Software Bug Detection Using the N-gram Language Model

by

Devin Chollak

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science, Software Engineering

Waterloo, Ontario, Canada, 2015

© Devin Chollak 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Over the years many techniques have been proposed to infer programming rules in order to improve software reliability. The techniques use violations of these programming rules to detect software defects. This thesis introduces an approach, *NGDetection*, which models a software's source code using the n-gram language model in order to find bugs and refactoring opportunities in a number of open source Java projects. The use of the n-gram model to infer programming rules for software defect detection is a new domain for the application of the n-gram model.

In addition to the n-gram model, *NGDetection* leverages two additional techniques to address limitations of existing defect detection techniques. First, the approach infers combined programming rules, which are a combination of infrequent programming rules with their related programming rules, to detect defects in a way other approaches cannot. Second, the approach integrates control flow into the n-gram model which increases the accuracy of defect detection.

The approach is evaluated on 14 open source Java projects which range from 36 thousand lines of code (KLOC) to 1 million lines of code (MLOC). The approach detected 310 violations in the latest version of the projects, 108 of which are useful violations, i.e., 43 bugs and 65 refactoring opportunities. Of the 43 bugs, 32 were reported to the developers and the remaining are in the process of being reported. Among the reported bugs, 2 have been confirmed by the developers, while the rest await confirmation. For the 108 useful violations, at least 26 cannot be detected by existing techniques.

Acknowledgements

I would like to thank my supervisor, Dr. Lin Tan, who introduced me to this topic and provided guidance during my research. I want to thank my readers Dr. Reid Holmes and Dr. Derek Rayside for taking the time to read my thesis and provide feedback.

Dedication

This thesis is dedicated to those who have helped me during my studies and to the ones I love.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Motivation . . . 1

1.2 Proposed Approaches . . . 2

1.2.1 N-gram Language Model . . . 2

1.2.2 Combined Rules . . . 3

1.2.3 Control Flow . . . 3

1.3 Contributions . . . 4

2 Related Work 6

2.1 Background . . . 6

2.2 Language Models for Program Analysis . . . 7

2.3 Rule Mining and Defect Detection . . . 7

3 Approach 9

3.1 Challenges and Key Techniques . . . 9

3.2 Tokenization . . . 11

3.3 N-gram Model Building . . . 14

3.4 Rule Generation and Violation Detection 14
3.4.1 Parameters for Rule Pruning 15
3.4.2 Minimum Gram Size 16
3.4.3 Maximum Gram Size 16
3.4.4 Upper Rule Threshold 17
3.4.5 Lower Rule Threshold 17
3.4.6 Anomaly Threshold 17
3.4.7 Minimum Method Count 18
3.4.8 Maximum Violation Count 18
3.5 Other Rule Pruning 18

4 Results **20**
4.1 Experimental Setup 20
4.2 Evaluated Projects 21
4.3 Programming Rule Results 22
4.4 Violation Detection Results 24
4.4.1 Impact of Combined Rules 25
4.4.2 False Positives 25
4.4.3 Example Bugs 26
4.4.4 Example Refactor Rules 28
4.4.5 Comparing With Existing Techniques 30
4.5 Parameter Sensitivity 31
4.5.1 N-Gram Size Limits 32
4.5.2 Rule Threshold Limits 32
4.5.3 Count-based Limits 33
4.6 Execution Time and Space 34

5 Conclusion and Future Work **35**

References **39**

List of Tables

4.1 Evaluated Projects 21
4.2 Number of Inferred Programming Rules 22
4.3 Violation Detection Results 24
4.4 Parameter Sensitivity of *NGDetection* on GeoTools 31
4.5 Execution Time in Seconds 33

List of Figures

3.1 Overview of *NGDetection* 10

3.2 Control Flow Example 11

4.1 Project Rule Sizes 23

4.2 Inferred Combined Rule 27

4.3 Detected Bug Example 28

4.4 Refactor Violation Example 29

4.5 Refactor Rule Example 30

Chapter 1

Introduction

Even in mature software projects, existing bug detection techniques have shown that it is possible to find bugs which were previously unknown [1, 2]. Some approaches operate by finding implicit programming rules in a software project. These programming rules can then be used to find violations which has the potential to find software defects. Various methods exist which can mine programming rules from a software project, such as mining program source code [3–11], version histories [5, 12, 13], and source code comments or documentation [14, 15]. Bug detection techniques will often use these mined rules in order to detect violations as potential bugs.

1.1 Motivation

As motivation for this work, a number of existing rule-based bug detection approaches [3–5, 7, 16] have obtained significant progress in finding bugs by making use of frequent itemset mining techniques. These bug detection approaches identify frequent method calls and variable accesses as programming rules, and then use these rules to detect violations. As a simple example, say three methods called `A`, `B`, and `C` are sequentially called together in multiple locations of the program. This would give a developer the impression that if method `A` and `B` were called, you would then be expected to then call method `C`. Bug detection tools such as PR-Miner [3] would infer the rule that calls to methods `A` and `B` should also have a call to method `C`, and a violation would be the a locations where method `C` is not called. This violation could then be reported as a potential bug for the developer to evaluate.

It has recently been demonstrated that *n-gram language models* can capture the regularity of software, which have been used for software engineering tasks including code completion and suggestion [17–20]. Naturally, other software engineering tasks may also be applicable to modeling software using the n-gram language model. Therefore it would be beneficial to leverage the n-gram language model to represent software and use it as the basis for detecting violations in source code. This the approach presented, referred to as *NGDetection*, uses an n-gram language model based bug detection technique explained in this thesis[1].

1.2 Proposed Approaches

This novel idea is to use a simple natural language processing technique in order to resolve some of the outlined downsides with other approaches. Specifically, *NGDetection* utilizes the three following techniques to complement existing rule-based bug detection techniques.

1.2.1 N-gram Language Model

Many existing techniques [3–5, 7] do not consider the order of method calls. This has potential to miss opportunities to detect bugs caused by incorrect calling order of methods. In addition, existing frequent itemset mining based approaches [3–5, 7] do not recognize the same method being called multiple times in the itemset. For example, if a project contains 99 sequences of calls to methods `A` and `B`, denoted as `AB`, together with a single sequence of calls to methods `A`, `B`, and `B`, denoted as `ABB`. Existing techniques represent all 100 sequences as a set {`A, B`}, thus fail to detect the potential bug of the `ABB` sequence. It is common that certain methods, e.g., `B` in the example, should not be called consecutively.

NGDetection makes use of the n-gram language model [21] which provides a Markov model for modeling tokens. The n-gram model considers the order of tokens and recognizes multiple calls to the same method, both of which are often neglected by existing approaches. Therefore since *NGDetection* uses the n-gram language model it can detect bugs that existing techniques cannot.

[1] This thesis is based off the work in a conference submission.

1.2.2 Combined Rules

Existing techniques typically infer a single frequent set or sequence as a rule, and do not take full advantage of multiple sets or sequences, missing opportunities to detect new bugs. For example, if a sequence of calls to methods `A`, `B`, and `C`, denoted as `ABC`, appear 49 times, and the sequence `ABD` appears 49 times. It would then make the two appearances of the sequence `ABE` be a potential violation of the programming rule where `AB` should be followed by `C` or `D`, denoted as `AB(C|D)`. Existing techniques such as PR-Miner [3] require the confidence for a single sequence, e.g., `ABC`, to be over 90% in order to avoid generating a large number of false violations. In this example, the confidence of the sequence `ABC`, which is the probability of `C` appearing after `AB`, is only $\frac{49}{49+49+2} = \frac{49}{100}$. Therefore, these existing techniques would miss the two potential violations of `ABE`.

To address this issue, *NGDetection* considers multiple sequences together to infer combined rules. The combined rule inference considers the combination of infrequent sequences for rules. In the `ABE` bug example, *NGDetection* combines relevant sequences and only requires the combined confidence, i.e., $\frac{49+49}{49+49+2} = \frac{98}{100}$ for the two sequences `ABC` and `ABD`, to pass a certain threshold. Our results confirm such combined rules are common in projects, which have helped us detect 14 new bugs (Section 4). Alattin [7] infers the combination of rules, however, it focuses on branch condition related rules. In addition, Alattin does not use an n-gram model (a detailed discussion in Section 2).

