Application of Natural Language Processing and Information Retrieval in Two Software Engineering Tools

by

Hazel Victoria Campbell

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

© Hazel Victoria Campbell, 2021

Abstract

Many software engineering problems have traditionally been approached by applying techniques based on static analysis and fixed sets of rules. I created two novel techniques to tackle three software engineering problems: typo location, fix suggestion, and crash report bucket creation. However, unlike previous techniques based on static analysis or a fixed set of rules, these techniques are based on methods commonly used to handle natural language artifacts.

Existing tools and previous work typically tries to be general and work with any valid program or theoretically possible output. In contrast, this thesis builds upon the success of prior work that successfully applied NLP models to code to improve code completion in an IDE (Integrated Development Environment). This thesis continues in that vein and presents tools that focus on the code that programmers actually write and the crashes that actually occur.

First, I applied natural-language models to locate errors in source code that cause the code to fail to compile or create an error when the code runs. Language models can adapt to coding styles and idioms. My co-authors and I showed that a tool using an n-gram model of code previously compiled successfully could supplement errors with locations produced by the Java compiler. Using our tool to suggest a location after each error message produced by the Java compiler resulted in an MRR score 11-40% closer to a perfect score than the Java compiler's score. Then, my co-authors and I showed that a similar

approach also worked with the Python interpreter, though it faced significantly more challenges. When combined with the Python interpreter's error messages, our approach correctly located an additional 9-23% of tested typos made by mutation. Next, my co-authors and I showed that the technique still worked in a more restricted offline setting. In addition, we showed that the approach could also accurately suggest changes to repair around a third of typos made by students.

I also applied the TF-IDF representation and distance function to the task of bucketing (clustering) software crash reports. In all cases, performance (in terms of F_1-score) matched or beat commonly used rule-based techniques. The TF-IDF-driven approach can adapt automatically to patterns in crash reports as they evolve. Additionally, several side benefits arose from using statistical techniques. Some errors in source code can be automatically repaired using a language model. Patterns in crash metadata can be extracted easily using a bag-of-words approach with a suitable tokenizer.

This thesis's results encourage research on approaches based on on-line off-the-shelf algorithms or models initially developed for natural-language artifacts with programming language and other software artifacts. However, this thesis's results do not necessarily guarantee that such uses will be successful; it does indicate that they should, at least, be considered.

Preface

Some of the research conducted for this thesis forms part of a research collaboration led by Professor Abram Hindle at the University of Alberta. Professor Abram Hindle also provided some of the software in Chapter 3. The experiments and data analysis procedures in Chapter 3 and Chapter 4 were designed by myself, with Professor Abram Hindle and Professor José Nelson Amaral's assistance. The experiments and data analysis procedures in Chapter 5 were designed by myself, with assistance from Eddie Antonio Santos, Professor Abram Hindle, and Professor José Nelson Amaral.

Chapter 3 has been published as Campbell et al. (2014). I was responsible for software development, data collection, data analysis, and the majority of manuscript composition. Dr. Abram Hindle provided the related work section and Figure 3.1. Professor Abram Hindle and Professor José Nelson Amaral provided editing and supervision and were involved with concept formation.

Some of the research conducted for this thesis forms part of a research partnership between the University of Alberta and BioWare, ULC, a division of Electronic Arts, led by Professor Abram Hindle at the University of Alberta. The software referred to in Chapter 7 was developed by myself with feedback from Professor Abram Hindle and BioWare.

Chapter 4 has been distributed without peer review as Campbell et al. (2015). I was responsible for software development, data collection, data analysis, and the majority of manuscript composition. Professor Abram Hindle and Professor José Nelson Amaral provided editing and supervision and were involved with concept formation.

Apart from tables, Chapter 5 has not been published elsewhere. I am the sole author of the text and algorithms presented in Chapter 5. Eddie Antonio Santos was heavily involved in concept formation, experimental design, and producing results. Professor Abram Hindle and Professor José Nelson Amaral were involved with concept formation, manuscript revision, and supervision.

Chapter 6 has been distributed without peer review as Campbell et al. (2016b). The data collection was performed by myself. Data analysis was performed by myself with assistance from Eddie Antonio Santos. Professor Abram Hindle was involved with concept formation, manuscript revision, and supervision.

Chapter 7 has been published as Campbell et al. (2016a). The data collection and data analysis were performed by myself. Manuscript composition, software development, and data analysis were performed by myself with Eddie Antonio Santos's assistance. Professor Abram Hindle and BioWare, ULC, were involved with concept formation. Professor Abram Hindle was involved with manuscript revision and supervision.

Chapters 1, 2, and 8 were written entirely by me and have not been published elsewhere. Professor Abram Hindle, Professor José Nelson Amaral, and Professor Sarah Nadi provided feedback for manuscript revisions.

To my cat, Senator Whiskers

For being there for me throughout my studies and my research.

Very nice!

— Peter Norvig, 2015.

Acknowledgements

The work for chapters 6 and 7 was funded by a MITACS Accelerate Cluster with BioWare, ULC.

The work for chapters 3 and 4 was funded by University of Alberta PhD recruitment and early achievement scholarships, and an NSERC Discovery grant.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

> It is a truism that most bugs are detected only at a great distance from their source. — Wand, 1986

1.1 Two Engineers

This thesis centers around two tasks: bucketing (clustering) automatically generated and gathered crash reports and the task of helping developers fix minor mistakes and typos in their code. This chapter starts with two stories concerning two software engineers, Margaret, the maintainer and Debra, the developer, to motivate, provide background, and ground this thesis with concrete examples.

Margaret is a maintainer. Her job is to fix crashes that have been reported in a large and popular piece of software, Widget. When Widget crashes on some user's computer, it collects information about the crash and information about the environment it was running in. The collected information includes the location in the code where the crash occurred. Widget takes this information, bundles it up in a crash report and sends that crash report to Margaret.

Margaret is quickly overwhelmed by the sheer number of crash reports being collected because Widget is popular and crashes often. Margaret decides to automatically organize these crashes, grouping them by which function Widget crashed in. Such information is readily available in the crash reports. Groups of similar crashes are called *buckets*. Then, Margaret gets to work on

fixing the crashes in the bucket with the most crashes since those crashes affect users the most often. Margaret relies on the assumption that the crashing function indicates the crash's cause, so she already has two problems.

The first problem with Margaret's initial, naive approach is that multiple different bugs could be causing Widget to crash in the same function. Multiple different bugs could be causing Widget to crash on the same line of code. When different crashes caused by different bugs cause crashes in the same bucket, that bucket has poor *precision* or *purity*. Poor precision means that Margaret could miss critical bugs if she does not carefully examine each bucket. Poor precision could also cause some buckets to have unrelated crashes, which is the opposite of what Margaret wants.

The second problem with Margaret's initial approach is that a single bug in Widget's code could be causing it to crash in several different functions. Margaret wants to fix the problems causing the most crashes first, but her approach may not target those problems. When a single bug causes crashes that Margaret's approach sorts into different buckets, those buckets have poor *recall* or *inverse purity*. Buckets with poor recall could hide the actual number of crashes caused by a single bug from Margaret by spreading out the crashes caused by that bug, making it seem like it caused many fewer crashes than it actually did.

Later, Margaret notices that Widget often crashes in an error-handling routine. To handle that, Margaret adds a rule to her crash grouping tool that ignores this routine and instead uses whatever function invoked the error handler to group the crashes. Now Margaret has a third problem, adding rules to her tool to deal with different circumstances as they come up. Adding rules to her tool uses Margaret's time: time that could be spent fixing bugs. The necessity of creating hand-tuned rules to improve bucketing as Widget evolves increases the maintenance effort required to continue using Margaret's naive tool over time.

This thesis aims to help Margaret by giving her a tool, PartyCrasher, that produces buckets with improved precision and recall while not requiring Mar-

garet to maintain a list of hand-written rules. PartyCrasher should also help Margaret find informative patterns among her crash reports.

This thesis also aims to help our second software engineer, Debra, the developer. Debra is working on the next version of Widget. Debra follows the familiar pattern of making incremental changes and then compiling and possibly testing them. Each time she makes a change and compiles a new version of Widget, Debra may feel the changes she made since the last time she committed are not significant enough to commit the new version. Debra may not commit until finishing a series of minor changes if she has a larger goal in mind.

For example, Debra may be tasked with adding a new button to Widget that runs a new method. To do so, she might first write the method in one source file, and ensure that it compiles before adding the code to display the button in another source file, testing the button, and committing the code. In this scenario, Debra committed both changes simultaneously. Thus, the version control system she uses will not record the intermediate state of the code between commits.

When Debra makes a small mistake, a typo, or a stray character ends up somewhere in the code, the compiler outputs an error message. If the compiler's error message gives Debra a suggestion or a good idea of how to fix the problem, then Debra can quickly fix it and move on. If the compiler's error message gives Debra an accurate location of the problem, that is almost as good: at least Debra knows where to look. However, if the compiler's error message describes the problem poorly or points to the wrong location, then she must look through the code herself and try to find the problem.

To make this example even more concrete, consider one instance of Debra making a typo. In this instance, Debra is writing a simple loop. Debra usually names loop counter variables i, but one time Debra made a mistake in her typing and accidentally named her loop counter variable o instead. (The keys are right next to each other on a QWERTY keyboard.)

Naming a loop counter variable *o* is a perfectly valid thing to do in most programming languages. Thus, the compiler will not report the error's location where Debra named a loop counter variable *o*. Instead, the compiler will produce an error message with the location of the first usage of *i* because *i* has not been defined, and using an undefined variable is invalid.

This example demonstrates when a compiler's error message is certainly *correct* but not *helpful* for Debra. It would be easier for Debra if the compiler told her the location of her mistake so that she can fix it immediately. Instead, Debra must work backward by first imagining or recalling what kinds of mistakes she could have made that would cause the compiler to complain about an undefined use of *i* and then searching through her code for such a mistake.

However, in this case, the compiler is not being as helpful as Debra would like either, because the actual location of the mistake is where she accidentally defined *o*. The compiler will not complain about this location, because naming a loop counter variable *o* is perfectly valid.

This thesis aims to help Debra by arming her with a tool, UnnaturalCode, that can, in combination with the compiler, provide Debra with a solution or the problem's location more often than the compiler alone can. Ideally, the tool would be able to use patterns in Debra's old code to help Debra fix her new code. By considering not only the single revision of the code being compiled but other versions as well, the tool would adapt to the Widget codebase, coding conventions, and Debra's style choices when writing code and use that information to help Debra fix problems.

1.2 Motivation

With Margaret and Debra from the previous section in mind, this section will motivate helping them on a more abstract level. Then, I will discuss how tools that can help Margaret and Debra are connected, although they are in two very different software engineering roles.

The motivation for this thesis is to reduce manual labour in both fixing source code with typos and in bucketing crash reports. Another motivation is to exploit information that is available to tools working with source code and crash reports.

This thesis is focused on reducing manual labour in two different scenarios, one at the small scale and one on a much larger scale. On the small scale, Chapters 3, 4, and 5 try to reduce the manual labour for an individual software developer by focusing on the very act of writing program code and problems that comes with it: small mistakes, typos, and spelling errors. On the large scale, Chapter 7 aims to reduce manual labour for software developers working on fixing crashes in large, popular pieces of software.

Collecting and organizing crash reports is economically motivated, as demonstrated by the existence of multiple companies that offer crash-report collection and organization services such as Raygun (Raygun, 2020), Airbrake (Airbrake, 2020), Rollbar (Rollbar, 2020), and BugSplat (BugSplat, 2020).

This work also attempts to ground the way tools deal with software artifacts in statistical reality. A compiler will compile any valid program, no matter how unlikely that program is. The space of valid programs is infinite. The space of programs that a human programmer has or will or might write is far smaller. This thesis focuses on tools that deal with the smaller space. This thesis also focuses on tools that adapt to programming style and context (for example, a specific software product) by taking into account the code that has actually been written by the programmers in that context.

A similar situation exists with tools for bucketing crash reports. They are built with the assumption that they need to handle any kind of crash anywhere in the software's code. Consider a simple and common rule for bucketing crash reports: if two crash reports show the software crashing in the same location in the code, they are placed in the same bucket. Compare that to the ideal output: if two crash reports were caused by the same bug, they will end up in the same bucket. The rule only works if the causes (the bugs) match up one-to-one with the effects (the crash locations).

Like the compiler, the simple rule for bucketing crash reports only considers every possible crash location individually. But developers charged with fixing crashing software based on crash reports typically only care about crashes that have actually happened. Otherwise, they would be using proactive tools, such as static analysis tools and proofs, rather than reacting to crash reports. Thus they may be better served by tools that take into account what crashes have actually happened and the patterns in those crashes.

1.3 The Road to Data-driven SE Tooling

Software engineers, in practice, commonly approach problems relating to the practice of engineering software itself with familiar tools based on rules. For dealing with problems during compilation, they add rules to the compiler to help locate problems. When receiving reports of problems during the operation of software, they use rules to divide these crashes into clusters. Statistical approaches, however, have enjoyed great success in the realm of natural language processing and information retrieval from natural-language text. Statistical approaches have been overlooked in software development and maintenance tooling.

Software artifacts such as source code and crash reports are very structured. Without considering software-related artifacts written by people for other people to read such as documentation and bug reports, software artifacts are either produced by software or consumed by software. For example, a crash report is produced by software (a debugger) and a program is consumed by software (a compiler). Compilers and debuggers are based on rules and exact sequences of steps. It is natural to build tools for working with such structured software artifacts using related rules and sequences of steps.

In comparison, natural language artifacts, such as news articles, books, and personal messages are far less structured than software artifacts. Thus, tools that rely on consistent structure are impractical. Tools for working with natural language artifacts typically rely on modelling a corpus of example

artifacts. For example, a search engine might count the number of artifacts a word appears in, and a language model might count the number of of times a sequence of words appears.

Consider a compiler. A compiler accepts every single possible input that meets the rules of the language being compiled. Compare that with a tool for finding natural language artifacts, such as a search engine, that deals solely with a collection of artifacts that already exist. For example, by focusing on code that already exists, NLP based tools can focus on the code that programmers have written, instead of any possible program that might be valid in a particular programming language.

The focus on rule-based approaches to dealing with software artifacts neglects the wide array of techniques that have been developed for dealing with natural language artifacts. Despite tradition, there is no reason that these approaches cannot be combined with or replaced by more dynamic, statistically-driven approaches that have been developed for handling natural-language artifacts.

Rule-based approaches also have a tendency to neglect additional information that may be available. Consider, for example, a computer programmer who invokes a compiler on a piece of a source code that they have just modified. If the source code compiles, then all is well. However, if the source code does not compile, then the compiler will produce some error and often a recommendation for how to fix the code. The compiler only considers that single version of the source code during compilation. However, there is more information available: the previous versions of the same code.

As a second example, consider a crash report. Crash reports usually contain a wealth of information about not only the state of the program as it was crashing but the environment that was running in. This information is included because it is useful to the human engineers that are responsible for reading crash reports and fixing the software that crashed. Typically, rule-based approaches to organizing crash reports have focused on one piece of information only: where in the program's code it crashed.

Unfortunately, rules-based approaches do not always work perfectly. It is impossible to devise a set of rules that covers every possible future situation. Techniques using dynamic, statistical approaches that can adapt to situations as they arise and use all of the information available may work better than hand-crafted sets of rules.

Rule-based approaches do not take into account any historical information. However, crashes do not happen independently. If a line of code crashes, it is likely that the same line will crash again, even after the bug responsible has been fixed (Seo and Kim, 2012b). Thus, it is reasonable to build tools for organizing crashes that take into account the crashes they have already seen: a Bayesian approach. A combination of both a rule-based approach and a statistical approach may be able to combine the best of both worlds.

When a working compiler produces an error, in one sense, it is never wrong: the program does not meet the rules of that programming language. Or, in the rare case the compiler is wrong, it is considered broken and in need of repair.

By their very nature, the rules that define a language define only what is considered valid source code for that language. This means that they are not focused on addressing source code that is almost, but not yet valid. Statistical approaches, on the other hand, are well-suited for dealing with artifacts that almost follow the rules.

The error messages the compiler produces are based on rules as well, and they are not always helpful. The compiler is doing its job as it was designed, taking valid programs and compiling them, or producing an error. However, the errors a compiler produces do not always meet the human needs of the software developer invoking compilation. Becker et al. (2019) states:

> [Finding the location] of errors is a particularly difficult problem. Syntax errors such as missing semicolons or curly braces may be detected at a later point than the source of the error, and error-recovery algorithms in parsers can further confuse users by producing messages referring to spurious errors.

What a human software developer wants, ideally, is for the compiler's error message to simply tell them how to fix whatever the problem is that caused the error. What they need is, at least, some information that will help them to discover how to fix their program on their own. However, as shown in chapters 3 and 4 sometimes the compiler's error message does not even mention the location of the problem in the source code.

A technique that takes advantage of other source of information that are commonly available may be able to help improve compiler error messages. Taking past compilations into account is a Bayesian approach that may be able to address cases that are handled poorly by rule-based approaches.

The task of improving compiler error messages can be broken down into pieces. The first piece is correctly identifying the location of the problem in the code more often. Compilation errors caused by mistakes in the code being compiled almost always come with a location: a line and column number in the code. Some compilers, such as the Java compiler, report more than one error, and thus more than one location at a time. The accuracy of the locations in the errors produced by such a compiler can be measured by employing mean reciprocal rank (MRR). MRR is a measure typically used with search engines. MRR is a weighted average of how many results need to be considered before reaching the correct result. In the case of compiler error messages, MRR is a weighted average representing how many errors need to be considered, and those error's locations in the code need to be inspected before the location of the actual problem in the code is inspected. For other compilers that only produce one error at a time, we can simply consider how often the location in the error is the correct location. The fraction of locations that are correct is equal to MRR when there is only one location.

Now, consider the task of grouping crash reports. Typically, this is done with a fast, rule-based approach. These approaches are labour-intensive, requiring new rules to be hand written when the old rules do not handle new patterns in the crash reports. These patterns could be derived statistically. A

statistical approach might eliminate the need for adding hand-written rules to deal with new patterns in the crash reports.

Grouping crash reports is a clustering problem. Measures designed for clustering algorithms such as BCubed F_1-score measure how closely the clusters produced by an algorithm match ideal clusters. Crash reports are grouped or clustered into "buckets." Ideally, each bucket of crash reports would contain all of the crashes caused by a single problem (for example, crashes caused by a single bug) and no crash reports caused by other problems.

