Bugram: Bug Detection with N-gram Language Models

Song Wang*, Devin Chollak*, Dana Movshovitz-Attias†, Lin Tan*
*Electrical and Computer Engineering, University of Waterloo, Canada
†Computer Science Department, Carnegie Mellon University, USA
*{song.wang, dchollak, lintan}@uwaterloo.ca, †dma@cs.cmu.edu

ABSTRACT

To improve software reliability, many rule-based techniques have been proposed to infer programming rules and detect violations of these rules as bugs. These rule-based approaches often rely on the highly frequent appearances of certain patterns in a project to infer rules. It is known that if a pattern does not appear frequently enough, rules are not learned, thus missing many bugs.

In this paper, we propose a new approach—*Bugram*—that leverages n-gram language models instead of rules to detect bugs. Bugram models program tokens sequentially, using the n-gram language model. Token sequences from the program are then assessed according to their probability in the learned model, and low probability sequences are marked as potential bugs. The assumption is that low probability token sequences in a program are unusual, which may indicate bugs, bad practices, or unusual/special uses of code of which developers may want to be aware.

We evaluate Bugram in two ways. First, we apply Bugram on the latest versions of 16 open source Java projects. Results show that Bugram detects 59 bugs, 42 of which are manually verified as correct, 25 of which are true bugs and 17 are code snippets that should be refactored. Among the 25 true bugs, 23 cannot be detected by PR-Miner. We have reported these bugs to developers, 7 of which have already been confirmed by developers (4 of them have already been fixed), while the rest await confirmation. Second, we further compare Bugram with three additional graph- and rule-based bug detection tools, i.e., JADET, Tikanga, and GrouMiner. We apply Bugram on 14 Java projects evaluated in these three studies. Bugram detects 21 true bugs, at least 10 of which cannot be detected by these three tools. Our results suggest that Bugram is complementary to existing rule-based bug detection approaches.

CCS Concepts

•Software and its engineering → Automated static analysis; Software testing and debugging;

Keywords

Bug Detection, Static Code Analysis, N-gram Language Model

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970341

1. INTRODUCTION

Software bug detection techniques have been shown to improve software reliability by finding previously unknown bugs in mature software projects [13, 16]. Rule-based bug detection approaches infer likely programming rules from source code [1, 4, 6, 8, 23, 24, 42, 45, 46, 54], version histories [4, 19, 53], and source code comments [43,44]. These approaches detect violations of these rules as potential bugs.

Frequent itemset mining techniques were used to mine rules that capture the co-occurrence of methods and variables. Violations of these rules are reported as bugs [4, 6, 23]. Along this line, more complex graph models are combined with frequent itemset mining techniques, which focus on mining programming rules that capture both method order and control flow information to detect violations of these complex rules [10, 32, 49, 50].

Let ABC denote a sequence of calls to the methods A, B, and C. Imagine a contrived program that includes 98 occurrences of the sequence ABC, two occurrences of ABD, and a single occurrence of EFG. Existing rule-based bug detection approaches, such as PR-Miner [23], JADET [50], Tikanga [49], and GrouMiner [32] infer rules based on the *conditional probabilities* of method calls, for example, the conditional probability $P(C|AB)$ which denotes the likelihood of seeing a call to method C after the sequence of calls AB. In our example, $P(C|AB) = \frac{98}{98+2} = 98\%$. This probability is higher than the threshold used by PR-Miner, which is 90%, and therefore, the potential rule that C should appear after AB, denoted as $\{AB => C\}$, is selected as a high-probability rule. The *confidence* of this rule is its conditional probability, which is 98%. Given this rule, the sequence ABD is flagged as a bug, because C instead of D is expected to follow AB.

It has recently been demonstrated that *n-gram language models* [7] can capture the regularities of software source code [15,36]. To take advantage of the **n-gram language model**, which provides us with a Markov model for tokens, we propose an n-gram language model based bug detection technique, called *Bugram*. The assumption is that *low probability token sequences in a program are unusual, which may indicate bugs, bad practices, or unusual/special uses of code of which developers may want to be aware*.

While existing studies leverage n-grams for detecting clone bugs [17], localizing faults [30,57], and code search [20], including some that use the term n-gram models [17, 20], these studies do not leverage n-gram models. Instead, they use *n-grams*, which are token sequences, while *n-gram models* are Markov models built on n-grams. On the other hand, n-gram models have been used for code completion and suggestion [12, 31, 37], fault localization [5], and coding style checking [2, 14]. The focus of this paper is leveraging n-gram models for bug detection, which has its own challenges and requires a different design, as detailed in Section 3.

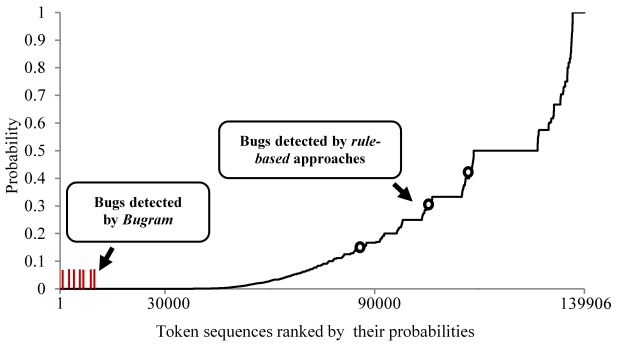


Figure 1: **Bugs detected by Bugram versus bugs detected by rule-based techniques in the latest version of Hadoop**

Instead of using *conditional probabilities*, Bugram highlights suspicious call sequences based on their *absolute probabilities* in the program. So in the example above, our approach evaluates the absolute probability $P(ABC)$ of the full sequence ABC, which is in contrast to PR-Miner which evaluates the conditional probability of $P(C|AB)$. The two sequences with the lowest probabilities in the program are EFG and ABD, which have a probability that is markedly lower than that of ABC. By selecting sequences with low absolute probabilities, Bugram is able to recognize that ABD and EFG are both suspicious sequences. Notably, EFG is not recognized as a suspicious sequence by PR-Miner, despite having only a single occurrence in the program, because there are no rules with high confidence related to EFG. In addition, it is known that even if a rule exists, but the rule pattern does not appear frequently enough, the rule cannot be learned, thus missing many bugs [8,23].

More broadly, rule-based approaches detect bugs from common program patterns, while Bugram detects sequences which are overall uncommon in the program. These two approaches target different types of program abnormalities, and will ultimately detect different types of bugs, as illustrated in Figure 1. The curve shows the probabilities of sequences in the Java project Hadoop sorted ascendingly. The bars depict examples of bugs that can be detected by our n-gram-based approaches, while circles represent examples of bugs that can be detected by rule-based approaches.

In this paper, we study whether our n-gram-based approach can detect bugs in real-world software that rule-based approaches cannot find. In addition, we study whether Bugram is more *precise* than rule-based approaches, i.e., whether Bugram reports a smaller portion of false bugs than rule-based approaches.

1.1 A Motivating Example

Existing rule-based techniques detect potential bugs by using mined rules with enough confidence and *support* (the number of occurrences) to avoid generating a large number of false bugs. For example, PR-Miner [23] requires the confidence of a method call sequence to be over 90% and the support larger than 15 to be identified as a rule, missing opportunities to detect many bugs. For example, Figure 2 shows a real bug detected by our tool in the latest version of Pig, which has already been confirmed by Pig developers. The code snippet in Figure 2(a) contains a bug: for the purpose of logging, the code snippet should convert the object `value` to a string by calling the `toString` method, but it does not. The method call sequence of the buggy code snippet is `[isDebugEnabled, debug, indent, stringify]`, which appears only once in the program. A similar but correct code snippet with a method call sequence `[isDebugEnabled, debug, indent, toString]` appears three times. One of the appearances is shown in Figure 2(b).