1.2.3 Control Flow

Most approaches do not consider control flow when learning programming rules. This results in similar sequences of method calls being considered equivalent when they are semantically different, which causes false bugs to be detected (Section 3.2). To improve the detection accuracy, *NGDetection* takes into consideration a program's control flow when inferring programming rules. For example, consider the following snippet of code:

```
1 A();
2 B();
3 if (condition) {
4     C();
5 } else {
6     D();
7 }
```

The sequence of method calls in the above code is `ABCD` where `C` is called in an if branch and `D` is called in the else branch. However, it can be easily seen that the rule `ABCD` does not make sense since the `C` and `D` methods will never be called in order. Instead it would be expected to have the condition be true then execute `ABC` or the condition be false then execute `ABD`. Intuitively, these are two separate control flow paths and should be treated as `AB(C|D)` instead of `ABCD`. The key concept behind using control flow is that control flow has a significant impact on semantics and should not be ignored. If ignored, it leads to situations where patterns such as `ABCD` are discovered when in reality `C` and `D` are never executed together.

1.3 Contributions

This thesis demonstrates the use of the n-gram language model on a new domain: software defect detection and makes the following contributions:

- Proposes a new approach, called *NGDetection*, to model source code and extract programming rules for detecting bugs. It differs from existing bug detection approaches by leveraging a natural language model. To the best of our knowledge, this is the **first work** to leverage n-gram language model for bug detection.
- Combines n-gram models with two additional techniques to address limitations of existing bug detection approaches. Specifically, the approach combines multiple rules to detect new bugs that are not detected by existing techniques. In addition, it incorporates control flow for rule inference and bug detection processes to improve the accuracy of the bug detection. While the approach in this thesis uses these two techniques with the n-gram model, these two techniques are orthogonal to the approach, and are applicable to other rule inference approaches.
- An evaluation of *NGDetection* on 14 open source Java projects ranging from 36 thousand lines of code to 1 million lines of code. Results show that *NGDetection* detects 310 violations in the latest versions of the evaluated projects, 108 of which are useful violations—43 bugs and 65 refactoring opportunities. 32 out of the 43 bugs are reported to the developers and the rest are in the process of being reported. Among the reported bugs, 2 have been confirmed by developers, while the rest await confirmation. Among the 108 useful violations, at least 26 cannot be detected by existing techniques such as PR-Miner [3].

In this thesis, Chapter 2 briefly goes into detail on the n-gram language model and outlines various related work in programming rule based bug detection. Chapter 3 explains the design and implementation of *NGDetection* as well as describing in detail about the

tunable parameters for the technique. Chapter 4 provides information on the setup for the experiments as well as detailed results for programming rules, bug detection, and parameter influence. Finally, Chapter 5 summarizes the findings of this thesis, the limitations of the current approach, and the future directions for this approach.

Chapter 2

Related Work

In this chapter, Section 2.1 details the n-gram language model and provides a natural language example of how it can be used to predict words. In Section 2.2, an overview of some of the related work on using statistical language models for program analysis. In Section 2.3, details some of the related work on program rule mining and its usage for defect detection.

2.1 Background

The n-gram language model has been used in modelling natural language [21] and solving problems such as speech recognition [22], statistical machine translation, and other related language challenges [23]. The n-gram language model typically consists of two components, words and sentences, where each sentence is an ordered sequence of words. The language model can build a probabilistic distribution over all possible sentences in a language using Markov chains. The model contains a dictionary D, which lists all possible words of a language, where each word is represented as w. The probability of a sentence in a language is estimated by building a Markov chain with $n-1$ words. That is, the probability of a sentence can be calculated by using the conditional probabilities of the previous $n-1$ words. Given a sentence $s = w_1 \cdot w_2 \cdot w_3 \cdot ... \cdot w_m$. Where its probability is estimated as:

$$P(s) = \prod_{i=1}^{m} P(w_i|h_{i-1})$$

where, the sequence $h_i = w_{i-n} \cdot ... \cdot w_i$ is the history. In the n-gram language model, the probability of the next word w_i depends only on the previous $n-1$ words.

For example, say we are given the following phrases and the number of times they occur in a document: "I played hockey" occurred 16 times, "I played football" occurred 3 times, and "I played tennis" occurred a single time. Then when we see the words "I played" we would expect "hockey" to be the next word with probability of $16/20 = 80\%$, "football" would be $3/20 = 15\%$, and "tennis" would be $1/20 = 5\%$ of the time. Thus, we would predict that if we saw the phrase "I played" there is an 80% chance the next word would be "hockey". While the n-gram model has applications in natural language processing this approach seeks to utilize it for representing and analyzing code to find bugs. Specifically, programming rules are mined from sequences of method calls with control flow information, the rules are then used to detect rule violations for bug detection.

2.2 Language Models for Program Analysis

Statistical language models have been successfully used for program analysis with a wide variety of techniques. Hindle et al. [17] first leveraged the n-gram language model to show that software source code has high repetitiveness, and the n-gram language model can be used in code suggestion and completion. This work laid the groundwork for using language models to model source code and demonstrated how they could be used in software tools. Han et al. [19] presented an algorithm to infer the next token by using a Hidden Markov Model for code completion and demonstrated a high level of accuracy. Nguyen et al. [20] proposed a code suggestion method named SLAMC, which incorporated semantic information into an n-gram model. It demonstrated how tokens can be seen more semantically instead of just syntactically. Raychev et al. [18] investigated the effectiveness of various language models, i.e., n-gram and recurrent neural networks, for code completion. By combining program analysis and the n-gram model, they proposed SLANG, which aimed to predict the sequence of method calls in a software system. Allamanis et al. [24] used the n-gram model to learn local style from a codebase and provided suggestions to improve stylistic consistency. Other work using the n-gram language model has focused on style [24] and code suggestion [18, 20]. This work leverages the n-gram model on a new domain—software defect detection.

2.3 Rule Mining and Defect Detection

Over the last decade many techniques have been developed for mining programming rules and bug detection [3–14, 25–30]. Many of these techniques are able to find many patterns,

where even the simple patterns show success in detecting bugs [1]. The patterns can be in the form of simple manually written code templates which are pattern matched to parts of a code base [4]. Other methods are more complicated and find patterns in the source code or comments which allow for a solution that can learn new conventions. Ramanathan et al. [31] used inter-procedural analysis to find and detect violations of function precedence.

Li et al. [3] developed PR-Miner to mine programming rules from C code and detect violations using frequent itemset mining. Benjamin et al. [5] proposed DynaMine which used association rule mining to extract simple rules from software revision histories for Java code and detect defects related to rule violations. Many frequent itemset mining-based tools do not consider program control flow or the combination of mined rules. Chang et al. [6] proposed a static bug detection approach to mine rules for detecting neglected conditions. Their approach considered control flow by combining frequent itemset mining and frequent subgraph mining on C code. Jeff et al. [32] proposed a sound predictive race detection method incorporating control flow. Suresh et at. [7] presented an approach to detect neglected conditions using different combinations of rules mined from condition check points in source code, i.e., conjunctive, disjunctive, exclusive, and disjunctive combinations. Their evaluation shows that the conjunctive pattern of their approach produces the best performance in terms of reducing false positives. Their work focused on detecting conditional related bugs differs from our work on control flow and method calls. *NGDetection*'s approach differs from the above approaches in two distinct ways. First, the use of the n-gram language model to represent source code which enables it to use the ordering information between method calls and handles repetitive method calls. Second, *NGDetection* considers both program control flow and the combination of rules, which are helpful for reducing false positives and detecting bugs that existing techniques cannot detect.

Chapter 3

Approach

Figure 3.1 displays how the approach consists of two main components: (1) taking a given software project and building the n-gram language model, and (2) using the n-gram language model to generate programming rules and detecting code violations to the discovered programming rules. This section first describes the challenges and key approaches proposed to address these challenges (Section 3.1). The next section presents how to collect high-level tokens from a project (Section 3.2), and then use the tokens to build the n-gram language model for the project (Section 3.3). Finally, an explanation on how to generate programming rules from the n-gram model and use the programming rules to detect potential bugs in a project (Section 3.4), and how to prune the low quality or invalid rules to reduce false positives (Section 3.5).

3.1 Challenges and Key Techniques

A main challenge in step 1 (Building the N-gram Model) is selecting a suitable level of granularity for tokens when building the n-gram model. Existing work builds n-gram models at the syntactic level using low-level tokens to suggest the next tokens for code completion and suggestion [17,19]. For example, after seeing "`for (int i=0; i<n;`", the model would continue to suggest the following tokens "`i++) {`". When building n-gram models at this level, it is likely to only detect simple syntactic errors, e.g., missing "`;`" or "`i++`" in a `for` loop, which would normally be caught by a compiler. At this level the inferred rules are highly localized to the syntax of the code and do not provide a significant amount of semantics about the code.

