class weight	encode type				
-	0.5	100.0	balanced	True				
					F1	0.0539	0.7158	0.8186
					Rec	0.3022	0.758	0.9782
					Prec	0.0296	0.678	0.7038
-	0.5	100.0	None	False				
					F1	0.0509	0.6829	0.8221
					Rec	0.2099	0.6899	0.9783
					Prec	0.0289	0.6761	0.709
-	0.5	100.0	None	True				
					F1	0.0507	0.7148	0.812
					Rec	0.3033	0.7626	0.9796
					Prec	0.0277	0.6726	0.6933
-	0.1	100.0	None	True				
					F1	0.0031	0.3411	0.9419
					Rec	0.0038	0.21	0.9207
					Prec	0.0027	0.9091	0.9641
0.5	-	100.0	balanced	True				
					F1	0.0	0.0295	0.9415
					Rec	0.0	0.015	0.9072
					Prec	0.0	0.9019	0.9784
-	-	100.0	balanced	True				
					F1	0.0	0.0122	0.8954
					Rec	0.0	0.0062	0.8165
					Prec	0.0	0.8855	0.9911
1.0	-	100.0	None	False				
					F1	0.0	0.0157	0.8855
					Rec	0.0	0.0079	0.8059
					Prec	0.0	0.8327	0.9825
-	-	100.0	None	False				
					F1	0.0	0.0036	0.8709
					Rec	0.0	0.0018	0.7754
					Prec	0.0	0.8095	0.9931
1.0	-	100.0	None	True				
					F1	0.0	0.0509	0.9355
					Rec	0.0	0.0261	0.8947
					Prec	0.0	0.9295	0.9802
0.5	-	100.0	None	True				
					F1	0.0	0.0317	0.9453
					Rec	0.0	0.0161	0.9124
					Prec	0.0	0.895	0.9806
-	-	100.0	None	True				
					F1	0.0	0.0103	0.9341
					Rec	0.0	0.0052	0.8858
					Prec	0.0	0.8567	0.9879
-	-	100.0	balanced	False				
					F1	0.0	0.0039	0.7969
					Rec	0.0	0.0019	0.665
					Prec	0.0	0.8527	0.9942
0.5	-	100.0	None	False				
					F1	0.0	0.0141	0.8984
					Rec	0.0	0.0071	0.8266
					Prec	0.0	0.883	0.9838

Table A5: F1, recall and precision scores for all parameter combinations of the *random forest classifier* on the manipulated holdout data. The "Base CC" column indicates the performance of the models trained on the CC type for comparison.

Parameter							RN	CCS	Base CC
over-samp.	under-samp.	stages	learn-rate	sub-sample	encode type				
0.5	-	200	0.2	1.0	True				
						F1	0.24	0.83	0.7519
						Rec	0.7676	0.978	0.9854
						Prec	0.1422	0.7209	0.6078
-	0.5	200	0.2	1.0	True				
						F1	0.2309	0.8291	0.7466
						Rec	0.8196	0.9773	0.9853
						Prec	0.1344	0.7199	0.601
-	-	100	0.2	1.0	True				
						F1	0.15	0.5254	0.8648
						Rec	0.0891	0.3629	0.8225
						Prec	0.4729	0.9513	0.9117
0.5	-	100	0.2	1.0	True				
						F1	0.1488	0.7314	0.6883
						Rec	0.8706	0.9781	0.9789
						Prec	0.0814	0.5841	0.5307
-	0.5	100	0.2	1.0	True				
						F1	0.1435	0.7303	0.6874
						Rec	0.9055	0.9783	0.9817
						Prec	0.0779	0.5827	0.5288
0.5	-	100	0.1	1.0	True				
						F1	0.1063	0.6377	0.5957
						Rec	0.891	0.9758	0.977
						Prec	0.0565	0.4736	0.4285
-	0.5	100	0.1	1.0	True				
						F1	0.0993	0.6454	0.607
						Rec	0.9077	0.9786	0.9769
						Prec	0.0525	0.4815	0.4402
-	-	200	0.2	0.4	True				
						F1	0.0963	0.3292	0.8948
						Rec	0.0558	0.1995	0.8805
						Prec	0.349	0.9406	0.9096
-	-	200	0.2	1.0	True				
						F1	0.0963	0.3292	0.8948
						Rec	0.0558	0.1995	0.8805
						Prec	0.349	0.9406	0.9096
-	-	200	0.2	0.7	True				
						F1	0.0963	0.3292	0.8948
						Rec	0.0558	0.1995	0.8805
						Prec	0.349	0.9406	0.9096
-	-	300	0.2	1.0	True				
						F1	0.0563	0.2333	0.9174
						Rec	0.0322	0.1331	0.9105
						Prec	0.2222	0.942	0.9243

Table A6: F1, recall and precision scores for all parameter combinations of the *gradient boosting classifier* on the manipulated holdout data. The "Base CC" column indicates the performance of the models trained on the CC type for comparison.