(a) **Method call sequence from a buggy code snippet (appears once):** `[isDebugEnabled(), debug(), indent(), stringify()]`

```
1 if (LOG.isDebugEnabled()) {
2     LOG.debug(indent(depth)+"converting from
3         Pig " + pigType + " " + value +
4         " using " + stringify(schema));
5 }
```

(b) **A similar but correct method call sequence (appears three times):** `[isDebugEnabled(), debug(), indent(), toString())]`

```
1 if (LOG.isDebugEnabled()) {
2     LOG.debug(indent(depth)+"converting from
3         Pig " + pigType + " " +
4             toString(value) +
5         " using " + stringify(schema));
6 }
```

Figure 2: **A motivating example from the latest version 0.15.0 of the project Pig. Bugram automatically detected a real bug in (a), which has been *confirmed and fixed* by Pig developers after we reported it.**

Using rule-based bug detection approaches such as PR-Miner, a potential rule that may detect this bug is `[isDebugEnabled, debug, indent => toString]`. Since PR-Miner groups method calls into a set, which means it ignores the order of method calls for example, all other rules that can potentially detect this bug are `[debug, indent => toString]`, `[isDebugEnabled, indent => toString]`, `[indent => toString]`, `[debug => toString]`, `[isDebugEnabled, debug => toString]`, and `[isDebugEnabled => toString]`. The confidence of each potential rule is only 75%, considering only these four code snippets. If we consider the entire Pig project, the confidences of these rules are even lower, ranging from 19.5% to 28.8%. Since PR-Miner requires rules to have confidences at least 90% (to avoid detecting too many false bugs), it filters out all these potential rules, thus missing this real bug. While it is possible to reduce the confidence requirement, the bug detection precision will likely be too low given that already 40–86% of bugs reported by PR-Miner are false bugs with the confidence 90% [23].

Different from these rule-based bug detection approaches, Bugram does not use programming rules. Instead, it detects potential bugs by reporting method call sequences of low probabilities in a project. We detect the real bug shown in Figure 2, because the method call sequence `[isDebugEnabled, debug, indent, stringify]` has a low probability of 2.855×10^{-5}, which is the 13th lowest probability of all 31,204 sequences of length five using a 3-gram model.

1.2 Contributions

In this paper, we propose Bugram to leverage n-gram models to detect real-world software bugs. Experimental results show that Bugram complements existing rule-based bug detection approaches. The contributions of this work are:

- We propose a new bug detection approach, called Bugram, that leverages n-gram models. Bugram learns a probability distribution of method call sequences with control flow and uses the probability distribution to detect bugs.
- We evaluate Bugram on the latest versions of 16 open source projects. The results show that Bugram detects 59 bugs, 42 of which are manually verified as correct, 25 of which are

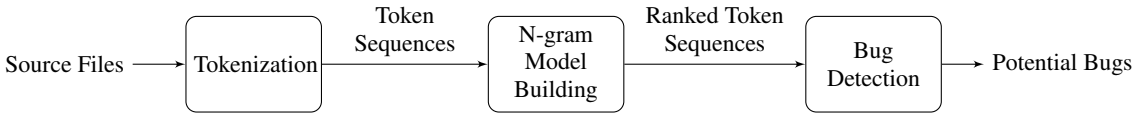


Figure 3: **Overview of Bugram**

true bugs and 17 are code snippets that should be refactored. Among the 25 true bugs, 23 cannot be detected by PR-Miner. The detection precision is 71.2%, higher than PR-Miner and other related work. We have reported these bugs to developers, 7 have been confirmed by developers (4 of them have already been fixed), while the rest await confirmation.

- We compare Bugram with three additional graph- and rule-based techniques, i.e., JADET, Tikanga, and GrouMiner. Specifically, we apply Bugram on the 14 projects evaluated by the three studies. Bugram detects 21 true bugs, at least 10 of which cannot be detected by these three tools. Since the 14 projects are not the latest versions, we did not report the bugs to developers. Instead, we have checked that 16 of the 21 true bugs have already been fixed in a later version. Since JADET is publicly available, we have also applied JADET on the 16 projects evaluated above. JADET reported 0 true bugs, while Bugram detected 25 true bugs. Our results suggest that Bugram is complementary to existing rule-based bug detection approaches.
- Our results show that 3-gram models are the most effective for bug detection among all n-gram models with gram sizes from two to ten.

2. BACKGROUND

The n-gram language model has been widely used in modelling natural language [7] and solving problems such as speech recognition [3], statistical machine translation, and other related language problems [38]. The n-gram language model typically has two components, words and sentences, where each sentence is an ordered sequence of words. A dictionary D contains all possible words of a language, and each word is represented as w. The language model can build a probabilistic distribution over all possible sentences in a language using Markov chains. The probability of a sentence in a language is estimated by generating the sequence word by word. The probability of each word in a sentence is only determined by the conditional probabilities of the previous $n-1$ tokens. Given a sentence $s = w_1 w_2 w_3 \cdots w_m$, its probability is estimated as:

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

where the sequence $h_i = w_{i-n} \cdots w_i$ is the history. In the n-gram model, the probability of the next word w_i depends only on the previous $n-1$ words. For example, if the sequence length m is four, the probability of the sequence $s = w_1 w_2 w_3 w_4$ using a 4-gram model is:

$$P(s) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)P(w_4|w_1 w_2 w_3) \quad (2)$$

If we use a 3-gram model, the probability of token w_4 depends only on the previous two tokens, and the probability of s is:

$$P(s) = P(w_1)P(w_2|w_1)P(w_3|w_1 w_2)P(w_4|w_2 w_3) \quad (3)$$

In this work, we build n-gram models to learn probabilities of using a method given different contexts. With the learned probability distribution, we further calculate the possibility of each token sequence and flag low probability token sequences as potential bugs.

3. APPROACH

Figure 3 shows the overview of Bugram. In this section, we first describe how to parse a project to convert it into tokens (Section 3.1), and then use the tokens to build n-gram models for the project (Section 3.2). Finally, we present how to leverage the n-gram models to detect bugs in the project (Section 3.3).

3.1 Tokenization

To build n-gram models, we need to tokenize the source code of a given project. A main challenge is selecting a suitable level of *granularity* for tokens when building the n-gram models. Existing work builds n-gram models at the syntactic level using low-level tokens to suggest the next tokens for code completion and suggestion [12, 15]. For example, after seeing "`for (int i=0; i<n;`", it suggests the tokens "`i++) {`". Building n-gram models at this level is likely to only detect syntactic errors, e.g., missing "`;`" or "`i++`" in a `for` loop, which will be caught by a compiler.

To detect bugs at the semantic level, we need to build n-gram models at a semantic level. Bugram selects high-level tokens that represent the structure and context of the code using a succinct semantic representation. Take the loop "`for (int i=0; i<n; i++) { foo(i); }`" as an example. Bugram will represent it with the following high-level tokens `[<FOR>, foo(), <END_FOR>]`.

As reported in existing work [49, 50], control flow information is important for the accuracy of bug detection. Inspired by the above work, during the tokenization process, Bugram also considers the control flow information of source code by adding the control flow elements into the token sequences.

Therefore, we 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; `try/catch/finally` blocks; `return` statements; `synchronized` blocks; `switch` statements and `case/default` branches, and `assert` statements.

A method call `methodA()` is resolved to its fully qualified name `org.example.Foo.methodA()` to prevent unrelated methods with an identical name from being grouped together. In addition, the type of exception in the `catch` clauses are considered as they provide important context information to help us infer more accurate contextual information of method sequences.

Bugram uses the Eclipse JDT Core[1] to tokenize the source files, construct the abstract syntax trees (ASTs), and resolve the type information for the tokens. In this work, we consider both *method* and *control flow* as tokens.

3.2 N-gram Model Building

In this work, we use n-gram models to learn a probability distribution over token sequences using all extracted sequences. For every sequence extracted from a method we add all its subsequences to the model. For example, given a token sequence ABC extracted from a method, we add all of its ordered subsequences, i.e., A, B, C, AB, BC, and ABC, to the model. Note that we ignore incontinuous subsequences, such as AC in this example.

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

Smoothing is a common process for n-gram models to help with handling unknown sequences. However, since the entire source code of a project is being used, we have the complete language of all possible sequences. This means smoothing is unnecessary since there are no unknown sequences.