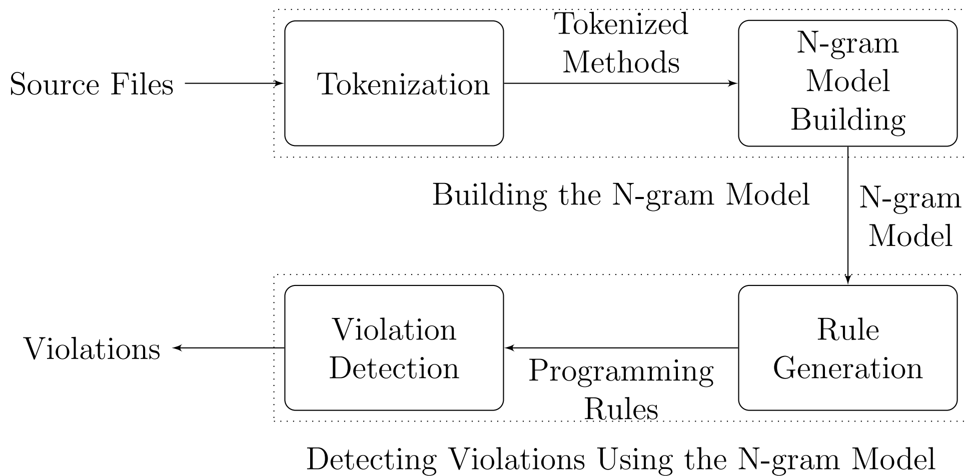

Figure 3.1: Overview of *NGDetection*

The approach aims to detect bugs at the semantic level, thus it is required to build the n-gram model at a higher level. *NGDetection* selects high-level tokens to build a semantic n-gram model for bug detection. This requires keeping certain elements while ignoring others which mostly create noise without providing significant semantic information. Additionally, some tokens are augmented with additional information, such as capturing the fully qualified names for method and constructor invocations. Take the following loop as an example:

```
for (int i=0; i<n; i++) {
    foo(i);
}
```

NGDetection will represent it with the following high-level tokens `[<FOR>, foo(), <END_FOR>]`, while a syntactic-level n-gram model may represent it with the following low-level tokens `[for, (, int, i, =, 0, ;, i, <, n, ; i, ++, ), {, foo, (, i, ), }]`. The details of the selected high-level tokens are presented in Section 3.3 as well as a detailed discussion about other possible options and considerations in Chapter 5.

As discussed in Section 1.2, control flow information can be important for the accuracy of bug detection. Figure 3.2 shows one of 40 code snippets in Elasticsearch from which the programming rule sequence `[startArray(), value(), endArray()]` can be learnt when not considering control flow. Using this rule, the violation shown in the figure would be detected because `[startArray(), value()]` are followed by `value()` instead of `endArray()`.

Rule without using control flow:
[startArray(), value(), endArray()]
Example source code that follows the rule:

```
startArray(name);
for (Object o : var) {
    value(o);
}
endArray();
return this;
```

Pattern of the false violation:
[startArray(), value(), value()]
False violation:

```
return builder.startArray().value(coordinate.x)
              .value(coordinate.y).endArray();
```

Figure 3.2: Lack of control flow introduces a false violation.

However, these two code snippets are quite different semantically, thus it should be legitimate for the second code snippet to call value() more than once. *NGDetection* adds the control flow regarding for loop to the sequence, the rule then becomes [startArray(), <FOR>, value(), <FOR_END>, endArray()] (all 40 code snippets have the for loop control flow), and avoids detecting the false violation shown in Figure 3.2.

3.2 Tokenization

NGDetection uses the Eclipse JDT Core[1] to tokenize the source files, construct the abstract syntax trees (AST), and resolve the type information for the tokens. Following PR-Miner [3], *NGDetection* considers *methods* as the main type of token, because a method is a natural unit for developers to organize semantically related functionalities. By using JDT's provided AST parser, key parts of the source code can be extracted such as: constructors, initializers, class methods, and inner/local classes. It then allows *NGDetection* to appropriately match each method call, and other control flow structures outlined below into the appropriate locations inside of the class.

[1] https://eclipse.org/jdt/core/index.php

As discussed at the beginning of Chapter 3, building an n-gram model requires determining the right level of granularity for tokens. Instead of considering all syntactic elements there is a focus on method calls and control flow which are:

- method calls, constructors and initializers,
- `if/else` branches,
- `for/do/while/foreach` loops,
- `break/continue` statements including optional labels,
- `try/catch/finally` blocks,
- `return` statements,
- `synchronized` blocks,
- `switch` statements and `case/default` branches, and
- `assert` statements

A method call, such as `methodA()`, is resolved to its fully qualified method name `org.example.Foo.methodA()`. This is to prevent unrelated methods with an identical name from being grouped together. It is an important aspect as it ensures the rules and more importantly the violations found are actually related to each other instead of simply having the same name. In addition, the type of exception in the `catch` clauses are considered as they provide important context information to help infer more accurate rules. Only knowing if a segment of code is in a catch clause does not provide enough information about the rules relating to them. For example, a project may log all IOExceptions, but ignore all InterruptedExceptions. Having a way to differentiate them is important for ensuring rules are being correctly identified and the violations detected are appropriately related.

Anonymous class definitions are resolved to their parent class where possible. While technically each anonymous class definition is treated as a different class by the compiler, it is not useful if they are all considered different classes. If the anonymous class definitions are considered different, then no patterns will be found for them because the signatures would be considered different. To further explain how tokens are extracted from projects, see the following method definition as an example:

```
1 public Properties loadProperties(final String name) {
2   final Properties properties = new Properties();
3   try (FileInputStream in = new FileInputStream(name)) {
4     properties.load(in);
5   } catch (final FileNotFoundException e) {
6     try {
7       final File file = new File(name);
8       file.getParentFile()
```

```
9         .mkdirs();
10       file.createNewFile();
11     } catch (final IOException innerE) {
12       Log.error("Error creating setting file " + name, innerE);
13     }
14   } catch (final Exception e) {
15     Log.error("Error loading setting file " + name, e);
16   }
17   return properties;
18 }
```

The above code segment is translated into the following sequence of tokens, which are used as input for building the n-gram model:

- <begin_method>,
- java.util.Properties.Properties,
- <TRY>,
- java.io.FileInputStream.FileInputStream,
- java.util.Properties.load,
- <CATCH_java.io.FileNotFoundException>,
- <TRY>,
- java.io.File.File,
- java.io.File.getParentFile,
- java.io.File.mkdirs,
- java.io.File.createNewFile,
- <CATCH_java.io.IOException>,
- org.example.Log.error,
- <END_TRY>,
- <CATCH_java.lang.Exception>,
- org.example.Log.error,
- <END_TRY>,
- <RETURN>,
- <end_method>.

As can be seen in the above sequence the method declaration is not included, but the method boundaries are included. This allows for rule definitions to include the beginning and end of a method body which is useful for the situations where a rule is defined at the start of method, such as a null check.

3.3 N-gram Model Building

For every sequence of tokens extracted from a method, all of its subsequences are added to the language model. Then for each set of sequences the conditional probabilities of each subsequence are calculated. It is normal for n-gram models to use smoothing as it helps with handling unknown sequences. However, since the entire source code of a project is being used and *NGDetection* has the complete language of all possible sequences for a given version. This means smoothing is unnecessary since there are no unknown sequences.

While open source software packages for building n-gram models exist, e.g., OpenNLP[2], *NGDetection* has it's own implementation for building the n-gram model. This is because it was necessary to not only build n-grams, but it also needed to use the n-grams to generate programming rules (Section 3.4), the latter of which is not well supported by existing software packages. In addition, the implementation was needed to be thread-safe which is not provided by existing packages. The algorithm used to build the n-gram model is a standard implementation, which is described in Section 2.1.

3.4 Rule Generation and Violation Detection

For step 2 of the process (Detecting Violations Using the N-gram Model), a key component is *combined rules*, this infers more complex rules such as `AB(C|D)`, which can enable *NGDetection* to detect bugs not found with previous approaches. Combined rules account for 104 out of the total 310 detected violations while only accounting for 1.7% of all rules. This strongly suggests these rules are not as common as single rules, but are significantly more potent. The results confirm combined rules are common in projects, which helped detect 14 new bugs (Section 4.4).

To leverage the n-gram model for detecting bugs, programming rules are inferred from the n-gram model, and then violations to the programming rules are detected. *NGDetection* infers two types of rules: *single rules* and *combined rules*. Single rules are single a sequence such as `ABC`, meaning method `C` should follow the sequence of calls to method `A` and `B`. Combined rules are a combination of single sequence rules, such as `AB(C|D)`, meaning method `C` or method `D` should follow the sequence of calls to method `A` and `B`.