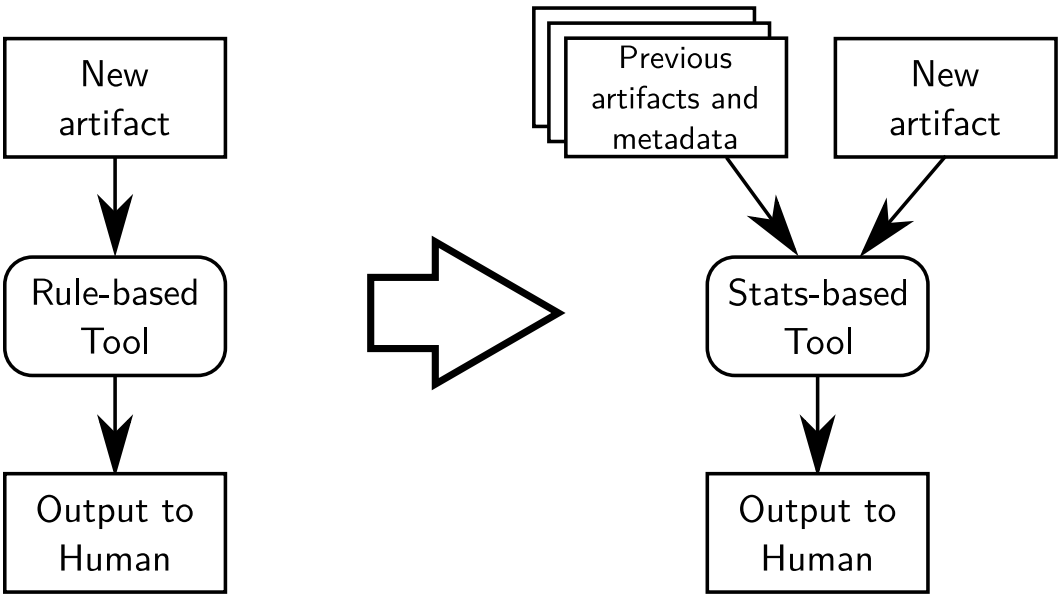

Figure 1.1: High-level representation

Figure 1.1 shows the general approach this thesis takes to creating tools to perform both tasks: bucketing crash reports and locating typos. This thesis provides tools based on the same high-level formulation to both Margaret and Debra.

1.3.1 Thesis Statement

An n-gram language model can be used to find errors in source code resulting in more precise reports of the location of errors. Term Frequency–Inverse Document Frequency (TF-IDF) representation can be used to cluster crash

reports automatically, leading to clusters that are more representative of the bugs that caused the crashes.

In order to evaluate the above thesis statement, I will use the MRR score to measure how precise reports of the locations of errors are and the B-Cubed F_1-score to measure how representative buckets are of the bugs that caused crashes.

1.4 Research Presented in this Thesis

Initially, the work presented in this thesis was inspired by Hindle et al. (2012). That work made use of a model traditionally used for natural language texts, the n-gram model, to improve code completion suggestions in an integrated development environment (IDE). That work lead to using techniques, specifically, n-gram models that come from the realm of natural-language processing (NLP) to improve tools.

First, a technique similar to Hindle et al. (2012) was used to improve the error messages produced by the Java compiler. The new technique focused on improving the locations (line and column number) the compiler reports for errors caused by typos and small mistakes. This work was published and is Chapter 3 in this this thesis. Results were obtained using a very rough-and-ready set of scripts in combination with with the n-gram software that had been used in Hindle et al. (2012) and the Java compiler.

The work on the Java compiler's error messages was then extended to the Python interpreter's error messages. This was quite a leap because Python is typically an interpreted language, so some typos, such as misspelled identifiers are not reported until the code is run. Thus, testing the same technique that worked with Java code was much more difficult to test with Python code. In order to test the use of n-gram models with python code, a new modular testing framework and tool was implemented. The new tool, UnnaturalCode, was designed to be extendable to multiple programming languages and multiple language models. This work is presented in Chapter 4.

Work in collaboration with a Master's student, Eddie Antonio Santos, began to compare the performance of the n-gram language model and a language model based on neural networks. It became apparent that not only could these models improve the locations in error messages, but that they could be extended to automatically repair programs with small mistakes such as typos. The language models could actually be used to turn code that did not compile or did not run into code that did compile or run. In addition, we gained access to a data set composed of real mistakes made by students to test our techniques against. This work was published and is presented in this thesis as Chapter 5.

Some of the research in this thesis was funded by a MITACS Accelerate grant to work with a local software company. The company was dealing with the problem of receiving far too many crash reports to consider individually. In order to help them manage the crash reports, a similar approach to the one that had worked for compiler errors was developed and tested: using traditional techniques usually applied to natural language text. Specifically, TF-IDF was chosen, a representation typically employed by search engines. In order to test this approach, a data set was gathered and is presented in detail in Chapter 6. The TF-IDF-based approach was published and is presented in this thesis as Chapter 7.

1.5 Contributions

This thesis shows that rule-based tools for dealing with software artifacts can be combined with or improved by statistics-based tools that take into account more than one artifact at a time.

The research presented in Chapter 3 was the first to use a technique not based on rules and static analysis to improve compiler error locations. It was also the first to do so with a technique that takes into account other source code that had been compiled previously.

The n-gram-model technique for locating mistakes in code initially required the compiler to first determine whether or not a program was valid. However, this technique was extended to a language that is not completely compiled, Python. In this setting, despite the lack of a true compiler, the n-gram-model technique could still operate, and even help with errors that are generated at runtime in an interpreted language.

The work presented in 5 was the first to show that a simple n-gram language model performed comparably to more complicated models based on neural networks for generating code repairs. It also bridged the gap between improving compiler error messages and repairing the mistakes that created those error message in the first place.

Though researchers had created techniques for grouping crash reports that employed statistics before the work in Chapter 5 was published, Chapter 5 was the first technique from the field of information retrieval (IR) applied to the problem. Previously, techniques only used information found in the stacktrace component of a crash report. The stacktrace is a list of routines that were executing at the time the program crashed. In order to test this new technique, the data set presented in Chapter 6 was gathered. This new data set consisted of crash reports that had been grouped manually by developers familiar with the crashing software.

The following contributions are presented in this thesis:

- A tool, UnnaturalCode, that:
  - uses an n-gram language model for locating errors in compiled source code
  - and interpreted source code,
  - as well as automatically suggesting changes to repair source code.
- A data set consisting of crash reports grouped by their underlying cause.
- A tool, PartyCrasher, that:
  - uses TF-IDF to bucket crash reports,

  - can take into account metadata along with stacktraces,
  - does not require manual rule-writing,
  - is designed to[1] handle a steady stream of new crash reports.

- Descriptions, evaluations, analyses, and discussion of UnnaturalCode and PartyCrasher.

1.6 Potential Applications

The techniques and tools developed in Chapters 3, 4, and 5 could be used by IDEs (integrated development environments) to help guide the user (programmer) towards code that be could causing a problem. Most IDEs already do this by parsing code and highlighting parts that they think are causing the problem, however, they use the same or similar rules as the compiler to do so. By combining those rules with the techniques and tools presented in this thesis, they could highlight the code that is actually causing the problem more often and provide more feedback to the user. In addition, the IDE could suggest actual fixes to repair broken code.

Software telemetry is the practice of remotely monitoring software performance and faults. When telemetry determines that software is crashing repeatedly or across multiple instances, the techniques and tools from Chapter 7 are intended to help find patterns in those crashes. Patterns in crash reports could also be linked to bug reports in issue trackers.

Lessons learned from these chapters, such as software-aware token splitting, indicate that there are techniques that enable the use of NLP and IR algorithms on software artifacts not discussed in this thesis and software artifacts of the future. For instance a lot of the techniques used to group crashes could be used to group other kinds of software events such as log entries.

[1]Not achieved in worst-case scenarios without modification, see Section 7.7.

1.7 Organization

The following chapters will focus first on typos, helping developers like Debra, and then on crash reports, and my tool to assist Margaret. I will treat these two topics completely separately until the conclusion, where I will return to unnatural tools once again. However, for both typos and crash reports, I will present tools that meet the definition of an unnatural tool, and follow the "Unnatural Way," described in the next chapter.

Chapters 3, 4, and 5 focus on locating and then correcting typos. In those chapters I will demonstrate and explore a tool that demonstrates the first part of my thesis statement. Then, Chapter 6 and 7, will focus on crash reports, with Chapter 7 demonstrating the second part of my thesis statement. Finally, in Chapter 8, I will conclude, summarize, remark upon and suggest future work for typos, crash reports, and unnatural tools in general.

Chapter 2

Terminology and Philosophy

In order to be clear and concise throughout the novel portions of this thesis (the introduction, this chapter, Chapter 5 and Chapter 8), I will discuss two key terms in this chapter, followed by a more philosophical discussion on the process of creating the tools presented in this thesis.

The first term is *typos*, which is usually understood in informal conversation to indicate the sort of mistake one might make while typing (or, earlier, typesetting) English text. However, for the purposes of this thesis and computer programming languages I will provide a more concise definition.

The second term, *unnatural tools*, does not already have a different definition in other contexts. However, I find it necessary to define it here, so that I can avoid repeatedly describing the characteristics of this group of tools throughout the rest of this thesis.

Then, in Section 2.3, I will expand the definition of an unnatural tool into a strategy for creating unnatural tools.

2.1 Typos

In this introduction, chapter 5, and the conclusion (chapter 8), a *typo* refers to a specific type of flaw in source code. Typos have the following properties:

- *Smallness*: a typo spans only a single token.
- *Independence*: fixing a typo requires modification, insertion, or removal of a single token.

- *Locality*: a typo causes an error when the source code file containing it is used.

- *Obviousness*: a typo does not require consideration of the logic, intended operation, or intended usage of the program.

The usage of the term "typo" in this thesis does not exactly match its general usage. However, a "typo" is the closest match to the specific kind of flaws discussed in chapters 3, 4, and 5. The main difference between "typo" when used in this thesis and its general usage, is that in this thesis a typo includes some errors caused by ignorance and faulty reasoning. I assume that programmers intend to write code that compiles without error.

A missing brace is a typical example of a typo for the purposes of this thesis. A missing brace is small, since a brace is a single token. A missing brace is independent because it can be fixed by inserting a single token. A missing brace is local because you do not have to parse multiple files for the missing brace to cause an error. Finally, a missing brace is obvious because it causes an error during parsing, and parsing must be performed before execution.

A misspelled identifier is a critical example of a typo for the purposes of this thesis. A misspelled identifier is small, since it is a single token; it is independent, since it only requires modification of the misspelled token to the correct spelling; it is local as long as it is not the declaration of something that is used by other source code files; and it is obvious as long as the misspelling does not match the correct spelling of some other identifier. This thesis counts misspelled identifiers as typos even if the identifier is misspelled because the programmer believed that the misspelling was correct when they typed it.

Logic errors are not considered typos, even if they are the result of an errant keystroke. Consider the example of a programmer who forgot to include a unary logical negation operator such as Java's "`!`" or Python's "`not`," and this caused their program to produce incorrect output for some specific input. The missing operator in their source code would not be considered a typo for

the purposes of this thesis because determining that the code is faulty would require testing the code with a specific input, making the mistake not *obvious*.

2.2 Unnatural Tools

Before describing what makes a tool unnatural, it is worth considering what would be referred to as a natural (language) tool. Informally, a natural language tool is some piece of software that performs or assists some task involving information in a natural language, such as English. A few good examples include:

- spell checkers,
- grammar checkers,
- autocorrect,
- predictive software keyboards,
- and encyclopedia search engines.

An unnatural tool would then be, informally, a tool that fills a similar role, but for constructed languages intended for processing by machines. Specifically, this thesis focuses on computer programming languages. However, there is no reason to exclude other languages that are intended to be read by computers, such as markup languages.

However, there are several other criteria that seperate unnatural tools from other software engineering tools. These criteria mostly revolve around using similar techniques, at their core, to the natural language tools that inspired them. Additionally, I specify that they should be fast and cheap enough to be used interactively, mirroring the interactivity of some (but not all) natural language tools, such as spell checkers.

For brevity I will use the term “unnatural tools” to refer to tools that are:

- software-engineering tools;

- intended to work with source code or artifacts having some other well-defined, rigid, artificial structure;
- not intended to work with natural language;
- on-line;[1]
- and designed to assist developers with existing tasks;

yet are built on top of a model or technique that:

- was originally developed for use with natural language;
- has been studied extensively in a natural language setting;
- can be updated quickly enough to be interactive;
- has multiple off-the-self implementations; and
- estimates and uses probability distributions based on prior samples, treating "surprising" new inputs (inputs with high perplexity) differently from "unsurprising" new inputs (inputs with low perplexity).

Perplexity is a measure of how well a probabilistic model predicts a sample input. Perplexity increases as the probability of a model predicting a sample decreases. The perplexity is usually defined, for observations (samples) x_i and the model's estimated probability of an event $q(x_i)$:

$$2^{-\underset{\forall x_i}{\text{average}}(\log_2 q(x_i))}$$

To make computation easier, we use only the exponent, which is the same as the cross-entropy of N observations: $\frac{1}{N}\sum_{i=0}^{N}(-\log_2 q(x_i))$.

Unnatural tools are a subset of the more general category of tools leveraging statistics obtained by data mining software repositories, a popular topic of publication since the annual Mining Software Repositories conference began in 2004 (Hassan et al., 2004). Additionally, they are differentiated from other categories of software engineering tools such as defect prediction by their goal of assisting with existing tasks.

[1] A discussion of the benefits of working on-line can be found in Section 5.6.6.

2.3 The Unnatural Way

A general strategy for creating an unnatural tool is to first consider the task that the tool is meant to solve. Then, consider how to treat the artifacts and information available as if it were natural-language text. Next, ask yourself, "can I write a simple algorithm to reduce this task to a task that can be solved by an existing technique?" For example, if you can reduce the problem to estimating a distribution, you may wish to use a language model; or, if you can reduce the problem to a document search, you may wish to build your tool on top of a search engine. Finally, ask yourself, "can I translate the artifacts into a format that the natural-language technique will understand?" Usually this means transforming the artifact into a sequence (or bag) of word-like tokens.

Choose a model or technique that can be updated and adjusted quickly as new information becomes available to it. For example, choose a search engine that can continuously add large numbers of documents to its index with little overhead; or choose a language model that can be updated continuously and retrained quickly. In general, the model or technique should be be able to adapt to each a new artifact in less time than it took for that artifact to be generated. For example, to get all the benefits of UnnaturalCode, UnnaturalCode has to be able to update its model from a successful compile in less time than it takes a human to edit a single character of a single file!

Fast on-line algorithms are not strictly necessary. It is possible to use a slow, off-line algorithm that was originally intended for use with natural language, such as a neural network, as has been demonstrated by Santos et al. (2018) and many others. However, during software development, artifacts are created and evolve. Thus, using algorithms that can keep up with development by adapting quickly in an on-line setting is important. Source code and software-engineering artifacts are also highly contextual: they are often specific to a single organization, project, or even an individual person. Using a fast on-line algorithm allows tools to adapt to the context they are being used in and to

keep adapting over time to follow the evolution over time of the source code and software artifacts. Adapting can dramatically improve tool efficacy, which I discuss in depth for the case of typo location and repair in Section 5.6.6.

Based on this thesis, an unnatural tool will likely take these steps:

1. A piece of software produces some information.
2. The tool obtains an artifact representing that information.
3. The tool tokenizes the artifact into word-like tokens.
4. The tool either evaluates the artifact using an underlying natural-language algorithm, or updates the underlying natural-language model or statistics, or both.
5. The tool produces some output for an engineer to interpret.

For UnnaturalCode, the relevant information in step 1 is produced by the compiler and represents whether the code compiled successfully. For PartyCrasher, the relevant information in step 1 is produced by the crashing program itself and the automated crash reporting system.

In step 2, it may be necessary for the tool to collect an additional artifact such as relevant source code or debugging symbols.

Step 3 has proven to be critical, and will be discussed in Section 8.3.1.

Step 4 is about adapting the underlying natural-language algorithms to the task at hand. For example, PartyCrasher relies on an algorithm to use a document search engine to perform a clustering task. This type of code is demonstrated in algorithm 1 and the following algorithms in chapter 5.

Step 5 depends on the tool and what part of the software engineering process the tool is focused on, as well as who will be using the tool. For example, generating a short, ranked list of suggestions, rather than a table of raw probabilities is preferred. Or, perhaps, integration into an IDE would be the most desirable way to deliver output to a developer.

The definition of what makes an "unnatural tool" is new in this thesis. All of the published reports that the following chapters present were published

before the invention of the term "unnatural tool." There are also unnatural tools that predate my tools, though I could not find an appropriate term to describe them as a group. One tool that predates the tools presented in this thesis, but is certainly an unnatural tool, generates suggestions for completing code as it is typed (Hindle et al., 2012).

Chapter 3

Dynamic Detection of Typographical Errors in Java Code

3.1 Preface

3.1.1 Acknowledgements

This chapter originally appeared as a report (Campbell et al., 2014).

I, Hazel Victoria Campbell, wrote the majority of the chapter, conducted the experiments, collected data produced results and analysis, and created most of the figures. Dr. Abram Hindle wrote the related work section, provided editing, some figures, and modifications to the software used, and supervised the research. Dr. José Nelson Amaral provided editing and supervised the research.

Table 3.4 and related discussion was written exclusively for this thesis.

3.1.2 Significance

The work presented in this chapter was the first time that Natural Language Processing (NLP) techniques had been applied to the problem of detecting syntax errors. According to my research, all previous techniques relied strictly on static analysis or heuristics, and did not include any statistical, modelling, or learning component.

Unlike prior work, this work does not rely on parsing, compilation, or heuristics to locate syntax errors within a file. Instead, it only uses parsing and compilation to detect that a file does or does not compile, from then on it uses lexical analysis and natural language models to locate syntax errors. This work uses parsing and compilation only to detect that some error exists, not what or where the error is.

This work shows that statistics, modelling, or learning can have a place alongside static analysis and heuristics when reporting compilation errors to a programmer.

3.1.3 Impact

Since its original publication, the report presented in this chapter has received 50 citations as reported by Google Scholar.