When building n-gram models, one important parameter is `Gram Size` (details are in Section 3.3.1), which defines the length of considered token sequences. N-gram models assume that each token depends only on the previous $n-1$ tokens. To leverage n-gram models to generate probabilities of token sequences, given a specific gram size n, we build a set of *internal probabilities*. For example, the probability of a token sequence ABC calculated by a 3-gram model is $P(ABC) = P(A) \cdot P(B|A) \cdot P(C|AB)$. We refer to $P(A)$, $P(B|A)$, and $P(C|AB)$ as internal probabilities. These internal probabilities can be reused to calculate the probabilities of sequences that shared common subsequences. Thus, we store internal probabilities to cut down the probability calculation time for building n-gram models of different gram sizes. After obtaining all the internal probabilities for an n-gram model, we use them to calculate the probabilities of token sequences.

Previous studies that leverage n-gram models for code completion [15, 37, 47] found that 3-gram, 4-gram, 5-gram, and 6-gram models generated reasonable results. However, the appropriate n-gram size for detecting bugs is unknown. To answer this question, we build n-gram models with gram size from two to ten to study the impact of gram size on the effectiveness of bug detection. The algorithm that we use to build the n-gram model is standard, which is described in Section 2.

3.3 Bug Detection

Bugram detects potential bugs by calculating and ranking the probabilities of all sequences. After obtaining the probabilities of all sequences, Bugram ranks them based on their probabilities in descending order, then reports sequences with the lowest probabilities as potential bugs.

3.3.1 Configurations

A few important factors affect the effectiveness of Bugram, i.e., the number of bugs Bugram can find. The four main factors are as follows. Section 4.2 describes the setup, tuning, and impact of these parameters.

- **Gram Size** - The size of an n-gram model.
- **Sequence Length** - The length of token sequences to be considered when building n-gram models and detecting bugs.
- **Reporting Size** - The number of sequences, in the bottom of the ranked list, which will be reported as bugs.
- **Minimum Token Occurrence** - The minimum number of times a token must occur in the software to be included in an n-gram model.

Gram Size n. As described in Section 2, the gram size is the size n in an n-gram model. The probability of a sequence is estimated by generating the sequence token by token, and the probability of each token is determined by the conditional probabilities using a history of up to $n-1$ tokens. For example, given a token sequence S = $ABCD$, a 2-gram model considers the probabilities of each two sequential tokens, and will calculate its probability with $P(S) = P(A) \cdot P(B|A) \cdot P(C|B) \cdot P(D|C)$. While a 4-gram model considers the probabilities of each four sequential tokens, and calculates its probability as $P(S) = P(A) \cdot P(B|A) \cdot P(C|AB) \cdot P(D|ABC)$. In this work, we build n-gram models with gram size from two to ten to find an appropriate gram size for detecting bugs.

Sequence Length l. For building n-gram models, token sequences are extracted from all methods of a project. The length of token sequences extracted from different methods varies, which can be as small as one, and as large as 200. Breaking these long token sequences into many small sequences may help us obtain fine-grained method usage scenarios and detect more bugs. In this work, we evaluate the impact of different sequence lengths on the performance of Bugram.

```
1 String q[] = qqf.bestQueries("body",20);
2   for (int i=0; i<q.length; i++) {
3       System.out.println(newline+
4       formatQueryAsTrecTopic(i,q[i],null,null));
5 }
```

Figure 4: **The filtering based on `Minimum Token Occurrence` can help Bugram avoid reporting this false bug from the latest version of Lucene.**

Reporting Size s. Different from rule-based bug detection techniques, Bugram detects bugs by identifying token sequences of low absolute probabilities. Thus, an important question is how to set an appropriate threshold to separate sequences that indicate bugs from common sequences that are not bugs. We use this parameter to determine the bottom s sequences in the ranked list and report them as bugs. In general, a larger s allows Bugram to find more bugs at the cost of examining more sequences that potentially indicate bugs. We expect that as the probabilities increase in the ranked list, the percentage of true bugs decreases. An appropriate s should help Bugram find as many bugs without losing much precision of bug detection.

The task of selecting the parameter s in Bugram is the counterpart of selecting a rule probability threshold in rule-based bug detection approaches, such as PR-Miner [23]. As described in Section 1, rule-based approaches select a high-probability rule based on the conditional probability of tokens. For example, if the probability $P(C|AB)$ is higher than the threshold, $\{AB => C\}$ will be selected as a rule, and occurrences of the sequence AB followed by a call other than C is reported as a bug. According to the definition of conditional probability, $P(C|AB) = P(ABC)/P(AB)$, meaning that rule-based techniques consider the probability of the sequence ABC and compare it to the background probability of AB. However, when the rule $\{EF => G\}$ is evaluated, the background probability is now that of the sequence EF since the conditional probability is $P(G|EF)$. To achieve optimal performance, rule-based methods should ideally find the individual correct threshold for each background probability. This is of course not practically feasible, which is the reason a single threshold is used in practice. In Bugram, we avoid this problem by directly evaluating the probability of the entire sequence. In this case, it is more theoretically sound to select a single threshold.

Minimum Token Occurrence y. After performing the tokenization process described in Section 3.1, Bugram keeps only tokens with occurrences greater than y in the project. Filtering out uncommon tokens is a standard technique for natural language processing (NLP) techniques [26]. In this paper, the rationale is that some methods are generally not well used, or too unique, making them corner-cases that are harder to evaluate on a statistical basis. As such, their inclusion leads to generating false bugs. Take the code snippet in Figure 4 as an example. Using a 3-gram model, the sequence `[bestQueries, println, formatQueryAsTrecTopic]` is ranked at the bottom of all token sequences by their probabilities. However, this token sequence is not a bug. It has a low probability because it uses two infrequent private methods `bestQueries` and `formatQueryAsTrecTopic`, each of which is used only once in the whole project.

To avoid reporting the above false bug, Bugram performs a token level filtering. It filters out all tokens that appear fewer than a given y. Such process can help Bugram avoid reporting many token sequences with low probabilities that are not bugs.

3.3.2 *Pruning False Bugs*

Bugram identifies token sequences with low probabilities as potential bugs. However, some low probability token sequences are unusual/special uses of code and are not bugs, they are false bugs for the purpose of bug detection. To filter out false bugs, we reduce the number of reported bugs (also called *candidate bug set*) by keeping only token sequences at the bottom of at least two ranked lists generated by different n-gram models with different sequence lengths. The rationale is that if a bug can be detected by multiple ranked lists, there is a higher chance that it is a true bug. Remember that given a specific gram size, we generate multiple n-gram models of different sequence lengths ranging from two to ten. Therefore, we only report token sequences that are at the bottom of at least two different n-gram models with the same gram size but different sequence lengths.

For example, if both sequences $ABCDE$ and BCD are ranked at the bottom of the list of 5-token-sequences and 3-token-sequences respectively, Bugram identifies them as an **overlap (two sequences contain a common substring)** and reports $ABCDE$ and BCD as one bug. More formally, we obtain a new candidate bug set using the following formula:

$$C(n,t) = \bigcup_{\forall i,j \in M, i \neq j} (Bottom(n,t,i) \cap Bottom(n,t,j)) \quad (4)$$

where $C(n,t)$ is the candidate bug set generated by an $n\text{-}gram$ model with the reporting size of t. M is the set of sequence length, and i and j are two different sequence lengths. n is the gram size. $Bottom(n,t,i)$ is the bottom t token sequences generated by an $n\text{-}gram$ model with sequence length of i. Note that $\cap$ denotes the **overlaps** that are at the bottom of the two different n-gram models, and $\bigcup$ denotes the union of the **overlaps**.

4. EXPERIMENTAL SETUP

We evaluate Bugram in terms of the number of detected bugs and detection precision, and explore appropriate parameters for Bugram. All our experiments are conducted on a 4.0GHz i7-3930K desktop with 64GB of memory.

4.1 Evaluated Software

We evaluate Bugram on 16 widely-used open source Java projects ranging from 36 thousand lines of code (KLOC) to almost one million lines of code (MLOC). Table 1 lists their versions, numbers of files, lines of code (LOC), and numbers of methods. We used the latest version of each project.