Parameter								RN	CCS	Base CC
over-samp.	under-samp.	emb-size	epochs	batch-size	#lstm-cells	dropout				
-	0.5	32	3	256	10.0	0.2|0.2	*F1*	0.3895	0.6568	0.9502
							Rec	0.847	0.5009	0.996
							Prec	0.2529	0.9534	0.9085
-	-	32	2	512	10.0	0.2|0.2	*F1*	0.0	0.3195	0.9694
							Rec	0.0	0.1902	0.9931
							Prec	0.0	0.9968	0.9467
-	-	16	3	256	10.0	0.2|0.2	*F1*	0.0	0.2831	0.9698
							Rec	0.0	0.165	0.9737
							Prec	0.0	0.9956	0.9659
0.5	-	32	3	256	10.0	0.2|0.2	*F1*	0.0	0.119	0.9692
							Rec	0.0	0.0633	0.9947
							Prec	0.0	0.9851	0.945
-	-	32	2	256	10.0	0.2|0.2	*F1*	0.0	0.1898	0.9728
							Rec	0.0	0.1049	0.988
							Prec	0.0	0.9938	0.958
-	-	32	3	512	10.0	0.2|0.2	*F1*	0.0	0.0037	0.9714
							Rec	0.0	0.0019	0.9929
							Prec	0.0	0.7047	0.9508
-	-	32	3	64	10.0	0.2|0.2	*F1*	0.0	0.0857	0.9721
							Rec	0.0	0.0448	0.9788
							Prec	0.0	0.9742	0.9655
-	-	32	3	256	10.0	0.2|0.2	*F1*	0.0	0.0451	0.9711
							Rec	0.0	0.0231	0.9937
							Prec	0.0	0.9645	0.9494
-	-	16	2	256	10.0	0.2|0.2	*F1*	0.0	0.5738	0.972
							Rec	0.0	0.4036	0.9676
							Prec	0.0	0.9925	0.9765

Table A7: F1, recall and precision scores for all parameter combinations of the *LSTM-based neural network* on the manipulated holdout data. The "Base CC" column indicates the performance of the models trained on the CC type for comparison.

Tokenizer Types

0: 'ENDMARKER'
1: 'NAME'
2: 'NUMBER'
3: 'STRING'
4: 'NEWLINE'
5: 'INDENT'
6: 'DEDENT'
7: 'LPAR'
8: 'RPAR'
9: 'LSQB'
10: 'RSQB'
11: 'COLON'
12: 'COMMA'
13: 'SEMI'
14: 'PLUS'
15: 'MINUS'
16: 'STAR'
17: 'SLASH'
18: 'VBAR'
19: 'AMPER'
20: 'LESS'
21: 'GREATER'
22: 'EQUAL'
23: 'DOT'
24: 'PERCENT'
25: 'LBRACE'
26: 'RBRACE'
27: 'EQEQUAL'
28: 'NOTEQUAL'
29: 'LESSEQUAL'
30: 'GREATEREQUAL'

31: 'TILDE'
32: 'CIRCUMFLEX'
33: 'LEFTSHIFT'
34: 'RIGHTSHIFT'
35: 'DOUBLESTAR'
36: 'PLUSEQUAL'
37: 'MINEQUAL'
38: 'STAREQUAL'
39: 'SLASHEQUAL'
40: 'PERCENTEQUAL'
41: 'AMPEREQUAL'
42: 'VBAREQUAL'
43: 'CIRCUMFLEXEQUAL'
44: 'LEFTSHIFTEQUAL'
45: 'RIGHTSHIFTEQUAL'
46: 'DOUBLESTAREQUAL'
47: 'DOUBLESLASH'
48: 'DOUBLESLASHEQUAL'
49: 'AT'
50: 'ATEQUAL'
51: 'RARROW'
52: 'ELLIPSIS'
53: 'OP'
54: 'AWAIT'
55: 'ASYNC'
56: 'ERRORTOKEN'
57: 'COMMENT'
58: 'NL'
59: 'ENCODING'
256: 'NT_OFFSET'

Figure A1: Possible token types of the tokenizer for our type encoding. Extracted from the `tokenize.tok_name` attribute.

Dataset

Organisation	Repository	Branch	Git Hash
scikit-learn	scikit-learn	master	8ba49f628092fe3fb1deea101910006aa6c76d49
pytorch	pytorch	master	2b70f827373fe5c90e15ce7ba0a1d331dcd57379
ansible	ansible	devel	9792d631b1b92356d41e9a792de4b2479a1bce44
django	django	master	9c92924cd5d164701e2514e1c2d6574126bd7cc2
pallets	flask	master	6d3f87ee071ea061f0af2ba7d29b148841090b59
keras-team	keras	master	1a3ee8441933fc007be6b2beb47af67998d50737
jakubroztocil	httpie	master	100872b5cf493b355453a9dff89f39b0c124d524
home-assistant	core	dev	dc84196202a4683881ddac9942fe1c9af6de3b5b
certbot	certbot	master	2a047eb526ee02d4bb3bb3668260ac6d481b99d3
pandas-dev	pandas	master	e666f5cc59a8631a5ee3e38c20ad8e08531cbec3
docker	compose	master	76963e44add9810c1d906b5fbd052dc3fb480479
tiangolo	fastapi	master	a6897963d5ff2c836313c3b69fc6062051c07a63
tornadoweb	tornado	master	4a4cf9ce5867d29b4557a0b28eca7a56e77207de
encode	django-rest-framework	master	7f3a3557a050147dd2420aa41d1bf7ddd7f9818e
google	python-fire	master	79d85df2706b2dab5dd5075f2a76953743ee9bf3
explosion	spaCy	master	7d7b65ffd42c56ce3a0aa73b18196eb20a1dcc24
bokeh	bokeh	master	66231e91c3a806de73d77f7c3617e028d28e7c6c
apache	airflow	master	3a349624a20d3432dc75e337d6ffb1109a50e451

Table A8: A list of open-source repositories that we used for our dataset.