3.2 Abstract

A frustrating aspect of software development is that compiler error messages often fail to locate the actual cause of a syntax error. An errant semicolon or brace can result in many errors reported throughout the file. We seek to find the actual source of these syntax errors by relying on the consistency of software: valid source code is usually repetitive and unsurprising. We exploit this consistency by constructing a simple N-gram language model of lexical source code tokens. We implemented an automatic Java syntax-error locator using the corpus of the project itself and evaluated its performance on mutated source code from several projects. Our tool, trained on the past versions of a project, can effectively augment the syntax error locations produced by the native compiler. Thus we provide a methodology and tool that exploits the naturalness of software source code to detect syntax errors alongside the parser.

3.3 Motivation

Syntax errors plague new programmers as they struggle to learn computer programming (Garner et al., 2005; McIver, 2000). Even experienced programmers get frustrated by syntax errors, often resorting to erratically commenting their code out in the hope that they discover the location of the error that brought their development to a screeching halt (Kummerfeld and Kay, 2003). Syntax errors are annoying to programmers, and sometimes very hard to find.

Garner et al. (2005) admits that "students very persistently keep seeking assistance for problems with basic syntactic details." They, corroborated by numerous other studies (Jackson et al., 2005; Jadud, 2005; Jadud, 2006; Tabanao et al., 2008), found that errors related to basic mechanics (semi-colons, braces, etc.) were the most persistent and common problems that beginner programmers faced. In fact these errors made up 10% to 20% of novice programmer compiler problems. As an example, consider the missing brace at the end of line 2 in the following Java code taken from the Lucene 4.0.0 release:

```
1 for (int i = 0; i < scorers.length; i++) {
2     if (scorers[i].nextDoc() == NO_MORE_DOCS)
5         lastDoc = NO_MORE_DOCS;
6         return;
7     }
8 }
```

This mistake, while easy for an experienced programmer to understand and fix, if they know where to look, causes the Oracle Java compiler[1] to report 50 error messages, including those in Figure 3.1, none of which mention the line with the mistake. This poor error reporting slows down the software development process because a human programmer must examine the source file to locate the error, which can be a very time consuming process.

Some smart IDEs such as Eclipse, aim to address this problem but still fall short of locating the actual cause of the syntax error as seen in Figure 3.2.

[1]Oracle's Java compiler, version 1.7.0_13, is available at
`http://www.oracle.com/technetwork/java/javase/downloads/index.htm`

Thus syntax errors and poor compiler/interpreter error messages have been found to be a major problem for inexperienced programmers (Garner et al., 2005; McIver, 2000; Tabanao et al., 2008).

```
/home/joshua/projects/lucene-4.0.0/core/src/java/org/apache/lucene/search/ConjunctionScorer.java:56:
 error: <identifier> expected
   ArrayUtil.mergeSort(scorers, new Comparator<Scorer>() { // sort the array
                    ^
                                        ... 48 errors omitted ...
/home/joshua/projects/lucene-4.0.0/core/src/java/org/apache/lucene/search/ConjunctionScorer.java:152:
error: class, interface, or enum expected
  }
  ^
[javac] 50 errors
```

Figure 3.1: Oracle Java error messages from the motivational example.

There has been much work on trying to improve error location detection and error reporting. Previous methods tend to rely on rules and heuristics (Corchuelo et al., 2002; Heeren, 2005; Hristova et al., 2003). In contrast to those methods, we seek to improve on error location reporting by exploiting recent research on the regularity and naturalness of written source code (Hindle et al., 2012). Recent research (Hindle et al., 2012) exploits n-gram based language models, applied to existing source code, to predict tokens for code completion. The effectiveness of the n-gram model at code completion and suggestion is due to the repetitive, and consistent, structure of code.

Using this exact same model, and exploiting the natural structure of source code, we can improve error location detection. The idea behind this new method is to train a model on compilable source-code token sequences, and then evaluate on new code to see how often those sequences occur within the model. Source code that does not compile should be surprising for an n-gram language model trained on source code that compiles. Intuitively, most available and distributed source code compiles. Projects and their source code are rarely released with syntax errors, although this property may depend on the project's software development process. Thus, existing software can act as a corpus of compilable and working software. Furthermore, whenever a source file successfully compiles it can automatically update the training corpus because the goal is to augment the compiler's error messages. Changes to the software might not compile. When that happens, locations in the source

file that the model finds surprising should be prioritized as locations that a software engineer should examine.

The following sections of the chapter examine the feasibility of this new method by testing its ability to locate errors in a variety of situations, as an augmentation of compiler error reporting. These situations include situations that are much more adverse than expected in practice.

The main contributions of this work include:

- A method of statistical, probabilistic syntax-error location detection that exploits n-gram language models.
- A prototype implementation of an n-gram language-model-based Java syntax error locator, UnnaturalCode,[2] that can be used with existing build systems and Java compilers to suggest locations that might contain syntax errors.
- A validation of the feasibility of the new syntax error location detection method.
- A validation of the integration of the new method with the compiler's own methods.
- A modified version of `MITLM`[3], with routines developed by the authors to calculate the entropy of short sentences with respect to a large corpus quickly.

3.4 Background

An n-gram language model at its lowest level is simply a collection of counts. These counts represent the number of times a phrase appears in a corpus. These phrases are referred to as n-grams because they consist of at most n words or tokens. These counts are then used to infer the probability of a

[2]UnnaturalCode is available at `https://github.com/hazelybell/unnaturalcode`

[3]The modified `MITLM` package used in this chapter is available at `https://github.com/hazelybell/MIT-Language-Mode{l}ing-Toolkit`

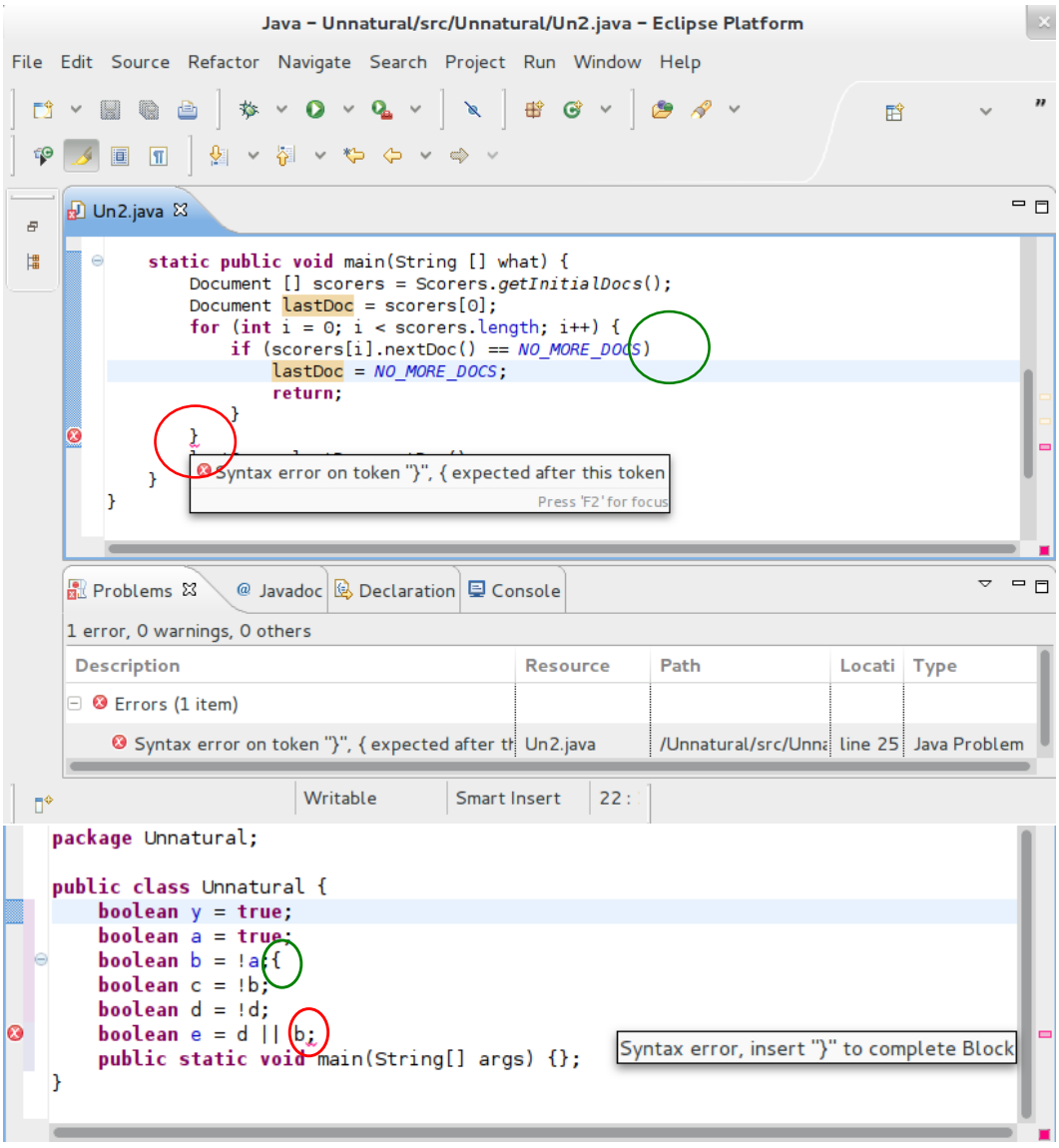

Figure 3.2: Top: Figure 3.1's code snippet put into Eclipse; bottom: another example of failed Eclipse syntax error location detection. Eclipse often detects syntax errors, but often reports them in the wrong location. Eclipse's suggested error location is circled in red, and the actual error location is circled in green.

phrase: the probability is simply the frequency of occurrence of the phrase in the original corpus. For example, if the phrase "I'm a little teapot" occurred 7 times in a corpus consisting of 700 4-grams, its probability would be .01. However, it is more useful to consider the probability of a word in its surrounding context. The probability of finding the word "little" given the context of "I'm a _____ teapot" is much higher because "little" may be the only word that shows up in that context — a probability of 1.

These n-gram models become more accurate as n increases because they can count longer, more specific phrases. However, this relationship between n and accuracy is also problematic because most n-grams will not exist in a corpus of human-generated text (or code). Therefore most n-grams would have a probability of zero for a large enough n. This issue can be addressed by using a smoothed model. Smoothing increases the accuracy of the model by estimating the probability of unseen n-grams from the probabilities of the largest m-grams (where $m < n$) that the n-gram consists of and that exist in the corpus. For example, if the corpus does not contain the phrase "I'm a little teapot" but it does contain the phrases "I'm a" and "little teapot" it would estimate the probability of "I'm a little teapot" using a function of the two probabilities it does know. In UnnaturalCode, however, the entropy is measured in bits. Entropy in bits is simply $S = -\log_2(p)$, where p is the probability. The higher the entropy, therefore, the lower the probability and the more surprising a token or sequence of tokens is.

Based on the entropy equation, as probability approaches 0, entropy approaches infinity. An uncounted n-gram could exist, which would have 0 probability effectively cancelling out all the other n-grams. Thus we rely on smoothing to address unseen n-grams.

UnnaturalCode uses Modified Kneser-Ney smoothing as implemented by `MITLM`, the MIT Language Model package (Hsu and Glass, 2008). Modified Kneser-Ney smoothing is widely regarded as a good choice for a general smoothing algorithm. This smoothing method discounts the probability of n-grams based on how many m-grams (where $m < n$) it must use to estimate

their probability. Probability can be estimated for a 7-gram from a 3- and a 4-gram; in this case the probability will not be discounted as heavily as when probability must be estimated from seven 1-grams. Modified Kneser-Ney smoothing is tunable: a parameter may be set for each discount. In UnnaturalCode these parameters are not modified from their default values in `MITLM`.

While `MITLM` and the entropy estimation techniques implemented within `MITLM` were designed for natural-language text, UnnaturalCode employs these techniques on code. Hindle et al. (2012) showed that code has an even lower entropy per token than English text does per word, as shown in Figure 3.3. That is to say, the same techniques that work for natural English language texts work even better on source code. Moreover, syntactically invalid source code will often have a higher cross-entropy than compilable source code given a corpus of only syntactically valid source code. Therefore, defective source code looks *unnatural* to a natural language model trained on compilable source.

3.4.1 Previous Work

Previous publications addressing this issue fall into two categories. Most work has focused on parser-based (Burke and Fisher, 1987) or type-based static analysis (Heeren, 2005; Lerner et al., 2007). There have also been heuristic analyzers that work alongside the parser such as the one presented in Hristova et al. (2003), but it is limited to a specific selection of common mistakes. Many modifications for particular parser algorithms have also been proposed to attempt to suppress spurious parse errors by repairing or resuming the parse after an error. Examples can be found in Corchuelo et al. (2002), Graham et al. (1979), and Kim and Choe (2001). In comparison, the new system proposed here requires no additional information about a language other than its lexemes, which may be extracted from its implementation or specification. A major difference between UnnaturalCode and those works is that UnnaturalCode does not attempt to parse the source code. There is also a more recent publication by Weimer et al. (2009) that uses Genetic Algorithms to mutate

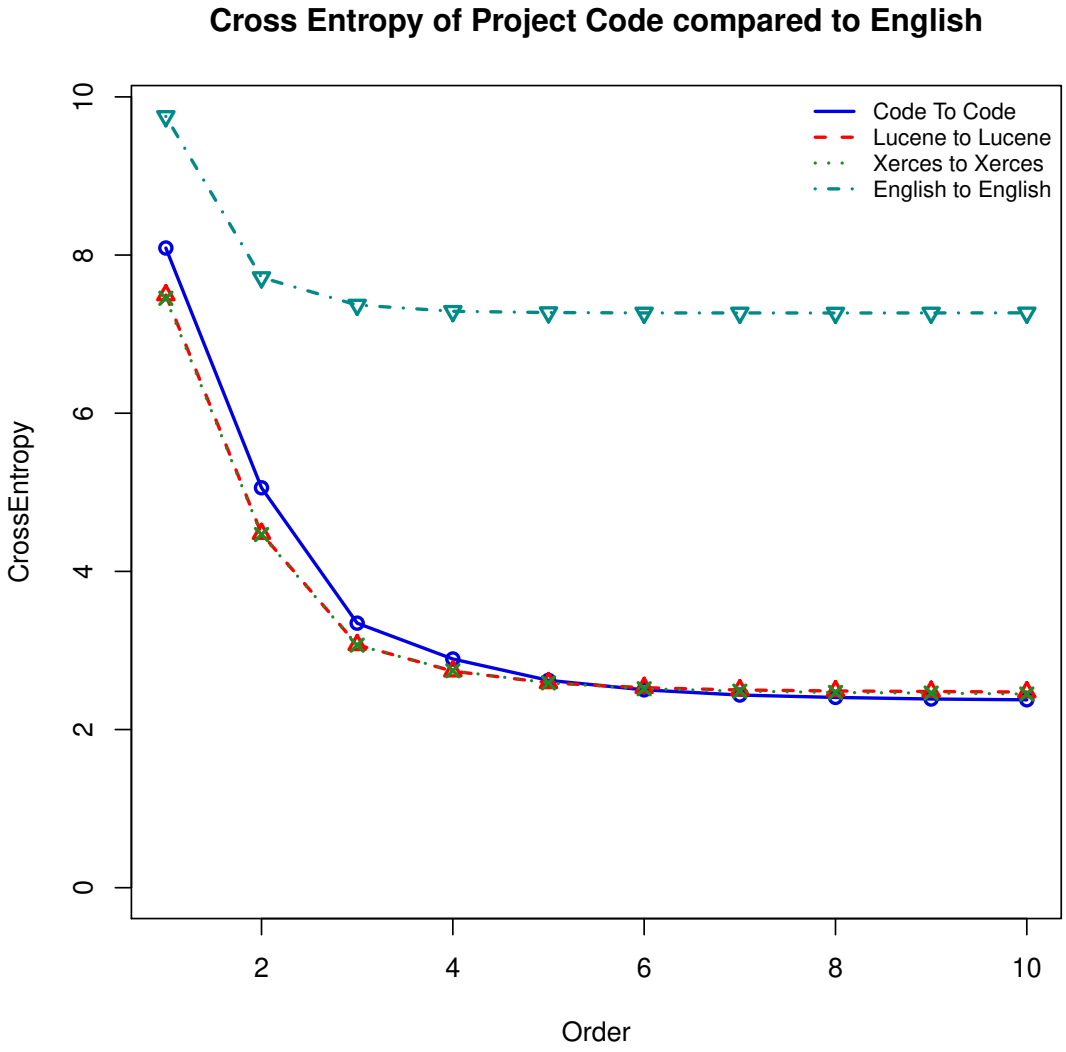

Figure 3.3: From Hindle et al. (2012), a comparison of cross-entropy for English text and source code vs gram size, showing that English has much higher cross-entropy than code.

parse trees in an attempt to fix defects, but this requires syntactically valid source code.

The frequency and importance of syntax errors among novice and experienced programmers has been studied by numerous authors (Garner et al., 2005; Jackson et al., 2005; Jadud, 2005; Jadud, 2006; McIver, 2000; Tabanao et al., 2008). Many of these studies evaluated new programmers (often undergraduate first year students) and have examined the frequency of syntax errors and other kinds of errors. According to these studies missing semi-colons and misplaced braces cause between 10% to 20% of the errors that novices experience (Jackson et al., 2005; Jadud, 2005; Jadud, 2006). Jadud Jadud (2005) and Jadud (2006) took over 42000 snapshots of first year undergraduate code before each compile and labelled the cause of compilation failure. They found that 60% of syntax errors were immediately solved within 20 seconds, but 40% took longer, 5% taking longer than 10 minutes. Tabanao (Tabanao et al., 2011; Tabanao et al., 2008) studied student performance correlated with syntax errors, for more than 120 students, and found that the number of errors and frequency of compilation negatively correlated with student midterm grades. Jackson Jackson et al. (2005) observed 559000 errors produced by 583 students during 1 semester; the second most common error was missing semicolons. Finally Kummerfeld and Kay (2003) studied the effect of student programming experience on syntax errors and found that experienced users relied on strategies to solve syntax errors; when these strategies failed, the experience programmers made erratic modifications just like their inexperienced counterparts. Thus we can see from numerous sources that syntax errors are a common source of errors that novice and experienced programmers run into and resolving these errors can often consume a lot of time and effort.