4.2 Parameter Setting and Sensitivity

To build n-gram models and detect bugs effectively, we need to tune these parameters proposed in Section 3.3.1. We use three widely-used and representative projects from Table 1, i.e., Pig, Hadoop, and Solr, to study the impact of different parameters on the performance of Bugram. Specifically, we tune three of the four parameters, i.e., 9 different gram sizes, 9 different sequence lengths, and 5 different reporting sizes. For minimum token occurrence, we remove any token that appears fewer than three times [26]. In total, there are 9*9*5 = 405 possible combinations of the four parameters. Note that, for each combination, we need to manually examine the reported bugs for each project, which is prohibitively expensive. To save efforts, we only pick the three representative projects to tune the four parameters (in total we need to manually examine the reported bugs of 405*3 combinations) in this work. In practice, if users can afford more time, they can tune on more projects to obtain an optimal parameter combination of Bugram to detect bugs more effectively.

Table 1: **Projects evaluated in our experiments**

Project	Version	Files	LOC	Methods
Elasticsearch	1.4	3,130	272,261	28,950
GeoTools	13-RCI	9,666	996,800	89,505
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	2.7.1	4,307	596,462	46,104
Hbase	1.1.1	1,392	465,456	42,948
Pig	0.15.0	948	121,457	9,323
Solr-core	5.2.1	1,061	146,749	9,938
Lucene	5.2.1	2,065	293,825	18,078
Opennlp	1.6.0	603	36,328	2,954
Struts	2.3.24	2,022	157,499	15,254
Zookeeper	3.5.0	492	61,708	5,034
Nutch	2.3.1	409	198,560	2,309
Cassandra	2.2.0	1,616	280,716	15,233

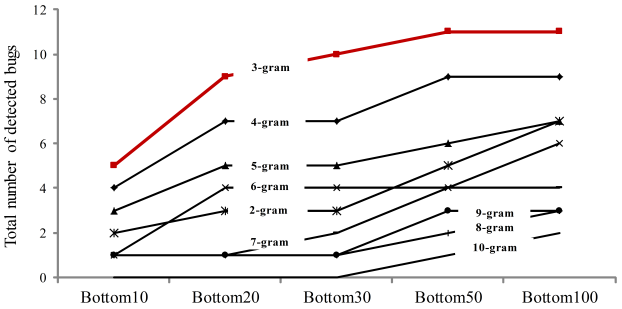


Figure 5: **Impact of the gram size on the number of true bugs detected**

Setting Gram Size. Different gram sizes enable Bugram to use different internal probabilities to calculate the probabilities of token sequences. We build n-gram models for each project, and the gram size ranges from two to ten. To evaluate the performance of n-gram models of different gram sizes, we calculate the probabilities of all token sequences and rank them based on their probabilities in descending order, then we examine the bottom 10, 20, 30, 50, and 100 sequences from each n-gram model respectively, and manually verify whether a token sequence contains a bug or not.

For each n-gram model, we count the number of real bugs detected in the three projects. Figure 5 shows the results of the three projects combined. The results show that Bugram finds the most number of true bugs with a 3-gram model. Thus, in this paper, we build 3-gram models for Bugram to detect bugs.

Setting Sequence Length. As described in Section 3.3, we break long sequences extracted from a function into small subsequences. Different sequence lengths enable Bugram to capture different program scenarios and further affect the performance of Bugram. To evaluate the impact of different sequence lengths, we perform Bugram with sequence length ranges from two to ten. For each sequence length, we build a 3-gram model, then calculate probabilities of all sequences. Based on generated probabilities, we rank all sequences. We examine the bottom 50 sequences with low probabilities to check how many true bugs are detected.

Table 2 shows the results of detected true bugs in the bottom 50 sequences with different sequence lengths. As we can see, sequence length can significantly affect the performance of Bugram.

Table 2: **Detected true bugs in the bottom 50 token sequences with different sequence lengths**

Project	Sequence Length								
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Pig	0	1	1	3	2	3	1	1	1
Hadoop	0	3	4	7	3	3	2	0	0
Solr	0	1	1	1	2	0	0	0	0

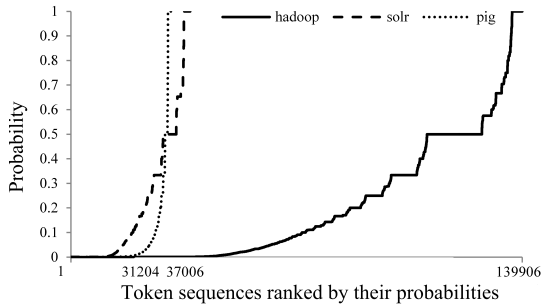


Figure 6: **Probabilities distribution of all token sequences in Hadoop, Solr, and Pig**

N-gram models, with sequence length ranges from three to eight, enable Bugram to detect bugs effectively. For example, when the sequence length is equal to five, we find seven true bugs on Hadoop, three on Pig, and one on Solr. When the sequence length is quite low (e.g., two) or quite big (e.g., nine, and ten), Bugram detects no bugs in two of the three examined projects. Thus, in this paper, sequence length ranges from three to eight.

Setting Reporting Size. In this work, we use this parameter to limit the number of sequences in the bottom of ranked sequence list to be reported as bugs. An appropriate reporting size might help us identify many true bugs with a small number of false positives. For each examined project, we build 3-gram models with sequence length ranges from three to eight.

We first examine the probabilities of all sequences, to explore whether there has a clear cutoff between low probability sequences and high probability sequences. We normalized the probabilities of all sequences in a project, since the range of sequence probability distribution varies in different projects, e.g., in Hadoop the sequence probability range is $[1.0 \times 10^{-11}\ 5.4 \times 10^{-4}]$, while this range for pig is $[2.84 \times 10^{-5}\ 2.7 \times 10^{-3}]$. Figure 6 shows the normalized probabilities of all sequences (ranked by probability), the X-axis is the number of sequences in different projects. As we can see, the probability curves are quite smooth at the bottom ten thousand sequences. However, it is prohibitively expensive to examine all these sequences.

Next, we narrow down the reporting size by only looking at the bottom 100 sequences. Specifically, for each project, we examine how many true bugs are detected when the reporting size is equal to 10, 20, 30, 50, and 100, which means we only examine the bottom 10, 20, 30, 50, and 100 sequences in the ranked list. In practice, if developers can afford more time, they can examine more sequences to find more bugs. The number of detected true bugs and detection precisions of the three projects are shown in Figure 7a, Figure 7b, and Figure 7c. As we can see with the increasing of reporting size, the number of detected bugs increases, while corresponding detection precision declines sharply. When the reporting size is equal to 100, the corresponding detection precision is smaller than 20%. In this study, we set reporting size equal to 20, which could enable us to detect 13 true bugs with an average detection precision of 49% on the three examined projects.

Setting Minimum Token Occurrence. This parameter is the minimum number of times a token is required to appear in a program to be included in sequences. An appropriate value of this parameter helps filter out token sequences that use unusual/special methods, thus have low probabilities, but are not bugs.

In this study, we remove any token that appears fewer than three times. This is a common practice in NLP research, aimed at improving system performance [26].

4.3 Comparison with Existing Techniques

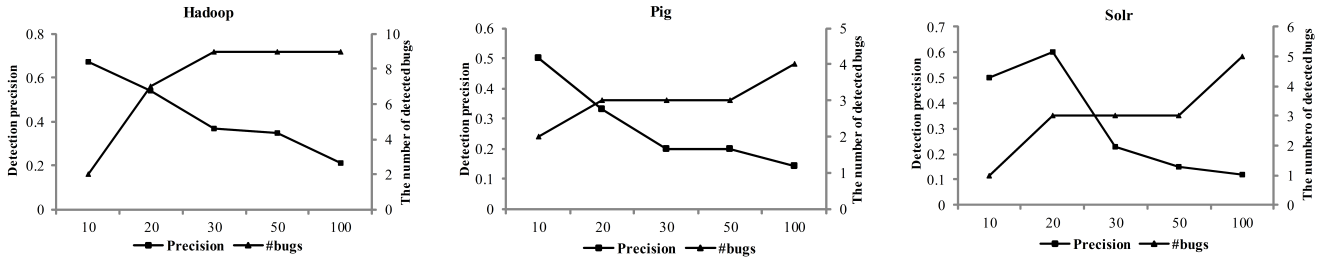
We compare Bugram with five existing graph- and rule-based approaches. First, we choose the most closely related work, PR-Miner [23]. While comparing with PR-Miner allows us to compare Bugram with an existing approach as is, we also want to study the sole impact of using n-gram models. In addition to using n-gram models, the differences between Bugram and PR-Miner include (1) Bugram uses control flow information, but PR-Miner does not, and (2) Bugram preserves the token order, while PR-Miner ignores the token order. As discussed in Section 3, we use control flow information because it has been shown to be beneficial for bug detection techniques [6, 50]. Therefore, our second approach for comparison is identical to PR-Miner except that it considers both the order of tokens and control flow information. Since PR-Miner is not publicly available, we have reimplemented our own version of it. We refer to our implementation of PR-Miner as *FIM*, which stands for **F**requent **I**temset **M**ining. We call our implementation of the second approach described above *FSM*, which stands for **F**requent **S**equence **M**ining, because *FSM* mines rules with order preserved, which is what frequent sequence mining does.