Combined rules are an important aspect of *NGDetection* which allows it to find new bugs. When determining the situations to infer a combined rule, a single rule, or no rule,

[2] https://opennlp.apache.org/

the conditional probabilities and *NGDetection*'s thresholds are used. As an example, given the following three sequences `ABC`, `ABD`, and `ABE`, each of which occurs 66%, 33%, and 1% of the time respectively. Since the sequence `ABC` occurs 66% of the time, it could arguably be considered a single rule: `C` must follow `AB`. However, this rule suggests the sequences `ABD` and `ABE` are wrong, indicating 34% of what the developers wrote is incorrect, which seems highly questionable. Instead, the rule `AB(C|D)` is inferred and `ABE` is considered a violation. In general, *NGDetection* requires the rule probability, either single or combined, to be at least a certain threshold, referred to as the `Upper Rule Threshold` (details in Section 3.4.1). A reasonable value for the `Upper Rule Threshold` is 80%, which is used in the experiments for all evaluated projects. Increasing it reduces the number of generated rules, thus detecting fewer bugs, but also detects fewer false positives. A detailed study of its sensitivity is presented in Section 4.5.

If the occurrences of the three sequences become 79%, 11%, and 10% respectively, and the `Upper Rule Threshold` is 80%, then there is no sequence greater than the `Upper Rule Threshold`. Since `ABD` is now in the minority at 11%, whether to consider it as part of a combined rule is now more questionable. To address this problem, the `Lower Rule Threshold` (details in Section 3.4.1) is used to filter out sequences from being combined. If the `Lower Rule Threshold` is set to be 15%, then `ABD` will not be combined with `ABC`. Thus, no rule would be inferred, which helps *NGDetection* avoid generating too many false positives.

In order to generate programming rules from an n-gram model, the n-gram model needs to keep track of parent and children sequences, as well as the source locations, for a given sequence. A *parent* sequence of the sequence `ABCD` is the sequence `ABC`. A *child* sequence of the sequence `ABCD` is the sequence `ABCDE`. The linking between parent and child sequences needs to be tracked in order to find which sequences are used in order to find a violation. The location of every sequence is added to the n-gram model as well. This allows for a violation or a rule to be traced back to its source location so it can be analyzed by a developer. While these are not normally part of the n-gram model, they are important when it comes to leveraging an n-gram model for inferring programming rules and finding bugs.

3.4.1 Parameters for Rule Pruning

There are a number of parameters to filter the generated rules. These parameters influence each other and affect the number of generated rules and violations as well as their accuracy. Below is a list of these parameters and a quick description of each:

- **Minimum Gram Size** - The smallest length of sequences to be considered.
- **Maximum Gram Size** - The largest length of sequences to be considered.
- **Upper Rule Threshold** - The percent a single sequence or combination of sequences must occur in order to be a rule.
- **Lower Rule Threshold** - The percent a sequence must occur in order to be considered part of a combined rule. Must be smaller than upper threshold.
- **Anomaly Threshold** - The percent limit a sequence can occur in order to be considered a violation of a rule. Must be smaller than lower threshold.
- **Minimum Method Count** - The minimum number of method calls required to be in a rule.
- **Maximum Violation Count** - The maximum number of violations detected by a rule for it to be included.

3.4.2 Minimum Gram Size

This is the smallest length of sequences to be considered. The rationale for this parameter is to prune the small grams from the data which lack utility. For example, the bigram `[<RETURN>, <end_method>]` would be added to the model. However, this bigram and many others like it would occur often in a codebase. Say it happens 98% of the time, then the 2% of the time where a `<RETURN>` is not followed by a `<end_method>` would be detected as a violation. In most situations it is clearly not a bug, thus the bigram is not useful for finding bugs because the shear number of them would result in an infeasible number of false positives to prune. For the evaluated projects, only grams greater than 4 elements are considered, this is based on empirical results.

3.4.3 Maximum Gram Size

This is the largest length of sequences to be considered. As the gram size increases, the total number of n-grams increases but each gram becomes more unique. This results in a large number of sequences which occur only a few times, and because they are so infrequent they will rarely be able to detect violations. Rules are generated based on the frequency of sequences and require a minimum number of occurrences before they can detect violations.

As can be seen in Section 4.5 changing this value makes very little difference other than changing the number of rules and the evaluation time of the tool. Due to the minor influence of this value, it has more flexibility so long as it is large enough to capture a large portion of the programming rules. This typically occurs around values larger than 10, based on empirical results.

3.4.4 Upper Rule Threshold

This is the percent a single sequence or combination of sequences must occur in order to be a rule. This value is used to ensure the rules must occur often enough such that the detected violations occur infrequently. This threshold is important for both single rules and combined rules, for single rules the probability of the rule must simply exceed this threshold. For combined rules, if given a set of sequences, where each one occurs 15%-20% of the time, their sum of probabilities must exceed this threshold. Otherwise, they will not be considered a combined rule. This is to ensure there is enough support to make a claim that a combined rule occurs often enough to be a valid rule.

3.4.5 Lower Rule Threshold

This is the percent a sequence must occur in order to be considered part of a combined rule. It must be smaller than the upper threshold. This value is to ensure a sequence must occur frequently enough to be considered part of a potential combined rule. Having a large number of infrequent sequences does not yield a viable pattern as it would tend to indicate that no rule exists. Raising this value decreases the number of sequences considered to be valid for a combined rule. This is the result of requiring each sequence to occur more often. Setting this value to 15% means if a set of sequences occur 20 times, any which happen at least 3 times are then considered part of a potential combined rule.

3.4.6 Anomaly Threshold

This is the percent limit a sequence can occur in order to be a violation of a rule. Meaning given a known rule, a sequence will only be considered a violation if the percentage it occurs is less than this threshold. This threshold must be smaller than the lower rule threshold. This value is very sensitive to changes, for example keeping all other values at the defaults and changing this value from 5% to 7% would change the number of violations in GeoTools from 80 to 120 yet only increase the number of useful violations by 4. While there is more

potential for finding bugs because of the larger number of results, since the large majority of the results are false positives it simply becomes counterproductive to evaluate all the results as this constraint is relaxed.

3.4.7 Minimum Method Count

This is the minimum number of method calls required to be in a rule. This value is important at ensuring the results focus on semantics. Rules that contain control flow only or call a single method often do not provide enough information to indicate a bug. For example, it is common to check parameters which would show up as a series of if-blocks but since we do not look at the semantics of the conditions being checked it would be nearly impossible to tell if a bug existed in a sequence. This value has a larger impact on smaller n-grams compared to larger n-grams as this is an absolute value and will require smaller n-grams to have a proportionately higher number of method calls.

3.4.8 Maximum Violation Count

This is the maximum number of violations a rule can detect. If the number of violations detected by a rule exceeds this value we invalidate the rule. The rationale behind this threshold is to remove rules which result in numerous violations, which is a strong indicator the rule should not be used.

3.5 Other Rule Pruning

In addition to *NGDetection*'s parameters, other approaches are used to filter rules to improve detection accuracy. Common class's methods, such as Object and String's methods, are filtered because they tend to dominate the rules and prevent other rules from being discovered. The ability to filter out certain classes is setup by a blacklist which can be configured to be project specific. This enables a developer to ignore rules related to a certain classes, methods, or constructors.

A common issue with extracting rules from source code is when repeat sets of methods are being called. This occurs when building collections, setting various configuration parameters, or simply checking null for a number of parameters. These cycles of method calls tend to create rules which trigger a violation when they stop. For instance, if we call

the method put(obj.toString(), obj) on a hashmap with a number of different objects, a rule will emerge which indicates `[put(), toString(), put(), toString()]` should be followed with a call to `put()`. If we instead have finished putting our data into the map and return, this will then be flagged as a violation even though it will inevitably happen. Thus, any cycles we encounter in rules are removed as they likely will not be useful in detecting actual violations.

In order to avoid repeatedly detecting the same violations we remove related subrules. For instance, if we have two rules `BCDE` and `ABCDE`, we only keep `ABCDE` since it is the most specific rule. This filtering approach may remove good programming rules that can potentially detect bugs. In the future, we would like to evaluate the impact of removing this filtering approach.

Chapter 4

Results

This section presents the results of applying *NGDetection* on the 14 evaluated projects as well as the experimental setup. Section 4.1 describes the evaluated machine, projects, and parameters used for the results. There are a number of aspects being evaluated, the first is on inferring programming rules (Section 4.3), second is detecting violations (Section 4.4), third is the sensitivity of experimental parameters (Section 4.5), and finally the execution time of *NGDetection* (Section 4.6).

4.1 Experimental Setup

All the experiments are conducted on a desktop machine with a 4.0GHz i7-3930K 64GB of memory. The JVM arguments used were "-d64 -Xms16G -Xmx16G -Xss4m" these values provide plenty of heap space but 4GB would have been sufficient given the set of projects used. More details about the performance of the tool are in Section 4.6.