Previous publications that attempt to improve syntax error messages and syntax error location fall into two categories: *parser-based* or *type-based*. Burke's parse action deferral (Burke and Fisher, 1987) is a *parser-based* technique that backs the parser down the parse stack when an error is encountered and then discards problematic tokens. Graham et al. (1979) implemented a

system that combined a number of heuristic and cost-based approaches including prioritizing production rules to be resumed. Many modifications for particular parser algorithms have also been proposed to attempt to suppress spurious parse errors by repairing or resuming the parse after an error. Recent examples can be found in Kim and Choe (2001) where the k-nearest neighbour algorithm is applied to search for repairs, or Corchuelo et al. (2002) where a modification is presented that can be applied to parser generators and does not require user interaction. Other researchers have focused on *type-based* static analysis such as Heeren's Ph.D. thesis (Heeren, 2005), which suggests implementing a constraint-based framework inside the compiler. Lerner et al. (2007) use a corpus of compilable software to improve type error messages for statically typed languages. There have also been heuristic analyzers that work alongside the parser such as the one presented in Hristova et al. (2003), but this approach is limited to predefined heuristic rules addressing a specific selection of common mistakes. In comparison, UnnaturalCode requires no additional information about a language other than its lexemes, which may be extracted from its implementation or specification. The implementation presented here differs from those works in that it does not attempt to parse the source code. There is also a more recent publication, (Weimer et al., 2009) that uses Genetic Algorithms to mutate parse trees in an attempt to fix defects, but this requires syntactically valid source code.

3.5 A Prototype Implementation of UnnaturalCode

UnnaturalCode is designed to assist the programmer by locating the coding mistakes that caused a failed compilation. To this end, it only provides suggestions if the compile fails. If the compile succeeds it adds the error-free code to its corpus automatically. This allows UnnaturalCode to adapt rapidly to changing codebases. The data-flow diagram of UnnaturalCode is depicted in Figure 3.4.

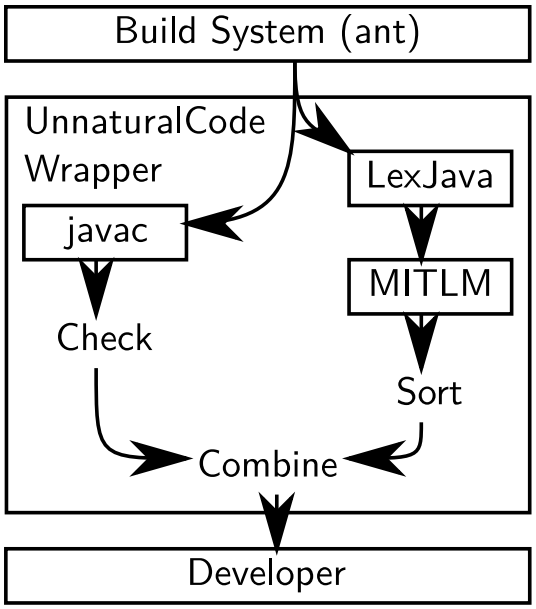

Figure 3.4: Data flow of the syntax-error location detector, UnnaturalCode.

The central component of the system is `javac-wrap.pl`, a wrapper script that is inserted in between the build system and the compiler, `javac`. Our tests were run with the `ant` build system and the Oracle 1.7.1 JDK `javac` Java compiler. Unfortunately, the package's `build.xml` file, which instructs `ant` how to build the package, must be modified to instruct `ant` to call `javac-wrap.pl` instead of merely instantiating the Java compiler classes from within `ant`. Furthermore, `javac` is called with a list of all java files to be compiled in a single run. This list often contains hundreds of source files. The first thing that `javac-wrap.pl` must do in the case of a failed compilation is to locate the file that failed to compile. It does this using heuristic regular expressions on `javac`'s error output.

In the case of a failed compilation, UnnaturalCode lexically analyzes the source file that failed to compile, and queries the smoothed n-gram model using a sliding window of length $2n$ tokens. The model is built using successfully compiled code from previous compilations of the same project. Both the query code and the corpus has comments removed and contiguous whitespace reduced to a single space.

Once `javac-wrap.pl` detects a failed compilation and locates the offending file, it must lexically analyze that file. No parsing is done, only lexical analysis is performed. In UnnaturalCode, comments are first removed from the input and then lexical analysis is performed by an ANTLR-produced Java lexical analyzer called `LexJavaMQ`. `LexJavaMQ` was originally written for Hindle et al. (2012) for use as a lexical analyzer server that any number of clients may connect to in order to analyze Java code. `javac-wrap.pl` and `LexJavaMQ` communicate over the network using the ZeroMQ library and protocol. `LexJavaMQ` tokenizes its input and returns a list of tokens as they appear in the original source. It does not return any meta-information about the tokens such as their type or location.

In the case of a failed compilation, once `LexJavaMQ` returns the lexically analyzed source back to `javac-wrap.pl`, `javac-wrap.pl` starts an instance of `MITLM`. It then sends `MITLM` a series of queries that are sequences of tokens from the lexically analyzed source file. Queries are generated by sliding a window of length $c \cdot n$ tokens. Therefore, there are at most $l - c \cdot n$ queries for a single file, where n is the gram size of the n-gram model being used in `MITLM`, and c is the chunk size used by `javac-wrap.pl`.

The prototype implementation of UnnaturalCode uses a modified version of the `MITLM` package that was first modified for use in Hindle et al. (2012) and then further modified for the system presented here. It has been modified to compute the entropy of a small sample of text in relation to a corpus. It has also been modified to do so without requiring a restart and for higher performance.

Once UnnaturalCode has finished computing results for all the queries, it ranks them by the entropy per token of each query, as in Figure 3.5. It then reports the top five strings with the highest entropy to the user as suggestions of where to look for mistakes in the code. The entropy is a measure of how unlikely the presented string of $2n$ tokens is, given the corpus. In this case, $n = 10$, a setting that works well while also keeping memory use low enough for machines with 512MB of memory.

The entropy, S, is calculated by `MITLM` in bits as the negative logarithm of the probability. The higher the entropy score, the less likely a given string of tokens was found to be by `MITLM`. Figure 3.3 shows that entropy per token values are typically between 2 and 3 bits, lower than English text, which typically has entropy near 7 bits per word.

```
Check near == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS; return
With entropy 4.552985
Check near () == NO_MORE_DOCS ) lastDoc = NO_MORE_DOCS
With entropy 4.498802
Check near NO_MORE_DOCS) lastDoc = NO_MORE_DOCS; return;
With entropy 4.244520
Check near ) lastDoc = NO_MORE_DOCS; return; }
With entropy 4.183379
Check near ) == NO_MORE_DOCS) lastDoc = NO_MORE_DOCS;
With entropy 3.858807
```

Figure 3.5: Example of UnnaturalCode output showing an accurate error location from the motivational example.

The current prototype of UnnaturalCode has several drawbacks. One problem is that during the lexical-analysis process, `LexJavaMQ` loses track of what line each token is on, thus `javac-wrap.pl` cannot generate line numbers along with the suggestions, however this is very easy to remedy. Another problem is that UnnaturalCode relies on at least three processes written in three different languages: `javac-wrap.pl`, written in Perl, `LexJavaMQ`, written in Java, and `MITLM`, which is written in C++. IPC between these three processes is often fragile. Furthermore, the system is not easily installed and run due to its many components and dependencies.

The current implementation is fast enough to be used interactively by a software engineer. Building the corpus takes much less time than compiling the same code, because only lexical analysis is performed. For the Lucene 4.0.0 corpus, results for a broken compile, in the form of suggestions, are found for a source file with over 1000 tokens in under 0.02 seconds on an Intel i7-3770 running Ubuntu 12.10 using only a single core and under 400MiB of memory. This is more than fast enough to be used interactively if `MITLM` is already running. However, `MITLM` start-up can be quite slow depending on the size of the corpus. For the Lucene 4.0.0 corpus `MITLM` takes about 5 seconds to start on the same platform.

Table 3.1: Validation Data Summary Statistics

n-gram order	10
Files in Lucene 4.0.0	2 866
Files in Lucene 4.1.0	2 916
Files in Ant 1.7.0	1 113
Files in Ant 1.8.4	1 196
Files in Xerces Java 2.9.1	716
Files in Xerces Java 2.10.0	754
Files in Xerces Java 2.11.0	757
Types of mutation	3
Mutations per type per file/hunk	≥ 120
Tests using Lucene 4.0.0 corpus	2 626 940
Tests using Ant 1.7.0 corpus	833 960
Tests using XercesJ 2.9.1 corpus	1 547 180
Total Tests Performed	5 008 080

Since UnnaturalCode is not a parser, it is not necessary for the model to receive an unmatched `}` to detect a missing `{`. UnnaturalCode instead relies on nearby contextual information such as `public static void main(String[] args)`. Therefore, the length of a body, block, parenthetical expression or other balanced syntactic structure is irrelevant despite the limited 20-token sliding window that UnnaturalCode operates with.

3.6 Validation Method

UnnaturalCode was tested primarily on three Apache Foundation projects: Lucene,[4] Ant,[5] and XercesJ.[6]

For each experiment the training corpus consisted of every Java source file from the oldest version of a single project. These source files were compiled and all successfully compiled files were added to the training corpus. This corpus was used for the duration of testing. No automatic updates to the training corpus were performed during testing. UnnaturalCode was also tested against the contiguous parts (hunks) of the diffs of the revisions of Lucene between

[4]Apache Lucene is available at `https://lucene.apache.org/`
[5]Apache Ant is available at `https://ant.apache.org/`
[6]Apache XercesJ is available at `https://xerces.apache.org/xerces-j/`

4.0.0 and 4.1.0. These hunks represent the changes that a product sees over time. Three different types of mutation tests on four different kinds of input source file were performed.

The following mutations were applied to files and relevant *diff hunks* (contiguous lines added in a patch):

- **Random Deletion:** a token (lexeme) was chosen at random from the input source file and deleted. The file was then run through the querying and ranking process to determine where the first result with adjacent code appeared in the suggestions.
- **Random Replacement:** a token was chosen at random and replaced with a random token found in the same file.
- **Random Insertion:** a location in the source file was chosen at random and a random token found in the same file was inserted there.

The resulting mutant files were actually compiled, and when compilation succeeded the mutant file was skipped. This had particularly dramatic results on the deletion tests, where 33% of the random token deletions resulted in a file that still compiled.

After compilation, the compiler's own error messages were considered. These error messages were also given a score for each file. The compiler was scored in a similar fashion to UnnaturalCode: the first result produced by the compiler mentioning the correct line number was considered correct.

Each of the three mutation tests was repeated on each input file at least 120 times, each time modifying a newly and randomly chosen location in the source file. For Lucene, all 3 tests were performed at least 120 times each on 1266 files. Millions of tests were run on Ant and XerecesJ as well. Thus, a total of over 5 million data points were collected as shown in Table 3.1.

For Lucene, 4 different kinds of source-code inputs were tested. First, for the Lucene 4.0.0 test, source files were taken from the exact same package as the corpus and were modified by the above process and then tested. These

source files exist unmodified in the corpus. Second, source files were taken from the next Lucene release, the 4.1.0 version, that had been modified by developers. Some of these source files exist in their 4.0.0 form in the corpus, but have been modified by developers and then by the above process. These files are listed in the results as the "Lucene 4.1.0 – Changed Files" test. Additionally, new source files were added to Lucene after the 4.0.0 release for 4.1.0. These new files do not exist in the corpus but are related to files that did. These are listed in the results as the "Lucene 4.1.0 – New Files" test. Finally, to test files completely external to the corpus, Java source files from Apache Ant 1.8.4 were tested. Not only do these files not exist in the corpus but they are not related to the files that do, except in that they are both Apache Foundation software packages.

In order to get the above results, the following steps were performed. First a corpus was created from the earliest release. For example, Lucene 4.0.0 was built, automatically adding all compilable source files to the corpus.

Next, we ran query tests. In each test, we choose a random token in the input file to mutate as described above. Then, we run UnnaturalCode on the input file and record the rank of the first correct result, r_q.

3.6.1 Mean Reciprocal Rank

The rankings are analyzed statistically using the reciprocal rank. The mean is reported as the mean reciprocal rank (MRR) Voorhees et al. (1999):

$$\mu = \frac{1}{|\mathcal{Q}|}\left[\sum_{q\in\mathcal{Q}}\frac{1}{r_q}\right].$$

$\mathcal{Q}$ is the set of all queries, and q is an individual query from that set. For example, $|\mathcal{Q}| = 120$ for an individual file and type of mutation. Using the MRR has several advantages: it differentiates the most among the first few (highest) ranks. MRR scores range from 0.0 to 1.0 where 0.0 is the worst possible score. In comparison, if we considered the average rank without taking the reciprocal, the worst possible rank would depend on the length of the input file. MRR represents the way that people tend to look through only the first few results.

In this case, we assume that a programmer will also only look through the first few error messages.

MRR is a very unforgiving measure of the performance of a system that returns multiple sorted results. In order to achieve an MRR greater than 0.75, the correct result must be the first result presented to the user most of the time. For example, consider three hypothetical token deletions performed as described above on a single file. If the correct result was ranked first for the first test, second for the second test, and third for the third test, UnnaturalCode would only have achieved an MRR score of 0.61 for that file.

MRR scoring was implemented for two different sets of interleaved JavaC and UnnaturalCode results. These combined results consist of a JavaC result followed by an UnnaturalCode result, followed by a JavaC result, and so on. The two variations are: 1) returning a JavaC result first; and 2) returning an UnnaturalCode result first. These combined results represent the intended use of UnnaturalCode as a way to augment compiler errors.

3.7 Validation Results

Figure 3.6 shows the distributions of the MRR scores of the files of versions of Lucene and Ant versus a Lucene trained corpus. The wider the shaded area is in these charts, the more files had that MRR. These plots also show the 25th, 50th and 75th percentiles as the beginning of the black box, the white dot, and the end of the black box in the center. Table 3.2 presents the cumulative mean MRRs for each data set and method.

UnnaturalCode performs very well at detecting mutations in code that it is familiar with. UnnaturalCode did very well with only the first Lucene 4.0.0 version in the corpus when tested against both Lucene 4.0.0 and Lucene 4.1.0. A test of Ant 1.8.4 against a foreign corpus (Lucene 4.0.0) results in poor performance. Syntax error detection performance is best with a corpus trained on the same or earlier version of the system.

Table 3.2: Cumulative Mean Reciprocal Ranks (Mean MRR)

Sources Tested	Corpus	Delete	Insert	Replace
Lucene 4.0.0	Lucene 4.0.0	0.88	0.99	0.98
Lucene 4.1.0	Lucene 4.0.0	0.77	0.91	0.91
Ant 1.8.4	Lucene 4.0.0	0.20	0.36	0.36
Lucene 4.1.0 Only new files	Lucene 4.0.0	0.30	0.47	0.48
Lucene 4.1.0 Only changed files	Lucene 4.0.0	0.68	0.86	0.85
Ant 1.7.0	Ant 1.7.0	0.86	0.99	0.98
Ant 1.8.4	Ant 1.7.0	0.55	0.75	0.74
Ant 1.8.4 Only new files	Ant 1.7.0	0.29	0.54	0.53
Ant 1.8.4 Only changed files	Ant 1.7.0	0.42	0.66	0.66
XercesJ 2.9.1	XercesJ 2.9.1	0.86	0.98	0.98
XercesJ 2.10.0	XercesJ 2.9.1	0.50	0.80	0.79
XercesJ 2.11.0	XercesJ 2.9.1	0.49	0.79	0.78
XercesJ 2.11.0 Only new files	XercesJ 2.9.1	0.25	0.46	0.47
XercesJ 2.11.0 Only changed files	XercesJ 2.9.1	0.50	0.81	0.79

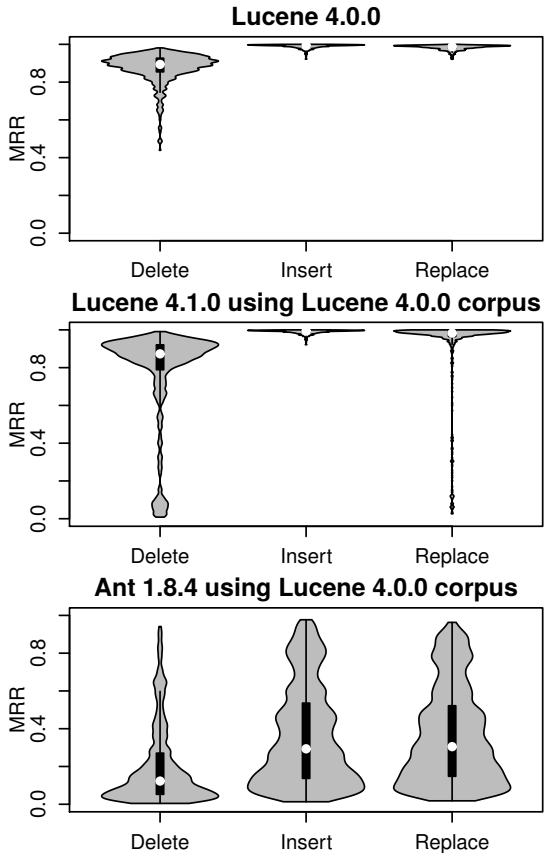

Figure 3.6: UnnaturalCode-only MRR Distributions of the files of Lucene 4.0.0, 4.1.0 and the files of Ant 1.8.4 tested against a Lucene 4.0.0 corpus.

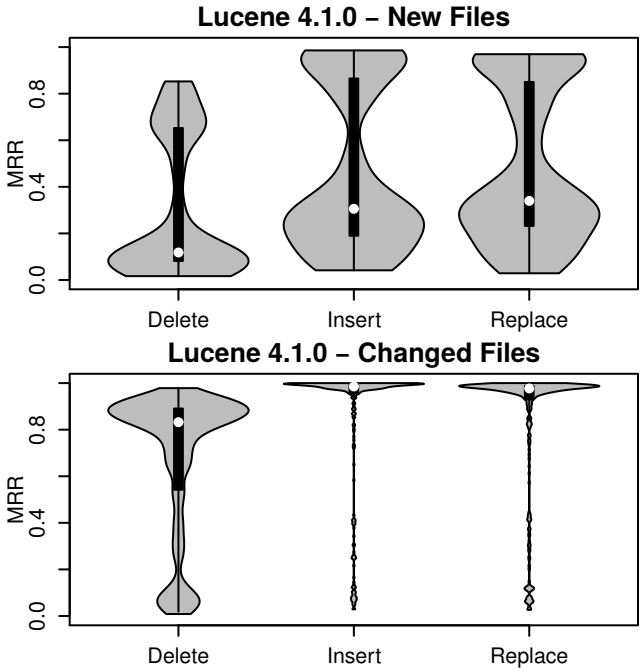

Figure 3.7: UnnaturalCode-only MRR Distributions of only new and changed files from Lucene 4.1.0, using Lucene 4.0.0 as the corpus.