To implement PR-Miner, we follow each step described in [23], we first parse the source code and extract variables, method calls, classes in a function. After that, we hash selected elements into numbers. Next, each function is mapped to an itemset. Then, using these itemsets, we perform frequent itemset mining, as provided by Weka [11] to mine frequent itemsets with a specific support and confidence. Finally, these frequent itemsets are treated as rules, and violations of these rules are identified as bugs.

Note that, FIM does not consider the order of tokens. Therefore, given a frequent itemset ABC, FIM may generate many rules, e.g, $\{A => BC\}$, $\{B => AC\}$, $\{AB => C\}$, $\{AC => B\}$, $\{C => AB\}$, and $\{BC => A\}$. Any violations of these rules will be reported as potential bugs. While FSM considers the order, thus given the same frequent itemset ABC, FSM generates at most two rules, i.e., $\{A => BC\}$ and $\{AB => C\}$. The more rules are inferred, the more potential bugs are likely to be reported. Thus, in practice, both the number of generated rules and the number of reported bugs of FIM are significantly larger than those of FSM.

To reduce false positives, PR-Miner set the support threshold to 15 and the confidence threshold to 90%. With these thresholds, PR-Miner reports many potential bugs, e.g., PR-Miner reported 1,447 potential bugs in the Linux kernel [23]. To save effort, they examined only the top 60 potential bugs ranked by confidence.

These thresholds are only evaluated on three C projects. We find that these thresholds produce poor results on the Java projects used in this paper, e.g., we find 0 true bugs in the top 60 ranked bugs in the three Java projects, i.e., Hadoop, Solr, and Pig. Therefore, to set appropriate support and confidence thresholds for FIM and FSM, we have explored FIM and FSM with different combinations of support and confidence on the three projects. For FIM, we find that when the support is equal to seven and the confidence is larger than 75%, it performs the best on the three projects when examining the top 80 sequences ranked by confidence (we have examined up to the top 100, and found the top 80 gives the highest precision and recall). For FSM, when the support is equal to five and the



(a) **Results of Hadoop** (b) **Results of Pig** (c) **Results of Solr**

Figure 7: **Detection precision and number of detected bugs in the overlaps of bottom s token sequences with low probability**

confidence is larger than 85%, it performs the best on the three projects. For a fair comparison, we tune all four parameters of Bugram on the same three projects and apply the best parameters on the rest of the evaluated projects.

Second, we compare Bugram with three graph- and rule-based bug detection approaches, i.e., JADET [50], Tikanga [49], and GrouMiner [32]. These three approaches leverage graph models and frequent itemset mining techniques to mine rules that capture both method order and control flow information to detect bugs. Different from PR-Miner that was evaluated on C projects, all these three approaches were evaluated on open source Java projects. Thus, Bugram can be applied to these projects directly. Since, two of these three tools, i.e., Tikanga and GrouMiner, are not publicly available, to compare with these three approaches, instead of implementing our own versions, we perform Bugram on the 14 projects evaluated by these three approaches, and compare our detection results with the results from these three studies. In addition, since JADET is an open source tool, we also apply JADET on projects listed in Table 1, and compare its detection results with Bugram.

4.4 Evaluation Measures

We manually examine the reported potential bugs and categorize the bugs into three types: *True Bugs*, *Refactoring Opportunities*, and *False Positives*. True bugs are faults and can be fixed by altering the code and correcting its behaviour. Refactoring opportunities are bad practices and can be fixed by refactoring the infrequent code snippets to make them more regular. Any reported bugs that do not fit into the above two groups are considered to be false positives. We refer to the number of true bugs and refactoring opportunities as *True Positives*.

To evaluate the performance of a bug detection approach, we use three measures: standard *precision*, *relative recall* [40], and *F1*. Note that we use *relative recall* not standard *recall*, because it is not practical to know all bugs in a project. The precision is: (*True Positive*)/(*Reported Potential Bugs*); To calculate the relative recall, we first define the *Relative Ground Truth* [40] as all the unique true positives reported by Bugram, FIM, and FSM. For each of the three approaches, we calculate its relative recall as: (*True Positive*)/(*Relative Ground Truth*); F1 is the harmonic mean of the precision and relative recall.

4.5 Manual Examination of Reported Results

Following prior work [4, 6, 10, 23, 35, 46, 49, 50], we manually check whether the bugs reported by Bugram are true positives. For manual evaluation, a token sequence was only considered buggy if both the first author and a non-author graduate student agreed. Given a reported buggy sequence, we consider it a bug if it meets one of the following conditions:

- It has obviously incorrect project specific function calls. For example, the true bug in Figure 2: for the purpose of logging, developers should convert the object value to a string by calling a local method `toString`, without such conversion, the logged information is the memory address of the object value, not its textual information. This bug has already been confirmed and fixed by PIG developers.
- It violates common API usages, e.g., exception-handling API usages[2] and log-related API usages[3]. For exception handling APIs, we detect several true bugs that violate their usages, e.g., in the true bug in Figure 8(b), developers did not handle all potential exceptions that might be thrown by method `waitForCompletion`, which may crash the system if any of these exceptions are thrown. For log-related APIs, one common usage is that the checked log level and the used log level should be the same. Several of our detected true bugs violate this usage, e.g., in the true bug in Figure 8(c): before calling the method `info()` to log messages, instead of checking whether the log level `info` is available, developers checked whether `debug` is available, which is incorrect.

We consider a reported buggy sequence a refactoring issue based on principles of code smells proposed in [27], e.g., *duplicated code*—when two code fragments look almost identical. Duplicated code could hinder maintenance because developers need to track and modify each repeating fragment. Developers could refactor the duplicated code by extracting it into a function.

5. EXPERIMENTAL RESULTS

This section presents the results of detected bugs (Section 5.1 and Section 5.2) including the comparison with existing graph- and rule-based techniques, detected bug examples (Section 5.3), and the execution time of Bugram (Section 5.4).

5.1 Comparison with FSM and FIM

Table 3 shows the number of bugs detected by Bugram, FSM, and FIM on each evaluated project. In total, Bugram reported 59 potential bugs, 42 of which are correct and useful—25 true bugs, and 17 refactoring opportunities. We have reported these true bugs to developers, 7 of which have already been confirmed by developers, while the rest await confirmation. The results suggest that Bugram is effective in finding real bugs in widely-used mature software projects to improve software reliability.