The values of the *NGDetection* parameters used:

- `Minimum Gram Size` = 5
- `Maximum Gram Size` = 12
- `Anomaly Threshold` = 5%
- `Lower Rule Threshold` = 15%
- `Upper Rule Threshold` = 80%
- `Minimum Method Count` = 2
- `Maximum Violation Count` = 5

Table 4.1: Evaluated Projects

Project	**Version**	**Files**	**LOC**	**Methods**
Elasticsearch	1.4	3,130	272,261	28,950
GeoTools	13-RC1	9,666	996,800	89,505
Java JDK	1.8.0_31	7,714	1,026,063	87,901
JavaFX	1.8.0_31	2,496	360,021	36,524
JEdit	5.2.0	543	110,744	5,548
ProGuard	5.2	675	69,376	5,919
Vuze	5500	3,514	586,510	37,939
Xalan	2.7.2	907	165,248	8,965
Hadoop	2.6.0	4,307	596,462	46,104
Hbase	1.0.0	1,392	465,456	42,948
Pig	0.14.0	948	121,457	9,323
Solr-core	5.0.0	1,061	146,749	9,938
Lucene	5.0.0	2,065	293,825	18,078
Opennlp	1.5.3	603	36,328	2,954

The parameter values were chosen because they give a reasonable number of results given empirical testing. The *same* set of parameters is used for all 14 evaluated projects, showing some generality of the parameters. It is certainly possible these parameters could be more finely tuned for specific projects in order to maximize the number of detected violations while ensuring the violations are useful. Instead of using specific parameters for each project, the sensitivity of the parameters for GeoTools is shown in Section 4.5 to provide insight into the behavior of the parameters.

4.2 Evaluated Projects

NGDetection is evaluated on 14 widely-used Java projects ranging from 36 KLOC to 1 MLOC. Table 4.1 lists their versions, numbers of files, lines of code, and number of methods. As can be seen, there are a wide variety of projects being evaluated which are of varying sizes and functionality. *NGDetection* may not be effective with all Java projects as there are a number of features required in order to maximize the potential of finding violations. For instance, projects should not be very small (less than 10 KLOC) as projects which are small tend to not have a significant number of patterns in them which reduces the number of rules. Without these patterns *NGDetection* will lack support to make claims about

Table 4.2: Number of Inferred Programming Rules

Project	**Total**	**Single**	**Combined**
Elasticsearch	373,084	366,733	6,351
GeoTools	719,918	706,215	13,703
Java JDK	1,068,611	1,044,473	24,138
JavaFX	391,924	384,319	7,605
JEdit	171,458	168,374	3,084
ProGuard	44,283	43,176	1,107
Vuze	707,637	695,447	12,190
Xalan	138,203	135,267	2,936
Hadoop	920,527	907,380	13,147
Hbase	743,232	735,648	7,584
Pig	162,265	159,320	2,945
Solr-core	239,262	236,843	2,419
Lucene	293,306	289,434	3,872
Opennlp	44,075	43,534	541
Total	**6,017,785**	**5,916,163**	**101,622**

violations and will result in only a few violations or none at all. Another factor, which is related to the number of patterns, is the project needs to have a number of method call sequences which are of reasonable length. What is meant by a reasonable length depends on the number of other control flow structures around the method calls. This is caused by having a minimum size restriction on rules, if a project has a large number of short methods with only a couple of method calls in each of them *NGDetection* will struggle to find usable patterns as they may be excluded by simply being too short.

4.3 Programming Rule Results

Table 4.2 shows the number of programming rules generated by *NGDetection*. It is split into two categories: single rules (such as `ABC`) and combined rules (such as `AB(C|D)`). *NGDetection* infers a total of 6,017,785 programming rules from the 14 evaluated projects, showing our approach can infer a large number of implicit, undocumented programming rules.

Of the discovered rules, 5,916,163 (98.3%) rules inferred in our experiments are single rules and 101,622 (1.7%) are combined rules. While a majority of the rules are single rules,

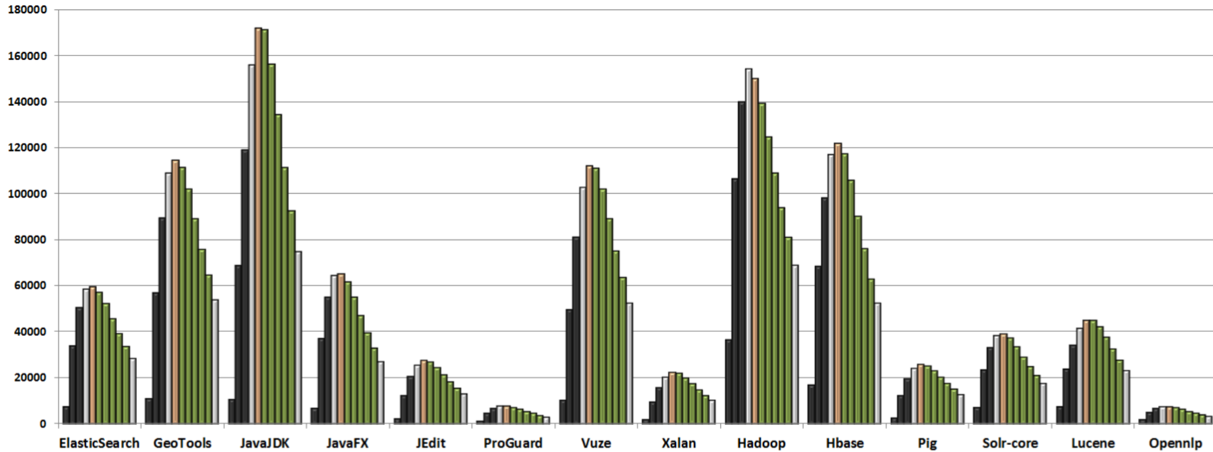

Figure 4.1: Project rule sizes. In each group of bars, the left most bar is the number of rules using bigrams, and the right most using 12-grams. The three left most bars (2,3,4-gram rules) are in gray because they are not used in our results. White indicates the 5-gram and 12-gram boundaries for rule generation. Orange indicates the 6-gram which is typically the maximum.

NGDetection generated a large number of combined rules, which cannot be generated by techniques such as PR-Miner [3]. This strongly suggests that a rules such as `AB(C|D)` are significantly less common than rules such as `EFG`. Despite the small percentage of combined rules, a large percentage (34%) of the detected violations are found by combined rules (Section 4.4). This suggests combined rules are more likely to be unknowingly violated because it might be harder for a developer to realize the more complicated combined rules, and therefore they are more likely to violate them. The results imply these combined rules could be particularly useful because they may help detect harder-to-find bugs.

Figure 4.1 shows the breakdown of the rules by size: the number of bigram to 12-gram rules. Interestingly, the peak number of rules are around 6-grams across all projects, despite the large variation in project sizes, styles, and functionality. While bigrams make up the largest number of n-grams generated, the `Minimum Method Count` parameter which requires a rule to have two method calls eliminates most bigrams, since only the bigrams which call two methods are kept. Requiring two method calls helps reduce false positives.

Table 4.3: Violation detection results. Bugs column is the number of bugs, RefV column is the refactor violations, RefR column is the refactor rules. Useful column is the total number of violations for the three types. Total column is the total number of violations detected. S is the number detected by single rules, C is the number detected by combined rules, and T is the total.

	Bugs			RefV			RefR			Useful			Total		
Project	**S**	**C**	**T**	**S**	**C**	**T**	**S**	**C**	**T**	**S**	**C**	**T**	**S**	**C**	**T**
Elasticsearch	0	0	0	5	3	8	7	1	8	12	4	16	24	13	37
GeoTools	11	5	16	5	2	7	9	0	9	25	7	32	57	23	80
JavaJDK	2	3	5	6	0	6	4	3	7	12	6	18	47	32	79
JavaFX	2	0	2	1	0	1	0	0	0	3	0	3	8	6	14
JEdit	2	0	2	1	0	1	0	0	0	3	0	3	17	6	23
ProGuard	1	0	1	0	0	0	0	0	0	1	0	1	1	0	1
Vuze	0	0	0	1	0	1	5	0	5	6	0	6	17	10	27
Xalan	0	4	4	0	0	0	0	0	0	0	4	4	5	4	9
Hadoop	3	0	3	0	1	1	3	0	3	6	1	7	10	3	13
Hbase	3	0	3	2	0	2	1	0	1	6	0	6	7	0	7
Pig	2	0	2	0	0	0	1	1	2	3	1	4	4	2	6
Solr-core	0	0	0	0	1	1	0	0	0	0	1	1	0	1	1
Lucene	2	1	3	1	0	1	1	0	1	4	1	5	7	3	10
Opennlp	1	1	2	0	0	0	0	0	0	1	1	2	2	1	3
Total	**29**	**14**	**43**	**22**	**7**	**29**	**31**	**5**	**36**	**82**	**26**	**108**	**206**	**104**	**310**

4.4 Violation Detection Results

Table 4.3 shows the number of violations detected per project as well as the category each violation belongs to. *NGDetection* detected 310 violations in total across the 14 projects. These violations are categorized into four types: *Bugs*, *Refactor Violations*, *Refactor Rules*, and False Positives. Bugs are when the violation should be fixed by altering the code and correcting its behaviour to match the rules. Refactor Violations are when the violation can be fixed by refactoring the violation's code to make it consistent with the rules. Refactor Rules are when the violation and the rule should both be refactored in order to eliminate the violation. This removes the rule and corrects the violation. Any violations that do not fit into the above three groups are considered to be false positives.