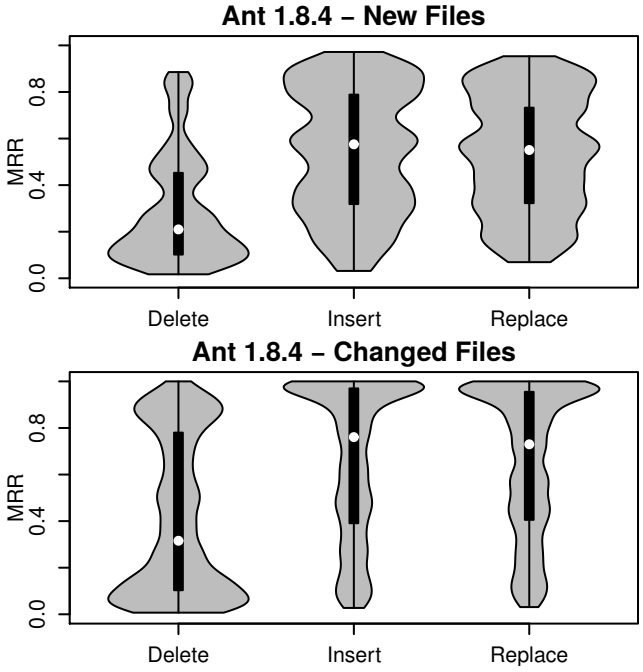

Figure 3.8: UnnaturalCode-only MRR Distributions of only new and changed files from Apache Ant 1.8.4, using ant 1.7.0 as the corpus.

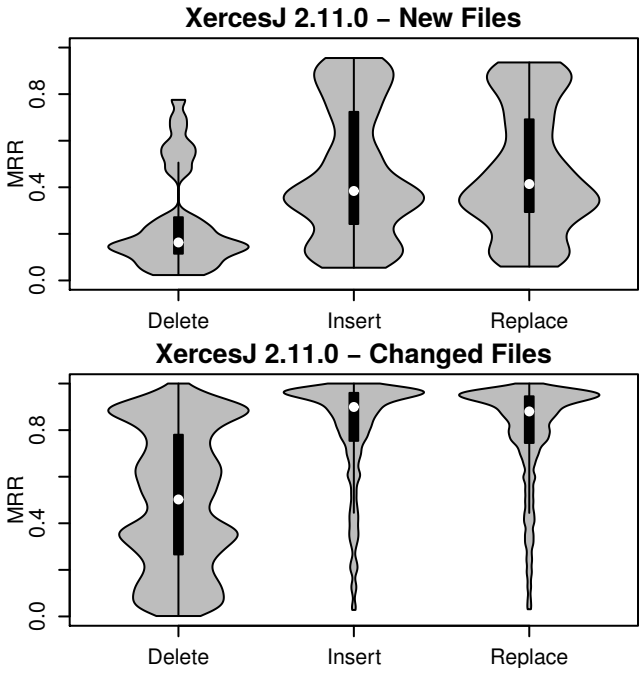

Figure 3.9: UnnaturalCode-only MRR Distributions of only new and changed files from XercesJ 2.11.0 Using XercesJ 2.9.1 as the corpus.

The scores and chart for the whole of Lucene 4.1.0 is not the entire story for that version. It contains three kinds of files: files unchanged from 4.0.0, changed files, and new files added since 4.0.0. Figure 3.7 clearly shows how these types files bring the MRR scores for 4.1.0 down from the scores for 4.0.0. The newly added files have very inconsistent performance with MRR scores near those of the scores for Ant's unrelated files, despite the fact that they are a part of the same project as the training corpus. Figures 3.8 and 3.9 show the same pattern on Apache Ant and XercesJ.

Figure 3.10 compares the performance of Ant 1.7.0 versus itself and Ant 1.8.4. The MRR behaviour is similar to the Lucene plots in Figure 3.6 for Lucene versus Lucene tests, and the poor performance of Ant versus Lucene has been negated by using Ant code in the corpus. The median remains very high, implying that UnnaturalCode scores very well on most files.

Figure 3.11 tests 3 consecutive major releases of XercesJ against a corpus of XercesJ 2.9.1. In all cases the median MRR of these tests are above or near 0.5: over 50% of the files have an MRR greater to or near 0.5. This is the

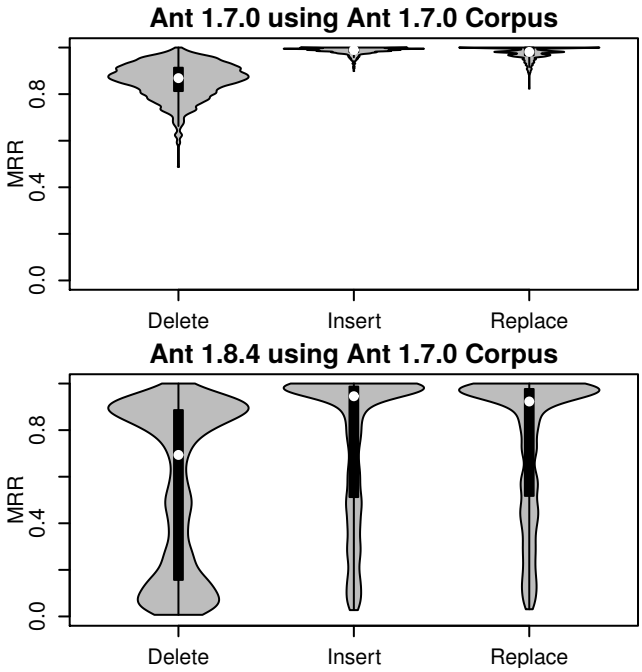

Figure 3.10: UnnaturalCode-only MRR Distributions of the files of Ant 1.7.0 and Ant 1.8.4 tested against the Ant 1.7.0 corpus.

same MRR score that a hypothetical system that always returns the correct result in second place would get. As with Ant, for XercesJ in the test on a past corpus (XercesJ 2.10.0 and 2.11.0) the bottom quartile extends further, but ends before an MRR score of 0.2, implying that for 75% of the files tested, UnnaturalCode performed similarly to, or better than, a hypothetical system that always returns the correct result in fifth place.

Table 3.3 shows the results of the interleaved tests and Figure 3.12 shows the MRR distribution for one of these tests. In this table, the column heading describes the output interleaving pattern: "UUUU" gives MRR means for UnnaturalCode results, "JJJJ" gives MRR means for JavaC, "JUJU" gives MRR means for interleaved results with JavaC's first result first in the output, and "UJUJ" gives MRR means for interleaved results starting with UnnaturalCode's first result. All four plots come from the same set of randomly chosen mutations. Both JavaC and UnnaturalCode perform well on their own. UnnaturalCode performs worse than JavaC on the deletion test and better than JavaC on the insertion and replacement tests. However, interleaved results

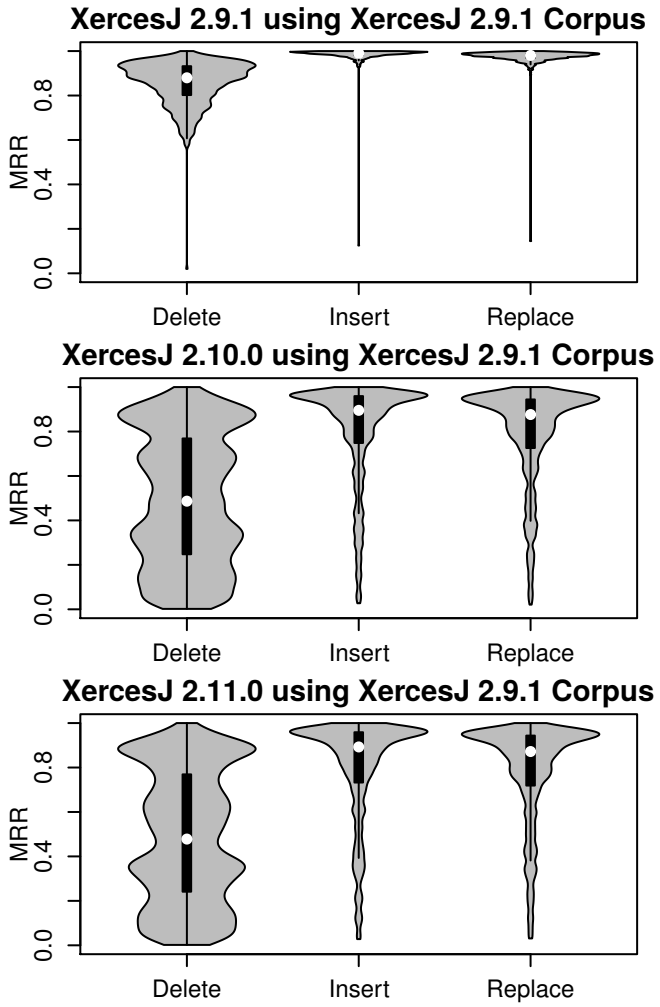

Figure 3.11: MRR Distributions of the files of XercesJ 2.9.1, 2.10.0, and 2.11.0 tested against the XercesJ 2.9.1 corpus.

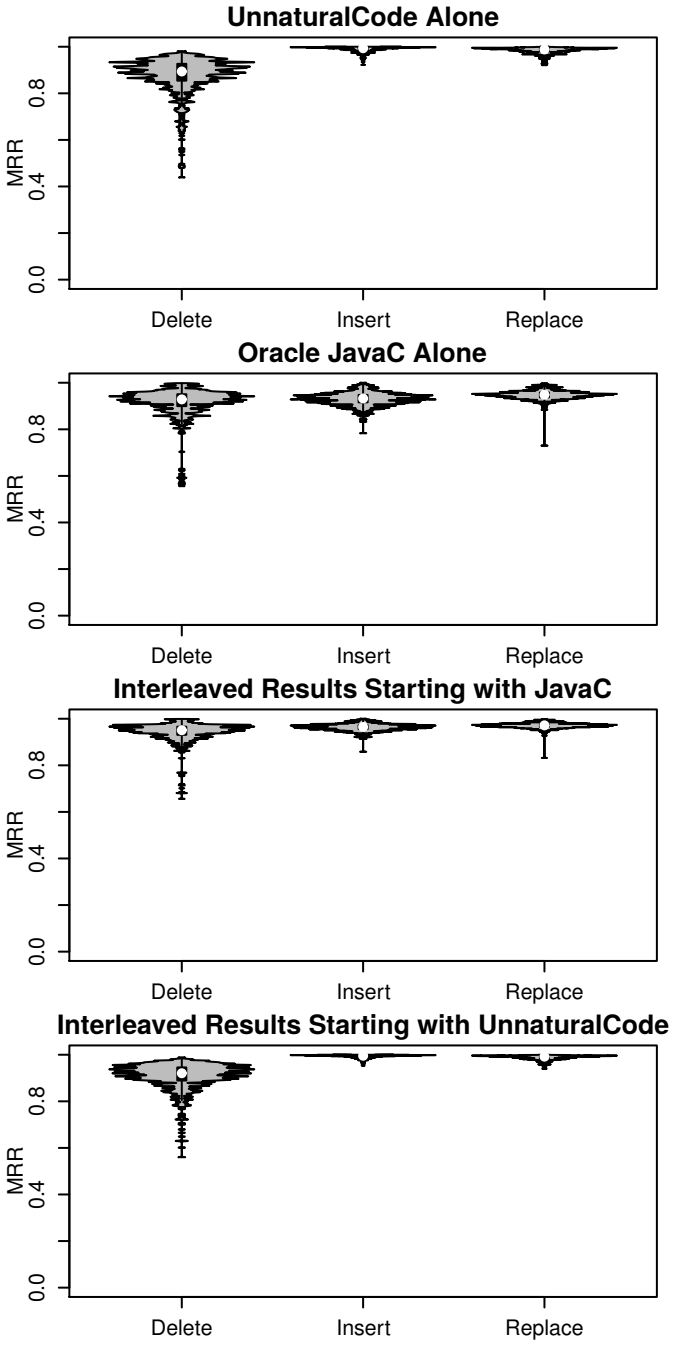

Figure 3.12: MRR Distributions of the files of Lucene 4.0.0 for the interleaved test.

Table 3.3: Interleaved Error Mean Reciprocal Ranks (MRRs).

		Interleaving Pattern			
Sources Tested	Corpus	UUUU	JJJJ	JUJU	UJUJ
Lucene 4.0.0	Lucene 4.0.0	.950	.932	.959	.963
Lucene 4.1.0	Lucene 4.0.0	.865	.937	.960	.914
Ant 1.7.0	Ant 1.7.0	.940	.927	.956	.958
Ant 1.8.4	Ant 1.7.0	.681	.923	.945	.806
Ant 1.8.4	Lucene 4.0.0	.308	.921	.930	.600
XercesJ 2.9.1	XercesJ 2.9.1	.939	.895	.937	.955
XercesJ 2.10.0	XercesJ 2.9.1	.694	.889	.916	.796
XercesJ 2.11.0	XercesJ 2.9.1	.688	.884	.911	.791

perform better than either system by itself: the best performing interleave depends on the file.

In the XercesJ tests, both UnnaturalCode and the Java compiler performed worse overall than the other projects. However, the interleaved results still dramatically improve performance. Interleaving allows the XercesJ results to approach the performance achieved with Ant and Lucene.

Table 3.4 shows how much closer to a perfect score of 1.0 each interleaved result ("JUJU" or "UJUJ") is compared to using the Java compiler's error messages exclusively ("JJJJ"). For example, if the Java compiler's MRR score was 0.9 and the interleaved score was 0.95, this would represent a score 50% closer to perfect. In cases where the the sources tested are different than the corpus, the UnnaturalCode-first interleaving, "UJUJ", makes the MRR score worse. However, the JavaC-first interleaving, "JUJU", always gets closer to a perfect score by 11-40%.

3.8 Discussion

The n-gram language-model approach was capable of detecting all mutations: inserted tokens, missing tokens, and replaced tokens. This is because the sequence of tokens will not have been seen before by the language model, assuming it has been trained on compilable code.

Table 3.4: Reduction of the gap between the Java compiler's MRR score and a perfect MRR score of 1.0.

		% closer to perfect MRR score	
Sources Tested	Corpus	JJJJ to JUJU	JJJJ to UJUJ
Lucene 4.0.0	Lucene 4.0.0	40%	46%
Lucene 4.1.0	Lucene 4.0.0	37%	-37%
Ant 1.7.0	Ant 1.7.0	40%	42%
Ant 1.8.4	Ant 1.7.0	28%	-151%
Ant 1.8.4	Lucene 4.0.0	11%	-406%
XercesJ 2.9.1	XercesJ 2.9.1	40%	57%
XercesJ 2.10.0	XercesJ 2.9.1	24%	-83%
XercesJ 2.11.0	XercesJ 2.9.1	23%	-80%

3.8.1 Performance on Milestones

Results were very inconsistent from file to file, and even, in the case of some files, such as `TermsFilter.java`, from version to version. One very clear pattern stands out, however. The deletion mutations were much easier for UnnaturalCode to detect than the insertion or replacement changes in the feasibility tests above. However, in the compiler integration test they were harder to detect.

In the milestone tests, some files would get consistently wrong results because the top results would always be the same regardless of where the changes were made. In particular, files from outside the corpus that contained strings of new identifier tokens would consistently produce poor results with UnnaturalCode.

Unfortunately, with only a single project in the corpus, performance was sometimes very poor. This poor performance could be easily triggered by adding new identifiers that were not present in the corpus, since those new identifiers were labelled with high entropy by the model. Sometimes this behaviour is accurate, as in the case of a misspelled identifier, but sometimes it is inaccurate, as in the case of a newly added, but correctly spelled, identifier.

These results are highly encouraging, however. Even the worst MRR score of 0.20 implies that there are files in Apache Ant in which errors are locatable

using a corpus trained on Lucene. If one runs the tests as described above on broken Java code that imports classes from packages that do not exist in the training corpus, these correctly specified imports will almost always be the top five results. However, this behaviour only persists until they are added to the corpus, and this happens after the first successful compilation of those files. For example, even though class imports in Ant 1.8.4 share the same `org.apache` class path, they also contain new identifiers. When `ProjectHelperImpl.java` imports `org.apache.tools.ant.Target`, this string contains three identifiers, "tools," "ant," and "Target," which are not in the training corpus. This will cause the average entropy of any string of 20 tokens containing "`tools.ant.Target`" to contain at least three high entropy contributions. In comparison, when the code is mutated for testing, at most one additional unseen token is introduced. One possible solution to the new identifier problem is to ignore identifier content and train on the type of the identifier lexeme itself (Allamanis and Sutton, 2013).

The syntax error detector can often misreport new identifiers, such as package names, as syntax errors.

3.8.2 Performance on Revisions

Some revisions have much higher MRR than others. An examination of some of these changes reveals what may have caused their score to be particularly high or low.

Lucene revision 1408367 had a particularly low mean MRR score of 0.15. In this revision there was no Java code change hunk larger than the 25-token minimum in UnnaturalCode, and the remaining large hunks were English language comments. These comments were not removed because `diff` had removed the leading `/*`. Another low-scoring revision, number 1419892, with a mean MRR of 0.12, contained a hunk with a fragment of the Apache 2.0 License.

> Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership. The ASF licenses this file to You under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

Thus, the actual performance on Java code from the diff hunks is likely higher than the mean presented in Table 3.2 because that result includes hunks of English text. The validation framework failed to remove these English texts from the hunks it considered. Sometimes diffs lack the context to resolve their content. Comments are often modified such that the code signalling the beginning of the comment is not recorded in the diff.

There are also high-MRR outliers in the set of revisions considered, such as revision number 1425817, in which the Lucene developers added new variables, "`CONN_LOSS`," and "`EXP`." However, these new variables were added with common Java attributes and types: "`private static final boolean`." This common sequence occurs 20 times in the training corpus. Therefore, they help keep the entropy of token sequences containing the new declarations low by averaging out the high entropy of "`CONN_LOSS`," and "`EXP`," that do not occur in the training corpus anywhere.