As described in Section 4.4, the relative recall shows the ability of a technique in finding new bugs, while the precision indicates the ability of a technique in avoiding reporting false bugs. F1 is the harmonic mean of the precision and recall. The relative recall of Bugram is 54.5%, which is higher than FIM's relative recall of 40.3% and FSM's relative recall of 26.0%. The detection precision of Bugram is 71.2%, which is higher than FIM's precision of 2.4%

[2]https://docs.oracle.com/javase/tutorial/essential/exceptions

[3]https://logging.apache.org/log4j/1.2/manual.html

Table 3: **Bug detection results. Reported is the number of reported bugs, TBbugs is the number of true bugs, and Refs is the number of refactoring opportunities. We manually inspect all reported bugs except for FIM whose 'Inspected' column shows the number of bugs inspected.** ***Numbers in brackets are the numbers of true bugs detected by Bugram that are detected by neither FIM nor FSM.***

	Bugram			FSM			FIM			
Project	**Reported**	**TBugs**	**Refs**	**Reported**	**TBugs**	**Refs**	**Reported**	**Inspected**	**TBugs**	**Refs**
Elasticsearch	3	1(1)	0	0	0	0	987	80	0	2
GeoTools	4	2(2)	1	4	0	2	1,203	80	1	2
JEdit	3	0	1	0	0	0	451	80	0	2
Proguard	1	1(1)	0	1	1	0	665	80	1	0
Vuze	2	0	2	0	0	0	435	80	0	4
Xalan	3	2(2)	0	0	0	0	378	80	0	0
Hadoop	13	7(6)	4	13	3	0	869	80	2	3
Hbase	1	1(1)	0	10	2	3	774	80	1	0
Pig	9	3(2)	4	6	0	2	605	80	1	3
Solr-core	5	3(3)	1	0	0	0	787	80	0	1
Lucene	2	0	0	10	2	2	676	80	1	0
Opennlp	6	2(2)	2	3	0	0	806	80	0	2
Struts	5	1(1)	2	9	1	0	232	80	0	0
Zookeeper	0	0	0	3	0	0	442	80	0	1
Nutch	2	2(2)	0	1	0	1	253	80	1	1
Cassandra	0	0	0	7	0	1	324	80	0	2
Total	**59**	**25(23)**	**17**	**67**	**9**	**11**	**9,887**	**1,280**	**8**	**23**
Relative Recall	**54.5%**			**26.0%**			**40.3%**			
Precision	**71.2%**			**29.9%**			**2.4%**			
F1	**61.7%**			**27.8%**			**4.5%**			

and FSM's precision of 29.9%. The F1 of Bugram is 61.7%, again higher than FIM's F1 of 4.5% and FSM's F1 of 27.8%. The results suggest that Bugram can find more bugs than examined rule-based approaches and is more precise than them, suggesting Bugram complements existing rule-based bug detection techniques.

In addition, as shown in Table 3, among the 25 true bugs detected by Bugram, only two can also be detected by FIM and FSM. The majority (23) of the true bugs can only be detected by Bugram, showing that Bugram can find real bugs in real-world software that examined rule-based approaches cannot find. In comparison, FIM detected eight true bugs, and 23 refactoring opportunities. FSM reported 67 potential bugs, and nine of which are true bugs, and 11 are refactoring opportunities. Since six true bugs are detected by both FIM and FSM, a total of 11 unique true bugs are detected by these two approaches, nine of which cannot be detected by Bugram. In total, there are 77 unique true positives generated by the three approaches.

Since FSM considers both the control flow and order of tokens, both the numbers of rules and bugs discovered by FSM are much smaller than those of FIM. Table 3 shows that FIM reported a total of 9,887 potential bugs, while FSM only reported 67 potential bugs.

As we described in Section 1, Bugram and FIM detected bugs based on different probability distributions of token sequences. Bugram identifies token sequences with absolute low probability as bugs, while FIM and FSM identify token sequences with relatively low probability as bugs. A relatively low probability token sequence might have a high absolute probability with a high rank in all token sequences extracted from a project. Thus, **our results suggest that Bugram and rule-based bug detection techniques complement each other to detect more bugs.**

5.2 Comparison with JADET, Tikanga, and GrouMiner

As described in Section 4.3, we also compare Bugram with three graph- and rule-based bug detection approaches, i.e., JADET [50], Tikanga [49], and GrouMiner [32]. These approaches have been evaluated on Java projects, and their authors have presented the number of detected potential bugs and manually identified true bugs. Since JADET, Tikanga, and GrouMiner each reported many potential bugs for the evaluated projects, to save effort, the authors of the three tools manually verified a subset of reported potential bugs, i.e., top 10 in JADET, top 25% in Tikanga, and top 15 in GrouMiner. For a fair comparison, we apply Bugram on the projects that are evaluated by these approaches, and use the same Bugram parameters that are used in the comparison with PR-Miner, meaning that Bugram parameters are not tuned for these projects. JADET was evaluated on five projects, Tikanga was evaluated on six projects, and GrouMiner was evaluated on nine projects. We exclude four projects which are not publicly available anymore. In total, 16 projects (14 unique) are available (Table 4).

Table 4: **Comparison with JADET, Tikanga, and GrouMiner. 'Fixed' denotes the number of true bugs detected by Bugram that have already been fixed in later versions.** * ***denotes the number of unique true bugs detected by Bugram that the tools in comparison failed to detect.***

	Graph-based tools	Bugram		
Project	**JADET**	**Reported**	**TBugs**	**Fixed**
AZUREUS 2.5.0	1	8	4	4
columba-1.2	0	4	1*	1
aspectj-1.5.3	2	5	0	0
	3	**17**	**5**	**5**
Project	**Tikanga**	**Bugram**		
aspectj-1.5.3	9	5	0	0
tomcat-6.0.18	0	13	4*	2
argouml-0.26	1	3	1	1
Vuze_3.1.1.0	0	8	0	0
columba-1.4	1	6	1	1
	11	**35**	**6**	**4**
Project	**GrouMiner**	**Bugram**		
columba-1.4	1	6	1	1
ant-1.7.1	1	3	1	1
log4j-1.2.15	0	7	2*	2
aspectjrt-1.6.3	1	12	2	1
axis-1.1	0	6	3*	1
jedit-3.0	1	5	1	1
jigsaw-2.0.5	1	1	1	1
struts-1.2.6	0	8	0	0
	5	**48**	**11**	**8**

Table 4 shows the detection results of Bugram and these three bug detection approaches. JADET detected three true bugs on the three projects. Bugram detected five true bugs on the same projects, at least one of which cannot be detected by JADET. Since these papers did not report the full list of detected bugs, we do not know if the bugs detected by JADET and Bugram overlap. However, since JADET detected 0 true bugs in `columba`, while Bugram detected one bug, we know that Bugram detected one bug that JADET can-

not detect. We also found that all five true bugs detected by Bugram are fixed in a later version by developers. For the same reason as above, we do not know if the bugs detected by JADET have been fixed in a later version. Tikanga detected 11 true bugs on the five projects, while Bugram detected six true bugs on the same projects, four of which are unique to Bugram (detected in `tomcat`). Four of the six true bugs have already been fixed. GrouMiner detected five true bugs on the eight projects, while Bugram detected 11 true bugs on the same projects, five of which are unique to Bugram (detected in `log4j` and `axis`). Eight of the 11 true bugs have already been fixed. In total, the three approaches detected 19 true bugs in the 14 projects, while Bugram detected 21 true bugs, and we have manually checked that 16 of them have already been fixed in a later version. In addition, at least 10 of the 21 true bugs cannot be detected by these three tools.

Since JADET is an open source tool, we further apply JADET (with recommended parameters) on the projects listed in Table 1. Results shown that JADET did not detect any true bugs, the top 10 potential bugs detected by JADET are missing method calls of JAVA library classes, e.g., `Map`, `List`, `Iterator`, etc. JADET, Tikanga, and GrouMiner are based on object usage graph models, so rules generated by them are method usages of classes (both library classes and project specified classes). For example, one of JADET's representative rules is the method `Iterator.next()` should always follow the method `Iterator.hasNext()`. Violations of this rule will be flagged as potential bugs. However, it is not necessary to use both the two methods in every scenario. Thus, it is possible for JADET to report large numbers of false positives related to the Java library classes listed above. The comparison results show that **Bugram is complementary to these graph- and rule-based approaches.**

In addition, the detection precision of Bugram is better than these three techniques. JADET [50] reported that three of the 30 potential bugs detected in the three projects listed in Table 4 are true bugs. Thus, JADET has a detection precision of 10.0%, while Bugram achieves a precision of 29.4% on the same projects. Tikanga's authors manually examined 118 potential bugs on the five projects, 11 of which are true bugs [49], indicating a detection precision of 9.3%. While Bugram achieves a precision of 17.1% on the same five projects. Similarly, GrouMiner has a detection precision of 4.2% on the eight projects [32], while Bugram achieves a precision of 22.9% on the same projects.