In total, *NGDetection* detected 108 useful violations—43 bugs, 29 refactor violations, and 36 refactor rules. 32 of the 43 bugs were reported to the developers and the rest are in the process of reported. Of the reported bugs, 2 of which have been confirmed by developers, while the rest await confirmation. As can be seen, *NGDetection* detected a number of violations across many different projects of various sizes; however, smaller

projects tend to have fewer violations than larger projects.

4.4.1 Impact of Combined Rules

Although the combined rules only make up 1.7% of all rules, they helped detect a larger portion of true violations, e.g., 14 out of the 43 bugs, and 7 out of the 29 refactor violations. Specifically, 206 (66%) detected violations are from single rules and 104 (34%) are from combined rules. Among these violations, 82 (76%) useful violations were detected using single rules and 26 (24%) using combined rules, which cannot be detected by techniques such as PR-Miner [3] that infer single rules only. The results show on average every 72,148 single rules detects 1 useful violation, but every 3,909 combined rules detects 1 useful violation, suggesting that combined rules provide more meaningful programming rules. This is likely the result of them being less clear to developers since combined rules allow developers to use different styles and procedures, or start methods with common code but then split into different paths for special cases. As discussed earlier, these combined rules could be useful in detecting potentially hard-to-find bugs.

4.4.2 False Positives

The false positive rates of *NGDetection* are comparable to existing techniques [3,6,7,20]. A main reason for false positives is the inferred rules do not always need to be followed, and the false violations tend to deal with special cases. For example, a developer of a project may choose to check if parameters being passed into a method are null. However, they may have additionally checks for certain parameters in different methods such as checking to see if a given list is empty or a string has a minimum length. These deviations from just performing null checks will show up as violations because they are not following the pattern of normally only checking for a null value. These deviations are not violations since they are simply applying additional checks depending on actions the method will perform. Another example is when if-else chains are used to split logic into various sequences depending on the state of the parameters or fields. Different objects will contain if-else chains which check for various aspects of a value such as: is it in a range, in a collection, null, or equal to another value. These similar if-else chains can result in rules being defined across many different classes, which results in violations for a class that do not perform one of these checks. Again, these deviations may not be violations because certain fields or parameters being checked in some classes may not even exist in a completely unrelated class. In the future, it would be interesting to leverage this information to see if these cases can be filtered to

reduce false positives. JEdit has a large number of false positives, this is a result of the violations being found in generated code where *NGDetection* is inappropriate to apply. This is because the code is not written by developers and therefore is not representative of developer programming rules and refactoring the code is not helpful. While this is not true for a majority of projects, other projects such as Xalan, also have generated code which results in violations being detected.

4.4.3 Example Bugs

Figure 4.2 shows an example of a **Bug** detected by our tool from the Elasticsearch project. This bug was reported to the Elasticsearch developers, who have confirmed this is a true bug. In the violation the `debug()` log method is called instead of the `warn()` log method which is normally called in this situation. This bug was found with a combined rule: `[begin_method, try, sendResponse(), catch, (warn() | onFailure())]`. The fix involves the replacing the `debug()` log method with the `warn()` log method.

Figure 4.3 shows another example **Bug** detected by our tool from the GeoTools project. In this example the code segments are related to closing streams and normally a common method in the GeoPackage class which is called on a stream when it should be closed. An instance where this method was not used was found and in the code above the finally blocks is a TODO comment indicating it should be changed. The pattern was to call the private static method in the current class which has a null check, closes the stream and logs a warning message if an error occurs. However, by directly calling the close() method on the BufferedInputStream there is no null check and no message will be logged when an error occurs.

Example Refactor Violations

Figure 4.4 shows an example of **Refactor Violation** detected by our tool from the Elasticsearch project. This was detected by a combined rule. The first option of the combined rule (`isValue()`) was to call the `isValue()` method on the parsed token to determine if a value is expected to be parsed. The second option (`if`) was to put an if-else chain inside of the else block and then call the equals method on the fieldName. The third option (`if_then`) was to do a check to see if the token is a special type of value. The code segments are related to the parsing sections of Elasticsearch. For simplicity only the code segment for the second option (`if`) is shown, which is most closely related to the refactoring of the violation.

Combined Rule:
[<begin_method>, try, sendResponse(), catch, (warn() | onFailure())]
Example source code that follows the rule:

```
1 @Override
2 public void onFailure(Throwable e) {
3     try {
4         channel.sendResponse(e);
5     } catch (Exception e1) {
6         logger.warn("failed to send response for get", e1);
7     }
8 }
```

Violation pattern:
[<begin_method>, try, sendResponse(), catch, error()]
Source code that violates the combined rule:

```
1 @Override
2 public void onNewClusterStateFailed(Throwable t) {
3     try {
4         channel.sendResponse(t);
5     } catch (Throwable e) {
6         logger.debug("failed to send response on cluster state processed", e);
7     }
8 }
```

Figure 4.2: A combined rule inferred for Elasticsearch and a true bug detected using this rule. The bug has been confirmed by Elasticsearch developers after it was reported.

The detected violation differs from the second option by directly continuing the if-else chain and by calling the `equals()` method to see if the token belongs to a field name. The (`else`) and (`else if`) are both represented as (`else_if`) the difference is the (`else if`) will have an (`if_then`) token following it whereas the (`else`) will not since it has no conditional statement. This is not a bug but a deviation from the 37 other rule patterns which do not do this. This is why it is called a refactor violation, because the program semantics of the violation should not be changed, but the code should be changed to follow the expected pattern to allow for the code to match the style used by the project.

Rule: [finally, GeoPackage.close(), end_try, finally, GeoPackage.close()]
Example source code that follows the rule:

```
1     finally {
2         close(ps);
3     }
4 }
5 finally {
6     close(cx);
7 }
```

Violation pattern: [finally, GeoPackage.close(), end_try, finally, BufferedInput-Stream.close()]
Source code that violates the rule:

```
1     finally {
2         close(ps);
3     }
4 }
5 finally {
6     bin.close();
7 }
```

Figure 4.3: A rule inferred for GeoTools and a bug detected using this rule.

4.4.4 Example Refactor Rules

Figure 4.5 shows an example of **Refactor Rule** detected by our tool from the Vuze project. In this example, the code segments are related to opening an interface element. The pattern is to normally call `openView()` on the `UIFunctions` object inside the `selected()` method of a `MenuItemListener` anonymous class. The detected violation differs from the pattern by directly performing the call inside of a loop. This is not a bug but a deviation from the 25 other rule patterns which do not do this, but it does not make sense in this instance to create a `MenuItemListener` since the intent is to directly call `openView()`. This is why it is called a refactor rule, because the program semantics should not be changed, but the code for both the violation and the rule should be refactored into common method as they are clones of each other. The refactor involves taking the common code (lines 1 - 4) and extracting it to a common method where the parameters to the `openView()` method are passed into the new common code. By moving the code into a common method

Rule: [if_then, while, nextToken(), if, if_then, currentName(), if_else, (isValue() | if | if_then)]
Example source code that follows the rule:

```
} else if (token == XContentParser.Token.START_OBJECT) {
    String currentFieldName = null;
    while ((token = parser.nextToken()) != XContentParser.Token.END_OBJECT) {
        if (token == XContentParser.Token.FIELD_NAME) {
            currentFieldName = parser.currentName();
        } else {
            if ("value".equals( currentFieldName) || "_value".equals(
                currentFieldName)) {
```

Violation Pattern: [if_then, while, nextToken(), if, if_then, currentName(), if_else, equals()]
Source code that violates the combined rule:

```
} else if ("collate".equals(fieldName)) {
    while ((token = parser.nextToken()) != XContentParser.Token.END_OBJECT) {
        if (token == XContentParser.Token.FIELD_NAME) {
            fieldName = parser.currentName();
        } else if ("query".equals(fieldName) || "filter".equals(fieldName)) {
```

Refactored code for the violating code segment:

```
...
else {
    if ("query".equals(fieldName) ||
        "filter".equals(fieldName)) {
    ...
}
```

Figure 4.4: A rule inferred for Elasticsearch, a Refactor Violation detected using this rule, and the suggested refactored code for the violation.

the violation will be removed and if a developer decides to change how this operation is performed they would only have to update it in one location.