3.8.3 Performance with Interleaved Errors

The UnnaturalCode/JavaC combination performed better than either JavaC or UnnaturalCode alone in every test case.

UnnaturalCode was often capable of detecting errors that JavaC was not and JavaC often was capable of detecting errors that UnnaturalCode was not. This fact allows UnnaturalCode to augment the accuracy, in terms of MRR score, of the compiler when its results were interleaved. The interleaving improved the MRR score from 0.915, using just JavaC, to 0.943 using interleaved

results for the deletion test on Lucene 4.0.0. This closes the gap between the Java compiler's MRR score and a perfect score of 1.0 by 34%. The best interleave choice is not consistent, so it may be best to present results to the user side-by-side.

3.9 Threats to Validity

Construct validity is affected by the assumption that single-token mutation is representative of syntax errors. This assumption may not be very representative of changes a software engineer would make between compilation attempts.

However, single-token mutations are the worst-case scenario and provide a lower bound on performance.

A file with two syntax errors is further from compiling than a file with one syntax error. Since UnnaturalCode only models files that do compile, a file with two syntax errors will have more entropy than a file with one syntax error. In this case a developer would be presented with the highest-entropy error first. If they chose to fix that problem and recompile, they would be presented with the next-highest entropy error. Multi-token errors would be easier for UnnaturalCode to detect.

Internal validity is hampered by using the MRR formula to score our ranking of the correct query results. These rankings are affected by the maximum gram length, n, where typically larger is better, and the number of total results. The number of total results is $l - 2n/s$ where l is the length of the file and s is the step size. Since correct results comprise at most $2n/s$ of the results, the chance of the correct result appearing in the top 5 if the results are sorted randomly is approximately proportional to $1/l$. In other words, the system will naturally perform better on short input files simply because there is less noise to consider. This is the reason diff hunks with less than 25 tokens are discarded. Since $\frac{25-2\times5}{1} = 5$, there are only 5 total query results and the correct one is always in the top 5. The revision tests can accidentally run on

context-less Java comments, when the comment tokens (`/*` , `*/` or `//`) are missing, which can have some effect on the MRR results.

External validity is affected by the choice of Java projects. The experimental evaluation covers 5 million tests of UnnaturalCode and JavaC across 3 different medium-to-large Apache Foundation Java projects: Ant, XercesJ and Lucene. JavaC compilation is much slower than querying UnnaturalCode alone.

Java has a syntax typical of many C-like languages. Thus, these tests are a good representative, and the results for these projects will generalize well to other projects. These results should generalize to other languages as well but this assumption has not been tested yet. The evaluation on 5 million single-token mutation tests across 3 distinct corpuses is a fairly significant evaluation of the new method.

3.10 Future Directions

This technology can be very useful to software engineers who are actively developing a piece of software. All the engineer must do is instruct their build system to call the wrapper instead of the compiler directly, and the system will begin building a corpus; if the compile fails it will return side-by-side ranked results from both the compiler and from UnnaturalCode. The performance evaluation indicates that a system using the ideas presented in this chapter will suggest the location of the fault in the top two results often enough to be useful as long as the corpus contains one successful compile of the same project.

The method proposed should be implemented not only as a compiler wrapper for general purpose command-line use, but also as a plug-in for an integrated development environment (IDE) such as Eclipse. In such an environment it could provide immediate and visual feedback after a failed compile of which lines were likely to cause problems, perhaps by colouring the background with a colour corresponding to the increase in entropy for those lines.

The generalization to other languages needs to be evaluated. One simply needs to replace the Java lexical analyzer used in UnnaturalCode with a lexical analyzer for the language they wish to use and modify the compiler wrapper to locate input file arguments and detect the failure of their compiler of choice. The n-gram model is flexible and will work with many programming languages (Hindle et al., 2012).

The effect of a multi-project corpus on syntax error detection should be investigated. The idea is to explore what makes a good corpus for general purpose syntax error detection.

Statistical information on what typical coding mistakes look like is missing from UnnaturalCode, as it can only characterize correct code. Therefore, software repositories could be mined for patches that fix coding mistakes and syntax errors. That data could then be used to characterize those mistakes and fixes statistically. Source code could then be examined for code resembling the broken code from those defective commits.

Several extensions to the system could be implemented. For example, the combination of this method with token prediction could automatically repair source code by statistically directing most-likely syntax repair searches based on a dynamic and project-specific corpus instead of statically defined least-cost repair searches such as those presented in Corchuelo et al. (2002) and Kim and Choe (2001). This approach should be much more efficient than the approach applied in Weimer et al. (2009). It may also be interesting for developers or project managers to be able to see the entropy of each line or token of source code themselves because the entropy may correlate with other properties of the source code that have not yet been considered.

3.11 Conclusion

This chapter presents a system to address the common issue of poor syntax-error location reporting by modern compilers. Syntax errors plague novice and experienced programmers alike. UnnaturalCode avoids parsing completely, in-

stead relying on a statistical language model from the field of natural language processing. The choice of model was guided by the promising results in code completion. UnnaturalCode consists of a compiler wrapper, a lexical analyzer, and a language model that cooperate and are more than fast enough to be used interactively. UnnaturalCode is tested by inserting, deleting and replacing random tokens in files from the software that it was trained on, and later versions of the software it was trained on, and a completely independent software package. The experimental evaluation shows that UnnaturalCode performed very well on the same version, well on the next version, and poorly on external software. When UnnaturalCode's results are combined with the compilers own results, the correct location is reported earlier than with the compiler's results alone.

This work helps bridge the gap between statistical language analysis tools and computer-language parsers. Bridging this gap solves a problem that has haunted the users of parsers since they were invented: accurately locating syntax errors. This approach is unique in that it employs a statistical model: all previous work in this area are based on parser modifications to repair or recover from syntax errors, static code analysis, predefined heuristics, or type analysis. Furthermore, the model can evolve with the changing contexts of individual engineers and projects as it continuously updates its training corpus. By combining this method with the current compiler's error messages we achieve better performance in syntax error localization than either technique alone.

The n-gram language-model is capable of enhancing the compiler's ability to locate missing tokens, extra tokens, and replaced tokens.

Finally, when you build a language model of working source code, "syntax errors just aren't natural."

Chapter 4

Dynamic Detection of Typographical Errors in Python Code

4.1 Preface

4.1.1 Acknowledgement

This chapter was originally published as a "pre-print" (Campbell et al., 2015). I, Hazel Victoria Campbell, wrote the majority of the chapter, conducted the experiments, collected data produced results and analysis, and created most of the figures. Dr. Abram Hindle wrote the related work section, provided editing, some figures, and modifications to the software used, and supervised the research. Dr. José Nelson Amaral provided editing and supervised the research.

4.1.2 Significance

This chapter's significance centers around the differences between strict, statically typed, and fully compiled languages like Java and less strict, dynamically typed, and interpreted languages like Python. Since Python is not statically compiled, the scope of this chapter was expanded to include run-time errors that can be introduced by typographical errors in the source code.

The work presented in this chapter was the first time that Natural Language Processing (NLP) techniques had been applied to the problem of locating

errors both at the time of compilation and during run-time. Due to the nature of errors at run-time the work presented in this chapter had to overcome many challenges that simply were not present in the previous chapter.

The software developed to conduct the research presented in this chapter forms the foundation of the software developed to conduct the research presented in the next chapter.

4.1.3 Impact

The original publication of this work has been cited by Derezinska and Hałas (2015) as a source for mutation operators to avoid when performing mutation testing. Since the Python interpreter often fails to detect the types of mutations introduced in this work (see Section 4.7), these types of mutations would be counter-productive during mutation testing.

4.2 Abstract

Dynamic scripting programming languages present a unique challenge to software engineering tools that depend on static analysis. Dynamic languages do not benefit from the full lexical and syntax analysis provided by compilers and static analysis tools. Prior work exploited a statically typed language (Java) and a simple n-gram language model to find syntax-error locations in programs. This work investigates whether n-gram-based error location on source code written in a dynamic language is effective without static analysis or compilation. UnnaturalCode.py is a syntax-error locator developed for the Python programming language. The UnnaturalCode.py approach is effective on Python code, but faces significantly more challenges than its Java counterpart did. UnnaturalCode.py generalizes the success of previous statically-typed approaches to a dynamically-typed language.

This chapter also presents a novel tool for random-edit mutation-based automatic exploration of program code variants. The random-edit mutation

tool is used to analyze both the implementation of the Python language and UnnaturalCode.py.

4.3 Introduction

This chapter seeks to help Python programmers find coding mistakes by creating an error location tool, UnnaturalCode.py, for use with Python.

Campbell et al. (2014) describes a prototype system, UnnaturalCode, for locating syntax errors in programs written in the Java programming language. UnnaturalCode produced additional diagnostic information for Java source files that failed to compile. UnnaturalCode used an n-gram language model to identify snippets of source code that it finds unlikely to be correct. UnnaturalCode was shown to be effective by locating some errors that the Java compiler was not able to locate. Syntax errors and syntax-error reporting are important because researchers have found that syntax errors are serious roadblocks for introductory programming students. Garner et al. (2005), corroborated by numerous others (Jackson et al., 2005; Jadud, 2005; Jadud, 2006; Tabanao et al., 2008) show that "students very persistently keep seeking assistance for problems with basic syntactic details."

In this chapter, a new UnnaturalCode.py system is applied to a new programming language. In the interests of diversity and generality, another popular programming language, which is very different from Java, was chosen: Python. Besides being a dynamically typed scripting language, Python also offers additional challenges for the localization of errors, and thus is a good candidate to evaluate UnnaturalCode in a different point in the space of programming languages. For instance, Python is an interactive scripting language, while Java is not. Python has different syntactical style from Java: it uses whitespace to define the end of statements and the extent of code blocks.

Python has taken hold as an instructional language (Bell et al., 2012; Krpan and Bilobrk, 2011). It is popular to teach Python and Python is now quite popular among scientists (Millman and Aivazis, 2011; Pérez et al., 2011).

Many new programmers are being introduced to programming with Python because Python lacks excessive syntax and forces structured and modular programming via forced whitespace block structures (Stajano, 2000).

However, the approach used in Campbell et al. (2014) does not work with Python because of the differences between Java and Python. A new approach was developed, to generalize the previous approach.

Differences between languages have implications for UnnaturalCode, the practice of software development, and the use of Python as a language for experimentation. Experiments and practices that involve generating Python code that may or may not execute are the focus of this chapter. These type of experiments are heavily impacted by Python's features, especially its lack of compilation and static analysis.

The contributions of this chapter are: 1) a corpus-based tool, UnnaturalCode.py, that can help users locate the causes of errors in Python software; 2) a novel methodology and tool for mutation testing of Python tools without restricting the forms those mutations can take; 3) an analysis of the differences in approach required for dynamic scripting languages as compared to approaches that relies on statically typed and compiled languages.

The trumpeting of Python as an excellent introductory language, raises the question "does Python's own language definition support new developers?" More explicitly, Python is considered a great instructional language despite lacking analytical tools to recognize basically correct programs. Studies such as Krpan and Bilobrk (2011) fail to show that Python is an effective introductory programming language.

Python cannot give timely and appropriate error messages to new programmers because error-ridden code might never be executed: syntax errors can exist in Python files that are currently runnable and executable. Java, on the other hand, requires a lot of boilerplate to get started, yet Java statically type checks programs before compilation. Programs written in Java must have appropriate syntax before they are compiled, unlike Python.

This chapter investigates automatic error discovery/reporting in Python programs via a corpus-based approach. This investigation revealed the challenges that liberal typing and scoping pose to such a corpus-based approach.

4.4 Background

UnnaturalCode was introduced in Campbell et al. (2014). It is a prototype system that effectively finds syntax errors by training itself on working Java source code and then trying to find phrases of source code that it has not seen before. This chapter differs from the prior work by introducing a new system with some of the same core ideas. UnnaturalCode.py includes a mutation-testing framework for Python implementations, software engineering tools, and test suites.

The UnnaturalCode prototype for Java depended on language features such as declarative scope, static typing, context-free syntax and pre-compiled libraries. Additionally, source-code compliance with the rules and requirements of these language features is testable with a single tool: the Java compiler. It is also reasonable to expect the Java compiler to halt (not halting would be a bug). In this chapter, however, arbitrary Python programs are used instead of a compiler, and cannot be reasonably assumed to behave in any particular way. This work discusses many of the issues faced with error location in a dynamic language, as well as mutation testing in a dynamic scripting language.

4.4.1 n-Grams in Software Engineering

Hindle et al. (2012) proposed the application of n-gram language models to source code. These models were classically applied to natural language text, but Hindle et al. (2012) showed that software had properties (low entropy) that made it compatible with these kinds of models.

N-grams are a phrase formed by at most n words or tokens. An n-gram language model is simply a collection of counts that represent the number of times a phrase appears in a corpus. Empirically the probability of an n-

gram is simply the frequency of occurrence of the phrase in the original corpus. For example, if the for loop "`for x in l :`" occurred 10 times in a corpus consisting of 1000 5-grams, its probability would be 0.01. One could consider the probability of a token in its surrounding context. For example, the token "`in`" might have a probability of 1 given the context of "`for x ------ l :`" because "`in`" may be the only token that had been observed in that context.

Performance of n-gram models tends to increase as n increases, but so does the memory required to store the model. Larger values of n cause sparsity — not all n-grams of size n will be observed if n is large. Empirically, unless the corpus is large enough, there will be unobserved and legitimate n-grams that are not in the corpus. If missing n-grams have a probability of zero, the model cannot estimate any probabilities on the unseen text. This short-coming is addressed by smoothing. Smoothing estimates the probability of unseen n-grams from the probabilities of the largest m-grams (where $m < n$) that are part of the unseen n-gram and that exist in the corpus. For example, if the corpus does not contain the phrase "`for x in l`" but it does contain the phrases "`for x in`" and "`l`" it would estimate the probability of "`for x in l`" using a function of the two probabilities it does know. Specifically, UnnaturalCode.py uses Modified Kneser-Ney smoothing (Hsu and Glass, 2008).

UnnaturalCode.py works by finding n-grams that are improbable – they have high entropy. Thus UnnaturalCode.py exploits the fact that syntactically invalid source code often has a higher entropy. The entropy of source code is computed with respect to a corpus of code that is known to be compilable. Unnatural Code looks for source code that is *unnatural* to a n-gram model trained on compilable source code.

4.4.2 Programming Errors

There is much empirical research on the effect of syntax errors on novices and experienced programmers (Garner et al., 2005; Jackson et al., 2005; Jadud, 2005; Jadud, 2006; McIver, 2000; Tabanao et al., 2008). The consensus is that

programming errors consume time, occur frequently for new programmers, and persistently for experienced programmers.

Kummerfeld and Kay (2003) studied the effect of syntax errors on programming experience and program comprehension. They found that inexperienced students made rapid and erratic modifications to their source code, in the hurried hope of achieving a compiling program. These furious modifications do not exhibit much strategy or reasoning, but represent a brute-force approach to programming. When studying experienced programmers, they found that after the failure of a set of syntax-error solving strategies, the experienced programmers would revert to this brute-force random edit strategy as well.

Student performance (grades) have been found to negatively correlate with the frequency of syntax errors. Tabanao et al. (2011) and Tabanao et al. (2008) studied more than 120 students and found that errors and compilation frequency were negatively correlated with midterm test performance.

Jadud (2005) and Jadud (2006) studied first-year students' programming errors by taking snapshots of code before compilation. Each failure was manually labelled. Of relevant failures, 5% took students longer than 10 minutes to resolve while 60% only took 20 seconds. Thus, addressing and fixing syntax errors is time consuming for inexperienced programmers.

4.4.3 Technical approaches to Syntax Errors

Two main methods used to identify and report coding errors are *parser-based* and *type-based*. *Parser-based* methods augment the parser to enable better reporting or to skipping over ambiguous sections in hopes of providing more feedback. Many of these techniques seek to create more error messages that might be more helpful. Burke's parse action deferral (Burke and Fisher, 1987) backs the parser down the parse stack when an error is encountered and then discards problematic tokens. This approach allows the rest of the file to be checked as well, assuming that the state of the parser is recoverable. In a similar vein, heuristics and cost-based approaches, such as production rule prioritization resumption were combined by Graham et al. (1979) in order to

provide better parse error messages. Parr and Fisher (2011) discusses the LL(*) parsing strategy used in parser-generators such as ANTLR. LL(*) parsers dynamically attempt to increase the look-ahead at parse time up to a constant maximum look-head, finally failing over to a backtracking algorithm. This method gains contextual error reporting from a top-down perspective enabling source-level debugging while still performing well enough to be a usable and popular parser.

Other parse-based methods avoid reporting spurious parse errors. They often work by attempting to repair the source code or by resuming the parse after an error. Corchuelo et al. (2002) and Kim and Choe (2001) apply search methods to find repairs to source code to enable a parser to continue, often without user intervention. These are merely two examples of such techniques: research on parser error recovery and diagnostics spans many decades and is far too voluminous to list here.

Type-based static analysis is leveraged as an alternative method of parse and syntax error fixing. Python employs a liberal type system and parser that can be viewed as a barrier to *Type-based* static analysis. Cannon concluded that one barrier to type inference and type-based static analysis was that "Object-oriented programming also affects the effectiveness of type inference." (Cannon, 2005). Many attempts at type inference, such as Salib (2004), Psyco by Rigo and Tismer (2001), Rigo (2004) and Python ignore part of Python's semantics or just handle subsets of Python's language. PyPy (Rigo and Pedroni, 2006) is the most modern effort at static analysis of Python code by enabling just in time (JIT) compilation of Python code. RPython by Ancona et al. (2007) attempted to limit the semantics of Python to work "naturally" within statically typed domains such the JVM or CLI. Thus there has been some work on Python, static analysis and *type-based* analysis, but much of it either chooses to work on safe subsets of the languages or restrict language semantics.

Type-based analysis focuses on reconciling the types of the identifier and functions called, rather than focusing on grammatical correctness. Heeren

(2005) leverages types to implement a constraint-based framework within the compiler.

Some *type-based* techniques leverage corpora either to aid their type-based reasoning with heuristics or to improve search performance. Hristova et al. (2003) leverage predefined heuristic rules to address common mistakes observed in code. Lerner et al. (2007) use a corpus of compilable software and its types to improve type error messages for statically typed languages.