5.3 Examples

Example Bugs. We show some of the detected true bugs in Figure 8. Specifically, Figure 8 shows three examples of detected true bugs in ProGuard and Nutch that are detected by our tool. FIM and FSM fail to detect them. We reported the three bugs to the ProGuard and Nutch developers, and all of them have been confirmed as true bugs. In addition, the bugs in Figure 8(a) and Figure 8(c) have already been fixed by developers.

The bug in Figure 8(a) is caused by an incorrect API usage. Specifically, the instantiation of `ConfigurationWriter (writer)` should be closed in a `finally` block instead of a `try` block. To fix this bug, instead of closing the `writer` in the try block, developers added a `finally` block, and closed the `writer` in it. Figure 8(b) also shows a true bug. The method `waitForCompletion` might throw several exceptions, e.g., `ClassNotFoundException` and `IOException`, while in this case, developers used this function without handling these potential exceptions. To fix this bug, developers should either use a catch block to handle the exceptions or raise the exceptions to be handled by the calling functions. Figure 8(c) shows another

(a) **A confirmed and fixed true bug from ProGuard (BugID: 582).**

```
try{
ConfigurationWriter writer = new
    ConfigurationWriter(file);
writer.write(getProGuardConfiguration());
writer.close();}
catch (Exception ex){...}}
```

(b) **A confirmed true bug from Nutch (BugID: NUTCH-2076).**

```
try {
   currentJob.waitForCompletion(true);
   } finally { ...
    }}...}
```

(c) **A confirmed and fixed true bug from Nutch (BugID: NUTCH-2256).**

```
if (LOG.isDebugEnabled()) {
LOG.info("Crawl delay for queue: "...);}
```

Figure 8: **True bug examples from version 5.2 of ProGuard (a) and version 2.3.1 of Nutch (b and c)**

```
1 byte dt1 = bb1.get();
2 byte dt2 = bb2.get();
3  switch (dt1) {
4  case BinInterSedes.BIGINTEGER:{
5  if{
6    int sz1 = readSize(bb1, bb1.get());
7    int sz2 = readSize(bb2, bb2.get());} }
8    ...}
```

Figure 9: **A refactoring bug example from version 0.15.0 of Pig**

true bug. This bug is caused by the inconsistency between the checked log level (debug) and the used log level (info). To fix this bug, developers replaced the method `info()` with the method `debug()` to make it consistent with the checked log level. **Example Refactoring Opportunities**. Figure 9 shows an example of refactoring opportunity detected by our tool from Pig project. In this example, developers first define two variables `dt1`, and `dt2` to keep data via method call `get()` of objects `bb1` and `bb2`. Next, in the `switch` block, one of the `case` branch needs data from objects `bb1` and `bb2`, while such data already kept in variables `dt1` and `dt2`. Without reusing these two variables, developers call `get()` of objects `bb1` and `bb2` to obtain data again. This costs extra memory and time and should be refactored by using variables `dt1` and `dt2` directly in lines 6 and 7.

Some of our detected bugs may appear to be simple, **but many (7) of them have been confirmed or fixed (4 confirmed and fixed, 2 confirmed with patches proposed, 1 confirmed) by the developers of these projects, suggesting the value of our approach.**

5.4 Execution Time and Space

We collect the time and space costs for all the 16 projects listed in Table 1, and details are presented in Table 5. We can see that the total execution time for tokenization, model building, and bug detection varies from 50 to 878 seconds. Our largest evaluated project, GeoTools, uses 2GB of memory. As shown in the table, most of the time is spent building ASTs with type information with a fraction of the time on building n-gram models and detecting bugs. The results demonstrate Bugram's practical value.

6. THREATS TO VALIDITY

Implementation of PR-Miner. To compare Bugram with rule-based bug detection approaches, we have reimplemented

Table 5: **Execution time in seconds**

Project	Total	Tokenization	Model Building and Bug Detection
Elasticsearch	162	160	2
GeoTools	878	872	6
jEdit	86	85	1
Proguard	61	60	1
Vuze	312	310	2
Xalan	80	79	1
Hadoop	447	443	4
Hbase	151	149	2
Pig	93	91	2
Solr-core	97	95	2
Lucene	206	203	3
Opennlp	50	49	1
Struts	223	220	3
Zookeeper	42	41	1
Nutch	36	35	1
Cassandra	157	155	2

a rule-based approach PR-Miner [23], since PR-Miner is not publicly available. The PR-Miner paper reported higher precision than what we reported with our implementation of PR-Miner in this paper. One possible reason is that PR-Miner has only been evaluated on C projects. Its false positive pruning approach, recommended threshold values of support and confidence may only be effective on C projects, while in this study we evaluate on Java projects. However, for our implementation of PR-Miner, we have tried our best to tune parameters, e.g., threshold values of support and confidence, to obtain the best results. This is our best effort given that PR-Miner is not publicly available. Our comparison is fair since both Bugram and PR-Miner are tuned and evaluated on the same projects.

Bugs are verified by the authors. Following prior work [4, 6, 10,23,35,46,49,50], we manually check whether the potential bugs reported by the tools are true positives. Although this approach is a common practice, this process contains bias since the authors of this paper are not the developers of these projects. We mitigate this threat by sending the bugs to developers for further confirmation, which can take a long time. So far, 7 have already been confirmed as true bugs by developers.

7. RELATED WORK

Statistical language models. Statistical language models have been successfully used for tasks including code completion [37,51], fault localization and coding style consistency checking [2, 14]. Hindle et al. [15] leveraged n-gram language models to show that source code has high repetitiveness. Han et al. [12] presented an algorithm to infer the next token by using a Hidden Markov Model. Pradel et al. [34] proposed an approach to generate object usage specifications based on a Markov Model. Yusuke et al. [33] leveraged n-gram models to generate pseudo-code from software source code. Ray et al. [36] used n-gram models to study language statistics of buggy code, which showed that software buggy lines are more unnatural than non-buggy lines. They also proposed a defect prediction model based on n-gram models. Specifically, they built n-gram models on an old version of a software project, and then used entropy from the n-gram models to estimate the naturalness of the source code lines in a later version. Source code lines with higher entropy values are flagged as buggy lines. There are three main differences between their approach and Bugram. First, given a software project, Bugram directly builds n-gram models on it and detect bugs in this project, while their tool requires an old version of this project as training data. Second, they build n-gram models at the token level, while we build n-gram models at a higher level (e.g., statements, method calls, and control flows) (Section 3.1) aiming to detect semantic bugs more effectively. Third, their approach leverages entropy while Bugram uses probability to detect bugs. Using probability and entropy to rank token sequences are two different approaches [26]. The entropy used in [36] combines probability and sequence length. It may worth comparing using entropy versus probability for detecting bugs in the future.

White et al. [51] and Raychev et al. [37] investigated the effectiveness of language models, i.e., n-gram and deep learning models, for code completion. Movshovitz-Attias et al. [29] leveraged n-gram models to predict class comments for program source file documents. Campbell et al. [5] built n-gram models with historical correct source code to locate the cause of syntax errors.

Some studies used n-gram token sequences instead of n-gram models to solve software engineering tasks. Nessa et al. [30] and Yu et al. [57] leveraged n-gram token sequences to help software fault localization. Hsiao et al. [17] proposed to use n-gram token sequences and *tf-idf-style* measures to detect code clone and related bugs. In contrast, Bugram is not limited to detecting clone bugs.

Rule mining and defect detection. Many techniques have been developed for programming rule mining and bug detection [1, 4, 6, 8–10, 16, 18, 19, 21–23, 25, 28, 39, 41–43, 45, 46, 48–50, 52–56]. Engler et al. [8] shown that checking the inconsistent programmers' beliefs can be an effective approach to detect real bugs. Li et al. [23] developed PR-Miner to mine programming rules from C code and detect violations of these rules. Chang et al. [6] proposed an approach to mine rules for detecting neglected conditions. Wasylkowski et al. [50] proposed JADET which combined frequent itemset mining and object usage graph models to detect object usage anomalies. Gruska et al. [10] extended JADET by mining object usages from over 6000 projects. Wasylkowski et al. [49] proposed Tikanga which combined JADET with model checking and concept analysis to learn and check operational preconditions. Nguyen et al. [32] extended JADET by mining the usage patterns of multiple objects. Different from existing rule-based bug detection tools, Bugram detects bugs by calculating and ranking the probabilities of token sequences based on the probability distribution of tokens in a project.