Rule: [getUIFunctions(), if, if_then, openView(), end_if, end_method]
Example source code that follows the rule:

```
1     UIFunctions uif = UIFunctionsManager.getUIFunctions();
2     if ( uif != null ){
3         uif.openView( UIFunctions.VIEW_CONFIG, "Pairing" );
4     }
5 }
```

Violation Pattern: [getUIFunctions(), if, if_then, openView(), end_if, break]
Source code that violates the combined rule:

```
1 UIFunctions uiFunctions = UIFunctionsManager.getUIFunctions();
2 if (uiFunctions != null) {
3     uiFunctions.openView( UIFunctions.VIEW_DM_DETAILS, dm);
4 }
5 break;
```

Figure 4.5: A rule inferred for Vuze and a Refactor Rule detected using this rule.

4.4.5 Comparing With Existing Techniques

Existing techniques such as PR-Miner [3] and DynaMine [5] cannot automatically detect many of the useful violations detected by *NGDetection*. These techniques cannot detect the violations detected by combined rules, which combine multiple infrequent patterns together as a rule. Of the 108 useful violations, 26 (24%) were found using combined rules. In addition, some of the single rule violations cannot be found by other techniques. For instance, the Refactor Rule example in the previous section which requires analysis of keywords and control flow was found by the technique and cannot be found by examining method calls alone or control flow sub-graphs [6].

In addition, PR-Miner [3] analyzes method calls and variables, thus fails to take into consideration the control flow surrounding the methods and therefore cannot indicate if a `close()` method is called outside or inside of a `finally` block. Control flow related violations are part of 40 out of 108 useful violations detected by *NGDetection*. In addition, the frequent itemset mining approach used does not consider repeat method calls or ordering information which means it cannot support situations where a method is called, the result is checked, and if the value is null, then a method is called again. Repeated method calls are part of 16 out of 108 useful violations detected. Our method compliments

Table 4.4: Parameter Sensitivity of *NGDetection* on GeoTools

Parameter Changed	**Value**	**Rules**	**Bugs**	**RefV**	**RefR**	**Useful**	**Total**	**FP** %
Default Values	N/A	719,918	16	7	9	32	80	60
Minimum Gram Size	4	809,296	21	10	9	40	122	66
Minimum Gram Size	6	610,871	8	5	8	21	48	56
Maximum Gram Size	11	666,386	16	7	9	32	80	60
Maximum Gram Size	13	751,465	16	7	9	32	80	60
Upper Rule Threshold	70%	720,913	16	8	9	33	104	68
Upper Rule Threshold	90%	719,053	9	5	8	22	48	54
Lower Rule Threshold	10%	720,438	16	9	9	34	91	63
Lower Rule Threshold	20%	719,551	12	6	9	27	69	61
Anomaly Threshold	3%	719,918	5	6	4	15	37	59
Anomaly Threshold	7%	719,918	16	11	9	36	120	70
Minimum Method Count	1	788,791	18	12	14	44	134	67
Minimum Method Count	3	580,447	2	4	5	11	32	66
Maximum Violation Count	1	719,918	5	3	5	13	32	59
Maximum Violation Count	5	719,918	16	8	12	36	121	70

PR-Miner [3], as both approaches are able to detect different violations. For instance, our approach is unable to handle long distance relationships within the same method due to the nature of n-gram; however, frequent itemset mining does not have this limitation.

DynaMine [5] uses association rule mining and does not utilize combined rules, thus missing the 26 useful violations. The types of patterns mined are either method pairs, state machines on a single object, complex rules involving multiple objects. DynaMine's complex patterns require user specification of patterns, while *NGDetection* is able to automatically infer a wide variety of complex patterns involving methods or control flow.

4.5 Parameter Sensitivity

In order to clearly show the sensitivity of the seven parameters described in Section 3.4.1 one project is presented with various values. Each parameter can either be set to be more restrictive (resulting in fewer rules and detected violations) or less restrictive (resulting in more rules and detected violations). For each of the seven parameters, one value was selected and tested with a value that is more restrictive and one value that is less restrictive for this experiment. GeoTools was selected because it is a medium sized project which has detected violations for each category.

Table 4.4 shows how changing the seven parameters of *NGDetection* influences the rules and detection results. The "Value" column indicates the new value for the parameter

to generate the set of results along that row. The "Rules" column indicates the new number of rules generated. The "Bugs", "RefV", and "RefR" columns indicate the number of bugs, refactor violations and refactor rules for the given parameter. The "Useful" column is the sum of all three types of violations, and the "Total" column indicates the total number of detected violations. The "FP" column shows the percentage of detected violations which were false positives to show how the results are influenced by the parameters. The results show that adjusting the parameters can result in noticeable changes to the quality of the rules and detection results. For example, the total number of detected violations ranges from 32 to 134, and the false positive rates range from 54% to 70%.

4.5.1 N-Gram Size Limits

Specifically, reducing the **Minimum Gram Size** to 4 increases the number of rules generated; however, these rules help detect an additional 42 violations but only 8 are useful violations. This results in a 6 percentage point increase (from 60% to 66%) in false positives. While using 4-grams does yield more rules and violations, there is a loss in the accuracy of the results. Increasing **Minimum Gram Size** to 6 it can be seen that the rules drops noticeably as well as the number of violations, the false positives also decreases slightly. Changing the **Maximum Gram Size** to 11 or 13 makes a noticeable difference to the number of generated rules but makes no difference to the detected violations. This is because most violations are detected by the smaller gram sizes, since there are fewer larger gram rules with enough support to detect violations. Empirically, increasing the maximum gram size beyond 12 only increases the run time of the tool with minimal benefit to the results.

4.5.2 Rule Threshold Limits

The **Upper Rule Threshold** has a large influence on the results. Decreasing it to 70% makes it less restrictive and results in more false positives. Increasing it to 90% makes it more restrictive and reduces the violations significantly but also reduces the number of false positives by 5 percentage points. The **Lower Rule Threshold** only influences combined rules. Decreasing it to 10% makes it less restrictive and finds a few more violations. Increasing it to 20% makes it more restrictive and reduces the number of violations. Decreasing the **Anomaly Threshold** to 3% makes it more restrictive and produces significantly fewer violations. Increasing it to 7% makes it less restrictive and produces more violations which are mostly false positives.

Table 4.5: Execution Time in Seconds

Project	**Total**	**Tokenization**	**Model Building and Violation Detection**
Elasticsearch	310	283	27
GeoTools	641	590	51
Java JDK	798	728	70
JavaFX	261	236	25
JEdit	65	56	9
ProGuard	65	63	2
Vuze	385	338	47
Xalan	90	83	7
Hadoop	516	454	62
Hbase	201	152	49
Pig	110	99	11
Solr-core	128	113	15
Lucene	230	212	18
Opennlp	54	52	2

4.5.3 Count-based Limits

Decreasing the **Minimum Method Count** to 1 makes it less restrictive and introduces a large number of rules and 12 more violations. It also increases the false positive rate by 7 percentage points. Increasing the value to 3 makes it more restrictive and reduces the number of rules and violations. The **Maximum Violation Count** also has significant influence on results. Decreasing it to 1 makes it more restrictive and reduces the number of detected violations to 32 without changing the false positive rate. Increasing it to 5 has a large increase in the detected violations and the false positive rate.

In summary, these tunable parameters can change the results, i.e., more or less rules or violations, but usually at a cost. The default values used in this thesis provide a reasonable result set for all evaluated projects. While it is easy to tune parameters for a specific project, the default values presented are a useful starting point for more fine-grained tuning for a project.

4.6 Execution Time and Space

Table 4.5 shows the total execution time, the time spent during the tokenization stage (parsing and building the ASTs), time spent building the n-gram model and determining the violations for a project. The table shows the total time varies from 54 to 798 seconds depending on the size of the project. The VM arguments are generous for heap memory allocation (16GB) since our largest evaluated project, Java JDK, only uses 3GB of memory. The results suggest the tool is efficient enough to be used in practice.

Most time is spent building ASTs with type information with a fraction of the time on building the rules and finding violations. The timing results attempt to ignore time taken for garbage collection to run as a gc is requested after each stage. The results suggest the performance scales linearly with the project's LOC, thus our approach is expected to scale to large projects. The implementation is designed so the tokenization stage can be cached allowing for the model building and violation detection stage to be run with different parameters for a fraction of the time. This enables a quick and iterative approach to tuning parameters for a specific project if desired.