One *mutation-based* technique was described by Weimer et al. (2009). They use Genetic Algorithms to mutate parse trees to fix defects. In comparison to previous mutation-based approaches, the mutations presented in this chapter are not guaranteed to produce either parsable text or a valid parse tree.

4.5 Implementation

UnnaturalCode.py is intended to augment the Python run-time environment's error messages with additional information about the probable location of a coding error. It is comprised of two major parts. The first is a piece of Python software, and the second is a modified version of the MIT Language Model (MITLM). MITLM is a software implementation of the n-gram language model written in C++.

UnnaturalCode.py wraps around the system's Python interpreter in order to add the desired functionality: UnnaturalCode.py is invoked to run Python programs instead of the system's Python interpreter. First, UnnaturalCode.py runs the desired Python program in exactly the same way as the system's Python interpreter. Then, UnnaturalCode.py checks the result of the execution.

There are two possible outcomes:

- In the case that the Python program exited successfully, or more preferably, the Python program's test suite *passed*, UnnaturalCode.py can add it to its corpus of known-good Python code.

- In the case that the Python program exits with an *error*, UnnaturalCode.py attempts to locate the source of that error and to give the user a suggestion of where to look for coding mistakes along with the standard Python error information.

UnnaturalCode for Python would fit within Beck's test-driven-development (Beck, 2003) loop whereby after a success run of tests the updated code would be used to update the n-gram model. Such updates would ensure that the code used to build the model is of sufficient quality.

UnnaturalCode.py could be integrated into Beck's test-driven-development (Beck, 2003) process in two places: as a method to help locate test-failing code in the case of an error inducing a test failure; after tests pass UnnaturalCode.py could be updated with the new working code so that it becomes more familiar with the successful test-compliant system.

UnnaturalCode.py does not interfere with the usual process of executing Python software. UnnaturalCode.py's goal is only to augment the usual Python diagnostics and error output with its own suggestions. In order to achieve this goal it performs the following functions: first, it lexically analyzes the Python program with its own custom lexical analyzer. This lexical analyzer is based on Python's standard lexical analyzer, but modified to continue in the case of an error instead of stopping.

Second, it breaks the Python file that caused the error into sequences of 20 contiguous tokens using a sliding window. A 20-token sliding window is used to ensure that MITLM has sufficient context to employ smoothing. Then, it sends each sliding window to MITLM. MITLM can either be running locally or on a server. MITLM returns, for each sequence, a single value representing the cross entropy (or, log probability) of that sequence versus the corpus of known-good Python code.

Finally, UnnaturalCode.py reports to the user the sequence that has the highest cross-entropy, along with its line number, file name, and the usual Python diagnostics and error messages. The entire UnnaturalCode.py process

usually takes less than 1 second and 300-500MB of RAM on a modern 64-bit PC for a single large Python project.

MITLM is configured to use a 10-gram model, and the modified Kneser-Ney smoothing, when estimating entropy values. This configuration allows UnnaturalCode.py to receive reasonable entropy estimates even for token sequences that have never been seen before.

UnnaturalCode.py also includes a program mutation tool, which it uses to test itself. This tool can be used to test test suites, error handling, and Python implementations. The mutation tool includes rules for 14 different types of mutations, all of which are designed to be extremely general.

11 of those 14 types of random mutations were studied in this chapter: token deletion, token insertion, token replacement, digit deletion, digit insertion, letter deletion, letter insertion, symbol deletion, symbol insertion, line dedenting, line indenting.

These 11 types of random-edit-mutations are intended to simulate mistakes that a typical developer may make when writing Python code, such as misspelling identifiers, typos, unbalanced parentheses, braces and brackets, bad indentation, missing characters, and using incorrect operators.

In all three token mutations, a token is chosen at random. Deletion simply removes the chosen token. Insertion inserts a copy of the chosen token at a random location. Replacement writes a copy of the chosen token over another randomly selected token. In digit, letter and symbol mutations, a single digit, letter, or symbol character is deleted or inserted randomly in the file. In the indentation mutations a line in the file is selected at random and its indentation is decreased or increased. Digit, letter, symbol, and indentation mutations may mutate any part of the file, including code and comments. Token mutations can only affect the code and never comments.

Other techniques for mutating Python code are far more limited, such as the operators presented by Derezińska and Hałas (2014) or the mutations employed by Moore (2000–2013). These mutators are designed to produce reasonably valid and executable Python programs, whereas the mutations used

in the experiments here are not. UnnaturalCode.py mutations are designed to be as general as possible, they do not guarantee an executable program after mutation. Thus the set of possible of programs we generate is larger than the set that Moore (2000–2013) produces. In this chapter these mutations are only applied once so that they produce general text files that are similar to known good Python programs. Two types of mutation rules are available: rules that are guaranteed to produce text that Python can successfully lexically analyze, and rules that do not have this guarantee.

In order to obtain as much information on whether a Python program is valid or not, while also preventing that program from affecting the operation of UnnaturalCode.py, UnnaturalCode.py runs code in a separate process. Execution is limited to 10 seconds, though this limit was never needed. The mutation testing system has several features that manage the execution of random, unpredictable, Python code and extract error-report data produced by Python and Python programs. Python types may change at runtime, Python does not enforce encapsulation, and Python does not have a standard external debugger. Thus, UnnaturalCode.py executes Python code in a separate process to prevent unknown programs and mutants from changing types, global and local variables used by UnnaturalCode.py. UnnaturalCode.py also obtains debugging information from the process under test, and exports that information back to the main UnnaturalCode.py process for examination. UnnaturalCode.py ensures that all testing processes halt by forcefully killing processes in case they exceed a preset amount of execution time.

UnnaturalCode.py depends only on its own code, the implementation of the Python language, and MITLM. MITLM is the only component of UnnaturalCode.py that was preserved from the prototype Java implementation of the tool. UnnaturalCode.py uses a slightly modified version of Python's own lexical analyzer. It does not use a lexical analyzer generated by a parser-generator from a grammar.

Additionally, UnnaturalCode.py is intended not only for research but also for practical use by Python developers. One can download and experiment

with UnnaturalCode.py as it is distributed freely on GitHub: `https://github.com/hazelybell/unnaturalcode`.

4.6 Experimental Validation Procedure

UnnaturalCode.py is designed to help the Python developer locate simple programming mistakes such as typos. Most of these mistakes are syntactic in nature, although some may be semantic errors, such as misspelled identifiers. Consider the following example Python program:

```
def functionThatExpectsTwoArguments(a, b):
  return True
def testA():
  functionThatExpectsTwoArguments("a" "b")
def testB():
  functionThatExpectsTwoArguments("a",-"b")
def testC():
  functionThatExpectsTwoArguments["a","b"]
```

The program listed above executes without error in the Python interpreter and loads without error if imported as a module, indicating that it has basic syntactic validity. However, importing this Python module and running any of the three test functions would result in a `TypeError`. A Python programmer could quickly identify the simple mistakes: there is a comma missing in `testA`; in `testB` there is a stray "`-`" and in `testC` square brackets were used instead of parentheses.

All three of these mistakes would be quickly caught by a compiler at compile time. However, Python must load this file and actually run one of the three broken lines of code in order to discover this mistake. The experimental validation that follows attempts to evaluate UnnaturalCode.py's ability to locate simple coding mistakes such as these, including both mistakes that Python can catch and mistakes that it cannot.

First, known-good Python source files were collected from a variety of Python projects including Django, pip, setuptools and Zope. These are popular, well-known projects that are used in real-world production Python environments. The files collected are assumed to be in good condition, and free of

syntax errors. Some files were modified to substitute relative import paths for absolute import paths in order to run. Not every file from these projects was used because some of them required configuration or configuration files, or relied on external libraries not available to the authors, such as Oracle database libraries and could not run without such dependencies. Files with less than 21 tokens were also excluded because they are too short to produce meaningful results. 936 files remained after removing files that were inappropriate for the mutation experiments. UnnaturalCode.py will always be able to locate the error in a file shorter than its sliding-window length. Including such files is akin to asking which line a syntax error is on in a file with only one line. The Python files used are available at `https://github.com/hazelybell/pythonCorpus`.

Collecting these 936 files required each to be tested, library dependencies installed, run-time dependencies configured, module paths specified or corrected. Programs with errors may not terminate; therefore a limit of 10 seconds was imposed on the execution of any single file. Among the collected test cases no known-good file would run for more than 10 seconds. These known-good files were then used to build a corpus, which was then used to build a 10-gram language model with MITLM. UnnaturalCode.py updates its own corpus as soon as a valid version of a new or changed file is discovered. Therefore, UnnaturalCode.py has all known-good versions in the corpus at the beginning of the validation procedure, including files from the current project. UnnaturalCode.py is designed to have the most-recent working versions of all files in its corpus during regular usage (that is the entire set of 936 files). Thus, starting UnnaturalCode.py with a corpus of all known good files is done to test UnnaturalCode.py's intended use case of running after modifying a Python source file. Each file was then repeatedly subjected to a random-edit mutation and tested against both Python and UnnaturalCode.py.

Once a file was mutated, the location of the mutation was noted, and the mutant file was '`required`', imported and executed, using *Python 2.7* because the original files were all designed to be run with this Python version. This is similar to running the file from the command line: `python2.7 filename.py`.

This tests if the file parses and executes. One of two possible outcomes was recorded: 1) the file ran successfully; 2) the file exited with an error; or 3) the file took more than 10 seconds to execute at which point it was terminated. The exact type of error and location were recorded and compared to the location of the mutation, see Table 4.1. For the sake of brevity, only common errors reported by Python are shown in the results in Table 4.5.

The language model is queried with every contiguous 20-token window present in the file, and reporting the single contiguous 20-token window with the highest entropy with respect to that model. The location of the 20-token window was then compared to the location of the mutation. A source file that is 200 tokens long will have $200 - 20 + 1 = 181$ 20-token phrases.

This validation procedure differs slightly from the one described in Campbell et al. (2014). The differences are: 1) some files were excluded because they had missing dependencies or run-time requirements; 2) some files were edited to specify full import paths or to remove run-time checks such as checks to ensure the database is configured properly; and 3) only the top result from UnnaturalCode was considered to avoid biasing against Python, which only returns one result.

4.7 Results

The data in this section is presented as the fraction of experiments for which the first result returned by Python or UnnaturalCode.py is near the location of the mutation, which is denoted *precision*. Precision is a measure of the performance of an information retrieval system. In this chapter, precision measures how often Python and UnnaturalCode.py *locate* a mutation. False positive rate is irrelevant to UnnaturalCode.py because it only runs in the presence of an error indicated by Python. Only a single result is considered. Therefore, precision is equal to *recall*, 1-precision, precision at 1 result, and mean reciprocal rank (MRR) with only 1 result. This metric was chosen because Python only produces at most a single result. Therefore, the comparison would be

unfair if UnnaturalCode.py was allowed to produce more than one location for examination by the user, although it is capable of doing so.

Table 4.1: Experimental Data Summary

Python Source Files	936
Source Files from django	623
Source Files from pip	159
Source Files from zope	75
Source Files from setuptools	52
Source Files from Python	25
Source Files from markerlib	2
DeleteToken mutations	560400
InsertToken mutations	560400
ReplaceToken mutations	532372
DeleteDigit mutations	79225
InsertDigit mutations	93500
DeleteSymbol mutations	93550
InsertSymbol mutations	93400
DeleteLetter mutations	93550
InsertLetter mutations	93523
Dedent mutations	93450
Indent mutations	93550
Total data points	2386920

Table 4.1 shows some summary statistics about the experimental data gathered. Each file was subjected to many different mutations of each type in order to obtain a mean precision value.

Table 4.2 shows the overall performance of Python and UnnaturalCode.py on the 11 types of mutations tested. Each number in the table represents the fraction of injected errors that were detected. The baseline for all fractions is the total number of errors injected. `Py` is the Python interpreter. `Py Only` are the errors detected by Python but not detected by UnnaturalCode.py. Similarly, the fraction of errors detected by UnnaturalCode.py appears in the `UC` column and the errors exclusively detected by UnnaturalCode.py are in the `UC Only` column. Then the table shows errors that were detected by both. `Either` is the union of detection by both methods and `None` are errors that are not detected.

Table 4.2: Fraction of Mutations Located by Mutation Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
DeleteToken	0.64	0.14	0.65	0.15	0.50	0.79	0.21
InsertToken	0.64	0.09	0.77	0.23	0.55	0.86	0.14
ReplaceToken	0.63	0.13	0.74	0.23	0.51	0.86	0.14
DeleteDigit	0.25	0.01	0.52	0.28	0.24	0.53	0.47
InsertDigit	0.33	0.02	0.62	0.31	0.31	0.64	0.36
DeleteSymbol	0.43	0.15	0.49	0.22	0.27	0.65	0.35
InsertSymbol	0.46	0.14	0.50	0.18	0.32	0.64	0.36
DeleteLetter	0.19	0.03	0.52	0.36	0.17	0.55	0.45
InsertLetter	0.31	0.03	0.58	0.30	0.28	0.61	0.39
Dedent	0.00	0.00	0.09	0.09	0.00	0.09	0.91
Indent	0.33	0.10	0.38	0.15	0.24	0.48	0.52

One example of a mutation that occurred during the experiment — that appears in the `UC`, `UC Only`, and `Either` amounts on the DeleteToken row in Table 4.2 — is the following code, which is missing the dot operator between self and discard:

```
def __isub__(self, it):
    if it is self:
        self.clear()
    else:
        for value in it:
            selfdiscard(value)
    return self
MutableSet.register(set)
```

Python does not report an error when running this code because the block containing the mutation is never reached, while UnnaturalCode.py reports the 20-token window indicated by the bold text above.

Assuming that UnnaturalCode.py is used in conjunction with the Python interpreter to improve error detection, the important data in Table 4.2 appear in the `Py` and `Either` column. For instance, for the set of programs used in the evaluation and for the random-edit insertions used, combining UnnaturalCode.py with the Python interpreter would improve the detection of token-replacement errors from 63% to 86%.

For all three types of token mutations, Python and UnnaturalCode.py perform comparably, with the combination locating 9-23% more of the total number of mutations than either Python or UnnaturalCode.py alone. This result is similar to the result obtained in Campbell et al. (2014) where interleaving UnnaturalCode.py and JavaC error messages always improved the score. Though the single-character and indentation mutations are harder for both Python and UnnaturalCode.py to detect, the combination of Python and UnnaturalCode.py detects the most mutations. Surprisingly, most indentation mutations did not cause errors on execution.

Another similarity between these experimental results and the previous results in Campbell et al. (2014) is that UnnaturalCode.py struggles more with deletion mutations than any other mutation type.

Table 4.3: Fraction of Error-Generating Mutations Located by Mutation Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
DeleteToken	0.85	0.18	0.71	0.04	0.67	0.89	0.11
InsertToken	0.70	0.10	0.81	0.21	0.60	0.91	0.09
ReplaceToken	0.70	0.14	0.75	0.19	0.56	0.89	0.11
DeleteDigit	0.74	0.04	0.83	0.13	0.71	0.87	0.13
InsertDigit	0.75	0.05	0.86	0.16	0.70	0.91	0.09
DeleteSymbol	0.71	0.25	0.56	0.11	0.45	0.82	0.18
InsertSymbol	0.77	0.23	0.63	0.09	0.54	0.86	0.14
DeleteLetter	0.67	0.09	0.73	0.15	0.58	0.82	0.18
InsertLetter	0.72	0.07	0.81	0.16	0.65	0.88	0.12
Dedent	0.00	0.00	0.03	0.03	0.00	0.03	0.97
Indent	0.71	0.21	0.60	0.09	0.50	0.80	0.20

Table 4.2 shows the performance of UnnaturalCode.py and Python under the assumption that every mutation is an error. However, this is clearly not the case for some mutations. This provides an upper bound on performance. In order to provide a lower bound, Table 4.3 shows the performance of Python and UnnaturalCode.py on the 11 types of mutations tested, while only counting mutations known to cause an error. By removing some data points that were counted against Python, Python's precision improves across the board and especially for deletion mutations. Python errors that are not near the

location of the mutation are still counted against Python's precision. UnnaturalCode.py performs similarly either way for the token mutations, but its extra contribution to precision when combined with Python is reduced for token deletion mutations.

Table 4.4: Fraction of Error-Generating Mutations Located by Token Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
NAME	0.75	0.08	0.87	0.20	0.67	0.95	0.05
OP	0.68	0.21	0.62	0.15	0.47	0.83	0.17
NEWLINE	0.17	0.01	0.51	0.34	0.16	0.52	0.48
STRING	0.61	0.05	0.83	0.27	0.56	0.88	0.12
INDENT	0.47	0.05	0.73	0.31	0.41	0.78	0.22
DEDENT	0.16	0.01	0.46	0.30	0.15	0.47	0.53
NUMBER	0.71	0.06	0.87	0.22	0.65	0.93	0.07

Table 4.4 shows the performance of Python and UnnaturalCode.py in relation to the lexical type of the token mutated, for token mutations. For token-replacement mutations, the type of the replacement token is considered, not the type of the original token.

One interesting result is that Python struggles the most with mutations involving token newlines and indentation tokens. Additionally, both Python and UnnaturalCode.py struggle the least with mutations involving names (identifier tokens).

Table 4.5 shows the frequency at which Python generates different types of errors based on which type of token mutation is performed. The "None" row indicates that Python did not detect a problem in the mutant file.

Many token mutations change the indentation of the code in Python. IndentationError is a type of SyntaxError, and therefore a large number of mutations result in some type of SyntaxError. The third most common outcome of running a mutant Python file is that no error is raised, and it is followed by relatively rare ImportErrors, NameErrors, and ValueErrors that are to be expected from mutations affecting library loading, identifiers, and literals. Other types of error occur, however they are extremely rare.