8. CONCLUSIONS AND FUTURE WORK

This paper introduces Bugram, that leverages n-gram models to detect bugs. Bugram detects potential bugs via calculating and ranking the probabilities of program tokens based on the probability distribution of program tokens in a project. Low probability token sequences are flagged as potential bugs. We evaluate Bugram in two ways. First, we compare it with two rule-based bug detection approaches on 16 projects. Results show that Bugram detects 25 true bugs, and 23 of which cannot be detected by PR-Miner. Second, we further apply Bugram on 14 projects evaluated in three graph- and rule-based tools, i.e., JADET, Tikanga, and GrouMiner. Bugram detects 21 true bugs, at least 10 of which cannot be detected by these three tools. Our results suggest that Bugram is complementary to existing rule-based bug detection approaches.

In the future, we plan to build n-gram models from multiple projects to perform cross-project bug detection, which may help us find more bugs more accurately. We also plan to explore different approaches to segment sequences when building n-gram models. Currently, Bugram breaks a token sequence from a method into sequences of fixed lengths. It would be promising to break sequences at the boundaries of code's semantic blocks. In addition, we plan to extend Bugram to C/C++ projects, and combine Bugram with rule-based approaches to detect more bugs.

9. ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

10. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *FSE' 07*, pages 25–34, 2007.

[2] M. Allamanis, E. T. Barr, and C. Sutton. Learning Natural Coding Conventions. In *FSE' 14*, pages 281–293, 2014.

[3] L. R. Bahl, P. Brown, P. V. de Souza, and R. Mercer. A Tree-Based Statistical Language Model for Natural Language Speech Recognition. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(7):1001–1008, 1989.

[4] L. Benjamin and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *FSE' 05*, pages 296–305, 2005.

[5] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax Errors Just Aren't Natural: Improving Error Reporting with Language Models. In *MSR' 14*, pages 252–261, 2014.

[6] R.-Y. Chang, A. Podgurski, and J. Yang. Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In *ISSTA' 07*, pages 163–173, 2007.

[7] E. Charniak. Statistical Language Learning. In *First MIT Press paperback edition*. MIT Press, 1996.

[8] 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*, volume 35. 2001.

[9] 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. In *Sci. Comput. Program.*, volume 69, pages 35–45, 2007.

[10] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. In *ISSTA' 10*, pages 119–130, 2010.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[12] S. Han, D. R. Wallace, and R. C. Miller. Code Completion from Abbreviated Input. In *ASE' 09*, pages 332–343, 2009.

[13] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE' 02*, pages 291–301, 2002.

[14] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli. Will They Like This?: Evaluating Code Contributions with Language Models. In *MSR' 15*, pages 157–167, 2015.

[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *ICSE' 12*, pages 837–847, 2012.

[16] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[17] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using Web Corpus Statistics for Program Analysis. In *OOPSLA '14*, pages 49–65, 2014.

[18] J. Huang, P. O. Meredith, and G. Rosu. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI' 14*, pages 337–348, 2014.

[19] H. Kagdi, M. L. Collard, and J. I. Maletic. An Approach to Mining Call-Usage Patterns with Syntactic Context. In *ASE' 07*, pages 457–460, 2007.

[20] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A Search Engine for Binary Code. In *MSR' 13*, pages 329–338, 2013.

[21] J. Lawall and D. Lo. An Automated Approach for Finding Variable-constant Pairing Bugs. In *ASE' 10*, pages 103–112, 2010.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI' 04*, pages 20–20, 2004.

[23] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *FSE' 05*, pages 306–315, 2005.

[24] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai. AntMiner: Mining More Bugs by Reducing Noise Interference. In *ICSE' 16*, pages 333–344, 2016.

[25] H. Liu, Y. Wang, L. Jiang, and S. Hu. PF-Miner: A New Paired Functions Mining Method for Android Kernel in Error Paths. In *COMPSAC' 14*, pages 33–42, 2014.

[26] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT press, 1999.

[27] F. Martin et al. *Refactoring: Improving the Design of Existing Code*. 1999.

[28] M. Monperrus and M. Mezini. Detecting Missing Method Calls As Violations of the Majority Rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1):7:1–7:25, 2013.

[29] D. Movshovitz-Attias and W. W. Cohen. Natural Language Models for Predicting Programming Comments. In *ACL' 13*, pages 35–40, 2013.

[30] S. Nessa, M. Abedin, E. Wong, L. Khan, and Y. Qi. Software Fault Localization Using N-gram Analysis. In *WASA' 08*, pages 548–559, 2008.

[31] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A Statistical Semantic Language Model for Source Code. In *FSE' 13*, pages 532–542, 2013.

[32] 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 *FSE' 09*, pages 383–392, 2009.

[33] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to Generate Pseudo-code from Source Code Using Statistical Machine Translation. In *ASE' 15*, pages 824—829, 2015.

[34] M. Pradel and T. R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE' 09*, pages 371–382, 2009.

[35] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-Sensitive Inference of Function Precedence Protocols. In *ICSE' 07*, pages 240–250, 2007.

[36] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu. On the "Naturalness" of Buggy Code. In *ICSE' 16*, 2016.

[37] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *PLDI' 14*, pages 419–428, 2014.

[38] R. Rosenfield. Two Decades of Statistical Language Modeling: Where Do We Go from Here? 2000.

[39] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting Resource-release Omission Faults in Error-handling Code for Systems Software. In *DSN' 13*, pages 1–12, 2013.

[40] M. Sampson, L. Zhang, A. Morrison, N. J. Barrowman, T. J. Clifford, R. W. Platt, T. P. Klassen, and D. Moher. An Alternative to the Hand Searching Gold Standard: Validating Methodological Search Filters Using Relative Recall. *BMC Medical Research Methodology*, 6(1), 2006.

[41] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static Specification Mining Using Automata-Based Abstractions. In *ISSTA' 07*, pages 174–184, 2007.

[42] B. Sun, G. Shu, A. Podgurski, and B. Robinson. Extending Static Analysis by Mining Project-specific Rules. In *ICSE' 12*, pages 1054–1063, 2012.

[43] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments? */. In *SOSP' 07*, pages 145–158, 2007.

[44] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc Comments to Detect Comment-code Inconsistencies. In *ICST' 12*, pages 260–269, 2012.

[45] S. Thummalapenta and T. Xie. Mining Exception-Handling Rules as Sequence Association Rules. In *ICSE' 09*, pages 496–506, 2009.

[46] S. Thummalapenta and T. Xie. Alattin: Mining Alternative Patterns for Defect Detection. *Automated Software Engineering*, 18(3-4):292–323, 2011.

[47] Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. In *FSE' 14*, pages 269–280, 2014.

[48] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining Succinct and High-coverage API Usage Patterns from Source Code. In *MSR' 13*, pages 319–328, 2013.

[49] A. Wasylkowski and A. Zeller. Mining Temporal Specifications from Object Usage. *Automated Software Engineering*, 18(3-4):263–292, 2011.

[50] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting Object Usage Anomalies. In *FSE' 07*, pages 35–44, 2007.

[51] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward Deep Learning Software Repositories. In *MSR '15*, pages 334–345, 2015.

[52] C. Williams and J. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. In *TSE' 05*, volume 31, pages 466–480, 2005.

[53] C. C. Williams and J. K. Hollingsworth. Recovering System Specific Rules from Software Repositories. In *MSR' 05*, pages 1–5, 2005.

[54] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *MSR' 06*, pages 54–57, 2006.

[55] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, and M. Chandramohan. Detection and Classification of Malicious Javascript via Attack Behavior Modelling. In *ISSTA' 15*, pages 48–59, 2015.

[56] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *ICSE' 06*, pages 282–291, 2006.

[57] Z. Yu, H. Hu, C. Bai, K.-Y. Cai, and W. Wong. GUI Software Fault Localization Using N-gram Analysis. In *HASE'11*, pages 325–332, 2011.