Chapter 5

Conclusion and Future Work

This thesis described *NGDetection*, an approach which models source code using the n-gram langauge model and extracts programming rules for detecting bugs as well as refactoring opportunities. In addition to the n-gram language model, it utilizes control flow elements and combined rules to detect violations to 6,017,785 programming rules across 14 projects. Using the approach, it was able to identify 108 useful violations out of 310 total violations. Of those useful violations at least 26 of them in the 14 evaluated projects cannot be found automatically by existing approaches. While *NGDetection* uses the n-gram language model, which is one of the simpler language models, there are other language models which may be able to achieve a different set of results. Additionally, there are many different ways which source code elements can be translated into tokens which would also yield a different set of results. Future work may consider looking into different language models and other ways of translating source code into tokens.

As with all approaches there are limitations to their performance, *NGDetection* is no exception to this. Below is a list of the limitations to the current approach and some details on how these limitations could be addressed.

- **Simply knowing a violation has occurred is often not enough**. The current approach requires manual verification for each reported violation. This leads to situations where a violation has been detected but is a perfectly acceptable code sequence. Some of these types of violations are detailed in Section 4.4.2, essentially developers may deviate from the anticipated pattern to handle special cases or different object types. While *NGDetection* is correctly identifying violations, these violations are not useful and refactoring may not be appropriate as it would lead to more complicated code. Filtering these violations automatically is very challenging without the

tool being able to fully comprehend the semantics of the code, something the n-gram model is simply not capable of handling.

- **Detected rules are highly localized**. One known limitation of the n-gram language model is its inability to identify long distance relationships. An attempt to alleviate this limitation in the n-gram model was to use skip-grams [33], as the name suggests it skips elements in the sequences. For instance a given sequence `ABCDEF` would be translated into the following set of 1-skip-2-grams `AC`, `BD`, `CE`, `DF`, combined with the set of bigrams. This may lead to a larger set of rules, but it removes some of the semantic information and would not be appropriate to couple with control flow information.

- **Patterns limited by method boundaries**. Related to the lack of long distance relationships, the approach only analyzes patterns inside of a method body. This is a limitation of static analysis, but some issues could be resolved by inlining method calls in a method body. However, this solution has limitations when dealing with abstract classes and interfaces where the invocation is ambiguous. Other issues with this limitation is handling calls to and from native code with Java, reflexive invocations or instantiations, or networking done using remote procedure calls. Some of these issues could be addressed by using dynamic analysis or a hybrid approach of mixing the two, such as what DynaMine [5] uses.

- **Choice of tokens to be used during tokenization**. Knowing which tokens to use is only one part of deciding what should be considered a token. This is important as these tokens are the input to the n-gram model. *NGDetection* uses method calls, constructors, and control flow elements; however, even these elements could be encoded in various ways. For instance, do you:

  - consider fully qualified names or simple names?
  - consider the scope when dealing with control flow or do you simply extract the keyword locations?
  - treat foreach loops like for loops?
  - provide special treatment for rarely occurring loops like do-while?
  - treat switch statements differently than if-else chains?
  - consider return types or treat them all equally? What if returning null or 0?
  - consider the parts of the condition expression for if statements?

  - treat each part of the condition expression as a token ([<IF>, <EQ>, <NULL>, <AND>, <NEQ>, <NULL>, <IF_THEN>]), or do you merge the entire expression into a single token ([<IF>, <EQ_NULL_AND_NEQ_NULL>, <IF_THEN>])?
  - analyze the exception types in the catch blocks or treat catch blocks equally?

  Each one of these questions was considered in the design, the final design was outlined in the approach chapter (Chapter 3), but as can be seen there are a large number of possible options to consider and each can have a large impact on the set of results.

- **N-gram model focuses on the last element in a sequence**. While frequent itemset mining is able to identify when an element is missing from the start of a method, the n-gram model does not consider this since it looks at the previous n-elements to find patterns. If one element is missing from the previous n-elements then the violation will not be found. Example, say the sequence ABCDE occurs 100 times and ABDE occurs once, *NGDetection* will not detect ABDE as a violation. More importantly the probabilities in the model are calculated based on the value of the nth element. Thus the model is limited to focusing on when the nth element does not match the pattern and not when one of the preceding elements is replaced with an unusual element. Example, say the sequence ABCDE occurs 100 times and ABFDE occurs once, *NGDetection* will not detect ABFDE as a violation. There are potential solutions to this problem, such as building the n-gram model different ways:

  - For example, given the sequence ABC instead of the n-gram model finding the probability of C if the previous two tokens were AB we could instead reverse it and ask, what is the probability of A if it is followed by BC?
  - Another way to look at it is to consider the tokens which occur before and after the token of interest. For example, given the sequence ABC what is the probability of B if it is preceded by A and followed by C?

  Naturally, combinations of these types of rules can be considered; however, they would increase the memory requirements significantly and is closer to a different approach known as frequent sequence mining [34].

- **Project size plays a dominant role**. By comparing the project sizes with the number of detected violations it can be clearly seen there is a correlation between the two values. Larger projects tend to have more violations detected, this is expected because frequency as well as probability of programming rules plays a role in determining rules. A side effect of this is that smaller projects tend to have few or no

violations being detected. This can be alleviated by using more relaxed parameters to make the thresholds more forgiving; however, this does not ensure the detected violations are of high quality.

References

[1] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[2] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th international conference on Software engineering*, pp. 291–301, ACM, 2002.

[3] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 306–315, ACM, 2005.

[4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, *Bugs as deviant behavior: A general approach to inferring errors in systems code*, vol. 35. ACM, 2001.

[5] L. Benjamin and T. Zimmermann, "Dynamine: Finding common error patternsby mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 296–305, ACM, 2005.

[6] R.-Y. mining, A. Podgurski, and J. Yang, "Finding what's not there: A new approach to revealing neglected conditions in software," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 163–173, ACM, 2007.

[7] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for defect detection," vol. 18, pp. 292–323, Springer, 2011.

[8] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 25–34, ACM, 2007.

[9] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 54–57, ACM, 2006.

[10] B. Sun, G. Shu, A. Podgurski, and B. Robinson, "Extending static analysis by mining project-specific rules," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 1054–1063, IEEE, 2012.

[11] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 496–506, IEEE, 2009.

[12] C. C. Williams and J. K. Hollingsworth, "Recovering system specific rules from software repositories," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pp. 1–5, ACM, 2005.

[13] H. Kagdi, M. L. Collard, and J. I. Maletic, "An approach to mining call-usage patternswith syntactic context," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pp. 457–460, ACM, 2007.

[14] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 145–158, ACM, 2007.

[15] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language api documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 307–318, IEEE Computer Society, 2009.

[16] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou, "MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pp. 103–116, 2007.

[17] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 837–847, IEEE, 2012.

[18] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, p. 44, ACM, 2014.

[19] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pp. 332–343, IEEE, 2009.

[20] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 532–542, ACM, 2013.

[21] E. Charniak, "Statistical language learning," in *First MIT Press paperback edition*, MIT Press, 1996.

[22] L. R. Bahl, P. Brown, P. V. de Souza, and R. Mercer, "A tree-based statistical language model for natural language speech recognition," vol. 37, pp. 1001–1008, IEEE, 1989.

[23] R. Rosenfield, "Two decades of statistical language modeling: Where do we go from here?," 2000.

[24] M. Allamanis, E. T. Barr, and C. Sutton, "Learning natural coding conventions," *arXiv preprint arXiv:1402.4182*, 2014.

[25] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal api rules from imperfect traces," in *Proceedings of the 28th International Conference on Software Engineering*, pp. 282–291, ACM, 2006.

[26] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," vol. 69, pp. 35–45, Elsevier, 2007.

[27] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 383–392, ACM, 2009.

[28] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 174–184, ACM, 2007.

[29] C. Williams and J. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," vol. 31, pp. 466–480, 2005.

[30] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," vol. 32, pp. 176–192, IEEE, 2006.

[31] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *Proceedings of the 29th international conference on Software Engineering*, pp. 240–250, IEEE, 2007.

[32] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 337–348, ACM, 2014.

[33] D. Guthrie, B. Allison, W. Liu, L. Guthrie, and Y. Wilks, "A closer look at skip-gram modelling," in *Proceedings of the 5th international Conference on Language Resources and Evaluation (LREC-2006)*, pp. 1–4, 2006.

[34] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Data Engineering, 1995. Proceedings of the Eleventh International Conference on*, pp. 3–14, IEEE, 1995.

[35] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: lightweight cross-project anomaly detection," in *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 119–130, ACM, 2010.

[36] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[37] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pp. 207–216, IEEE, 2013.

[38] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 35–44, ACM, 2007.

[39] C. Lindig, "Mining patterns and violations using concept analysis," Citeseer, 2007.

[40] W. Weimer and G. C. Necula, "Mining temporal specifications for error detection," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 461–476, Springer, 2005.

[41] A. Wasylkowski and A. Zeller, “Mining temporal specifications from object usage,” vol. 18, pp. 263–292, Springer, 2011.