Table 4.5: Python Exception Frequency by Mutation Type

	DeleteToken	InsertToken	ReplaceToken
SyntaxError	0.60 (336117)	0.68 (383365)	0.61 (323125)
IndentationError	0.11 (62263)	0.23 (128356)	0.29 (152778)
None	0.25 (139131)	0.09 (47771)	0.09 (47522)
ImportError	0.02 (12338)	0.00 (23)	0.01 (2789)
NameError	0.01 (3177)	0.00 (355)	0.01 (4250)
ValueError	0.01 (4876)	0.00 (100)	0.00 (144)
TypeError	0.00 (1685)	0.00 (339)	0.00 (987)
AttributeError	0.00 (641)	0.00 (32)	0.00 (595)
OptionError	0.00 (5)	0.00 (15)	0.00 (135)
IndexError	0.00 (73)	0.00 (3)	0.00 (12)
error	0.00 (29)	0.00 (14)	0.00 (9)
InvalidInterface	0.00 (42)	0.00 (2)	0.00 (0)
KeyError	0.00 (3)	0.00 (12)	0.00 (5)
HaltingError	0.00 (3)	0.00 (8)	0.00 (6)
ImproperlyConfigured	0.00 (7)	0.00 (0)	0.00 (5)
ArgumentError	0.00 (1)	0.00 (1)	0.00 (4)
UnboundLocalError	0.00 (2)	0.00 (2)	0.00 (1)
OGRException	0.00 (3)	0.00 (0)	0.00 (2)
AssertionError	0.00 (1)	0.00 (0)	0.00 (2)
GEOSException	0.00 (2)	0.00 (0)	0.00 (0)
TemplateDoesNotExist	0.00 (0)	0.00 (1)	0.00 (1)
OSError	0.00 (1)	0.00 (0)	0.00 (0)
FieldError	0.00 (0)	0.00 (1)	0.00 (0)

Some deletions lead to mutant Python programs that do not contain syntax or semantics errors that could be raised by an automatic tool. For instance, deleting a unary "not" operator changes the semantics of the program, but this change does not cause identifier, syntax, argument or type errors. The results in Table 4.5 indicate that **25%** of token deletion mutations yield Python programs that can be successfully ran or imported. For most Python files this means that they successfully defined classes, variables and functions. However, some of the functions defined may not have been executed. It is unlikely that all 25% of the deletions that lead to no error detection actually resulted in error-free programs.

In comparison, the digit, symbol, letter and indentation mutations do not always lead to a change in the semantics of the python program. These muta-

tions can occur in comments or literals. Additionally, the indentation mutations, indent and dedent, may change a line of code but not affect the block structure if their change is to a single-line block of code. The results in Table 4.3 do not include any mutations that did not change the semantics of the Python program.

A significant number of token deletions leading to no automatic error detection has troubling implications for Python's use as an instructional language for beginning programmers. If any significant fraction of accidentally omitted Python tokens result in an erroneous Python program, but one that produces no error message when executed, many significant coding mistakes may not be reported to novice Python programmers by the Python interpreter.

Table 4.6: Fraction of Mutations Located by Exception Type

	Py	Py Only	UC	UC Only	Both	**Either**	None
SyntaxError	0.78	0.15	0.73	0.11	0.62	0.88	0.12
IndentationError	0.63	0.10	0.83	0.30	0.53	0.93	0.07
None	0.00	0.00	0.49	0.49	0.00	0.49	0.51
ImportError	0.98	0.07	0.93	0.02	0.91	1.00	0.00
NameError	0.95	0.04	0.96	0.05	0.90	1.00	0.00
ValueError	0.99	0.08	0.92	0.00	0.92	1.00	0.00
TypeError	0.45	0.06	0.92	0.53	0.39	0.98	0.02
AttributeError	0.49	0.04	0.87	0.42	0.45	0.91	0.09
OptionError	0.00	0.00	0.85	0.85	0.00	0.85	0.15
IndexError	0.02	0.00	0.90	0.88	0.02	0.90	0.10

Table 4.6 shows the performance of Python and UnnaturalCode.py in relation to the type of error seen by Python. For this measurement, a file for which Python does not detect an injected error is counted as an irrelevant result. Thus, the "None" reports zero mutations as detected by Python — it is not the fraction of files that do not contain mutations.

The results in Table 4.6 indicate that Python's ability to detect indentation error is rather poor: this is unsurprising and mirrors the examples shown in UnnaturalCode on Java (Campbell et al., 2014). While it is difficult to determine whether there is a missing indentation or an extra indentation (comparable to a missing `{` or an extra `}` in Java) using static analysis, it is easier for Unnat-

uralCode.py to locate such errors because UnnaturalCode.py has information about the author's or project's coding style and history.

Unfortunately, it is very difficult to determine why UnnaturalCode.py reports a location that is not the location of the mutation. This could be because of poor performance or because UnnaturalCode.py is reporting actual faulty code that is present in the code as it was released by project authors, before mutation. The experimental results are computed under the assumption that the code, as it was released by its authors, is syntax-error-free. Determining whether the code, as released by the authors, is actually syntax-error-free, or if UnnaturalCode.py is reporting locations near bugs that were shipped would require expert auditing of the locations suggested by UnnaturalCode.py.

The cumulative proportion of results falling into several range of distances is shown in Table 4.7. Each range is a column. Each report that counts toward the "0" proportion counts also toward the "0-1" proportion, "0-2" proportion, and so on. Python reports a line number with its error messages, and UnnaturalCode.py reports a 20-token window that may be one line, less than one line, or multiple lines long. The results for UnnaturalCode.py in Table 4.7 are computed by taking the distance between the mutation and the beginning of the window reported by UnnaturalCode.py.

If we only consider the exact line of the error, Python usually outperforms UnnaturalCode.py in terms of the location of the error, according to Table 4.7. However if we consider up to five lines before and after the reported error, UnnaturalCode indicates a line near the error more often than Python.

4.7.1 Comparison to UnnaturalCode with Java

Table 4.8 shows the performance of Python and UnnaturalCode.py compared to the results obtained previously for Java and the prototype version of UnnaturalCode. Only mutants that produced an error in Python are considered. Mean reciprocal ranks are reported for Java and both versions of UnnaturalCode. Precision is computed for Python as the proportion of times that Python reports the error on the same line as the mutation. This metric is very

Table 4.7: Distance in Lines of Code by Mutation Type

		0	0-1	0-2	0-5	0-10	0-20	>20
DeleteToken	Py	0.85	0.90	0.92	0.93	0.94	0.95	0.05
	UC	0.28	0.57	0.73	0.93	0.98	0.99	0.01
InsertToken	Py	0.70	0.70	0.70	0.71	0.74	0.77	0.23
	UC	0.35	0.65	0.80	0.95	0.99	1.00	0.00
ReplaceToken	Py	0.70	0.73	0.74	0.75	0.77	0.80	0.20
	UC	0.32	0.63	0.78	0.95	0.99	1.00	0.00
DeleteDigit	Py	0.74	0.75	0.76	0.77	0.79	0.81	0.19
	UC	0.20	0.43	0.59	0.77	0.88	0.93	0.07
InsertDigit	Py	0.75	0.82	0.83	0.85	0.86	0.87	0.13
	UC	0.17	0.45	0.63	0.85	0.93	0.97	0.03
DeleteSymbol	Py	0.71	0.84	0.88	0.91	0.91	0.92	0.08
	UC	0.30	0.53	0.67	0.84	0.92	0.95	0.05
InsertSymbol	Py	0.77	0.86	0.88	0.90	0.91	0.92	0.08
	UC	0.24	0.51	0.67	0.86	0.93	0.97	0.03
DeleteLetter	Py	0.67	0.70	0.71	0.73	0.74	0.77	0.23
	UC	0.17	0.44	0.62	0.84	0.92	0.96	0.04
InsertLetter	Py	0.72	0.81	0.82	0.83	0.85	0.86	0.14
	UC	0.18	0.46	0.64	0.85	0.93	0.97	0.03
Dedent	Py	0.00	0.01	0.02	0.05	0.11	0.16	0.84
	UC	0.25	0.48	0.66	0.92	0.98	1.00	0.00
Indent	Py	0.71	0.79	0.82	0.86	0.88	0.90	0.10
	UC	0.08	0.36	0.57	0.84	0.93	0.97	0.03

Table 4.8: MRR Comparison

	Python	Java	UC.py	UC.java
DeleteToken	0.85	0.92	0.74	0.87
InsertToken	0.70	0.93	0.83	0.99
ReplaceToken	0.70	0.93	0.77	0.98

similar to *mean reciprocal rank* (MRR) because MRR places as much weight on the first result as it does on the second through last results combined. Thus the results here differ in the methodology used to present results in Campbell et al. (2014) where MRR was used.

Given that Python produces at most one result, this metric is somewhat comparable to the mean reciprocal rank. The precision is computed for Python in the exact same way as the MRR was computed for the Java compiler in the case that the Java compiler only returned a single result. Though the Java compiler can produce up to 100 results, it does produce single results occasionally.

The results show that the new UnnaturalCode.py system does not perform as well as the prototype version of UnnaturalCode, and that Python's own error detection mechanism does not perform as well as the Java compiler error reporting. The distribution of MRR scores for the Java compiler and UnnaturalCode is shown in Figure 4.2.

As was found with the UnnaturalCode prototype for Java, some source files simply seem to be more difficult for UnnaturalCode.py than others. Files that seem to be difficult for UnnaturalCode.py have long sequences of tokens that do not appear elsewhere, such as lists of string literals. Figure 4.1 shows the distribution of UnnaturalCode.py, Python, and combined "either" precision over the files tested. Python also seems to have difficulties with specific files. Typically, random-edit mutations tend to produce random results from Python's own parser: regardless of whether Python's error messages identify the correct location, different mutations tend to cause Python to identify different lines as the source of the location.

On some files, however, such as `posixpath.py`, from the Python standard library, the Python parser often reports a parse error at the same location. In `posixpath.py`, this location is on line 182 and 72% of random-edit mutations cause Python to report an error at this location. This effect occurs only when the actual mutant line is after the line that Python's parser tends to point to. In the case of `posixpath.py`, the line that the Python parser consistently reports is shown below. This may indicate that Python's own parser struggles with multi-line statements when the line-continuation character \ is used. Multi-line statements such as the one shown below are typically handled dur-

ing lexical analysis in Python, but the errors observed orginate in the parser, not the lexical analyzer.

```
return s1.st_ino == s2.st_ino and \
```

4.8 Discussion

4.8.1 UnnaturalCode.py Performance

The results described in the previous section show that both UnnaturalCode.py and Python are able to locate mutations made to Python source files most of the time for most mutation types. UnnaturalCode.py is able to identify most mutations that cause a program to fail to execute. UnnaturalCode.py is able to locate some mutants that Python misses. Python is able to locate some mutants that UnnaturalCode.py misses. When used alongside Python, UnnaturalCode.py improves the chances of the correct location being reported to the user. UnnaturalCode.py struggles more with random deletion mutations than any other type of mutation.

Despite the fact that the Python version of UnnaturalCode.py has a lower MRR score than the Java version had, the MRR score still indicates that most of the time the correct result is in the top few. *These results indicate that the usefulness of natural language techniques used with programming languages includes scripting languages like Python.*

The Java version of UnnaturalCode was shown to perform much worse on code it had never seen before in Campbell et al. (2014). It is safe to assume that UnnaturalCode.py also performs worse on new code that it has not seen before. However, this is not the intended usage of UnnaturalCode.py. UnnaturalCode.py is designed to have good code added automatically to its corpus as soon as it is compiled or run. UnnaturalCode.py's corpus is therefore updated much more often than the project's source code repository. This allows UnnaturalCode.py to follow code evolution very closely. *Testing UnnaturalCode.py with completely unrelated test and training code would not relate to its real-world use case.*

4.8.2 Properties of Python

Python is a very flexible language with many useful and ultimately powerful features. But some of this power limits other aspects and properties of the language. The following differences between Python and Java needed to be accounted for in UnnaturalCode.py: 1) Python is not compiled; 2) Python programs may remove or add identifiers to or from any scope during execution; 3) Python types may change during execution; 4) Python has indentation-based syntax; 5) Python's lexical analyzer does not produce whitespace tokens; 6) run-time dependencies and requirements exceed compile-time dependencies and requirements; and 7) Python only produces at most a single syntax error.

No Compilation

Python is not a compiled language. This means that, in contrast with compiled languages such as Java, there is no oracle for basic Python program validity. The absence of an oracle creates several challenges for UnnaturalCode.py, and requires that both UnnaturalCode.py's goals and implementation be generalized.

The main challenge is that there is no way to tell if a Python program is valid. Executing the program will check it for basic syntactic validity, but some issues such as misspelled identifiers can not be checked merely by parsing the program. While there is a fine line between syntactic and semantic errors, due to their simple and typo-driven nature, UnnaturalCode.py is able to assist in locating semantic errors caused by typographical mistakes made when working with identifiers, expressions, values, and indentation. UnnaturalCode.py has no ability to discern between a syntactical Python error and a semantic Python error.

Determining if a Python program is valid is an undecidable question because making such determination requires running the program. So, the approach taken by UnnaturalCode.py is to run the program. Besides being an interactive scripting language, in Python any scope, name-space, type, constant, or library may change at any time during program execution.

A Python script may not necessarily halt. Furthermore, a Python program may execute without error even though it contains misspelled identifiers, broken imports, or other problems with the code that are easily caught in any compiled language.

The results in the previous section indicate that this is not merely a theoretical footnote, but that random-edit mutations produce cases in which it is difficult to determine the validity of a Python program a significant fraction ($\frac{1}{4}$) of the time.

The interactive, dynamic nature of Python has implications not just for the experiments presented in this chapter, but also any other experiment that depends on the execution of Python code of unknown quality. Techniques such as genetic programming and mutation would clearly be impacted: in order to be certain that programs produced by a genetic algorithm were valid in basic ways, the programs produced would have to be severely limited in terms of what mutations and combinations were allowed. Indeed, this is the approach taken in Derezińska and Hałas (2014), and Jester (Moore, 2000–2013). For example, one would have to ensure that no programs that referenced a variable before assignment were produced by limiting the set of possible output programs, because use before definition can not be checked easily after program generation. Another consideration is that generating Python code with correct indentation, or repairing Python indentation, requires more memory and time than checking pairs of braces.

Dynamic Scope

Even if a given Python program seems well-formed at one point during execution, it may not be at another point. For example, if a program uses an identifier, not only is that identifier not added to any scope until execution, but it may be removed or changed during execution. Holkner and Harland (2009) found in Python software advertised as production-stable quality that variables are created and removed, and their types changed *after* program ini-

tialization. Politz et al. (2013) discusses the differences between Python scope and traditional compiled languages.

Though the assumption that types, scopes and names will not change at runtime is tempting to make, it is not a given. For example, Django, a popular Python web-framework, makes use of this flexibility. This flexibility is used to enable developers to inject their own code and types into exceptions when debugging Python code.

The presence of runtime changes to built-in exception types in Python programs created a challenge for the implementation of UnnaturalCode.py. UnnaturalCode.py's built-in mutation testing system uses Python's standard `multiprocessing` library to run mutant Python code in a separate process, to implement the 10-second timeout, and to communicate the result of running the mutant Python code back to the main UnnaturalCode.py process. Sending the raw exception caused by the mutant code, if any, is impossible because the type of the exception generated by the mutant code may not exist in the master process, or may exist in a different form. Relying on serialization and deserialization for inter-process communication is unreliable because the types may be different. They may be different even when both processes are running the same program. Thus, UnnaturalCode.py must convert exceptions and other debugging information to plain strings.

In comparison to Python, other interactive scripting languages such as Perl may have fewer disadvantages compared to compiled languages such as Java. Idiomatic Perl programs declare all variables and scope by enabling a mode called `use strict`, enabling the Perl interpreter to check identifiers before execution (Hall, McAdams, et al., 2010). Additionally, Perl uses semicolons and braces to define the extent of statements and blocks, respectively, in much the same way that Java does.

No Constants

The ability to change semantics at runtime is not limited to identifiers. Python has no real concept of constants: even `math.pi` (π) may be changed during

execution. Furthermore, even the type system is not constant at runtime. For example, because libraries and modules are imported during execution, a Python program with two threads may lose the ability to communicate between its threads if one thread imports something that overwrites a built-in Python type. This is not merely a theoretical consequence, but one that actually occurs in some of the software used for the experiments presented in this chapter.

Furthermore, the lack of constants in python has direct impacts on the practice of software engineering when using Python. Dynamic typing has been shown in recent research by Hanenberg (2010), Kleinschmager et al. (2012), and Mayer et al. (2012) to either have negative or neutral impact on various aspects of software engineering. Even if you assume that types, constants, or names are not changed in surprising ways at runtime, test suites are a requirement to ensure some types of basic program sanity that compilers, optimizers, and static analysis tools provide automatically.

Any changes that the program made to the state of the Python run-time environment — including changes to local scopes, global scope, and types — are discarded by terminating the process after successful execution, exception, or failure to halt. This allows UnnaturalCode.py to protect itself from having its semantics disrupted by other software.

UnnaturalCode.py cannot improve the detection of coding mistakes, it only attempts to locate them once a problem occurs. Thus, it can be completely bypassed in several ways. The existence of a misspelled identifier in any relevant scope may not be verified if said identifier is used in a conditional block that does not run. Furthermore, SyntaxErrors produce exceptions that can be caught and ignored, which will prevent both Python and UnnaturalCode.py from reporting these errors.

Indented Blocks

Python's use of indentation to define code blocks creates a very different kind of lexical analyzer than most programming languages. While it is considered

usual to specify tokens with regular expressions and parse trees with context-free grammars using parser generators such as ANTLR (Parr and Quong, 1995), this is not the case for Python. In order to track indentation levels, even the lexical analyzer must be context-sensitive (Jim, 2012).

During lexical analysis, Python produces tokens of type "NEWLINE," "INDENT," and "DEDENT." These are roughly equivalent to semi-colons, opening braces and closing braces in Java, respectively. However, they do not necessarily correspond to whitespace in a Python source file. In fact, "DEDENT" corresponds to a lack of whitespace, and "NEWLINE" is only present once at the end of a multi-line statement.

UnnaturalCode.py employs a modified version of the official Python lexical analyzer, `tokenize.py`, which produces token streams even in the following cases: 1) bad indentation; 2) unterminated multi-line statements; and 3) unterminated multi-line literals. The modified lexical analyzer is only used for querying the language model. It is never used to parse or execute Python code. Changes made to the Python lexical analyzer were minimal: it merely continues in whatever state it was in before encountering an error, or assumes that the end of the file terminates multi-line statements and literals.

In comparison with other lexical analyzers, such as ANTLR (Parr and Quong, 1995), Python does not produce whitespace tokens. Instead, whitespace is tracked by computing the difference between the end of one token and the beginning of the next. Whitespace contains semantic information, even in languages where they are syntactically irrelevant. Though this information may be irrelevant to Python, it is relevant to the human developers. Since UnnaturalCode.py cannot see whitespace in Python, its efficacy both in practice and in the experimental validation procedure described in may be negatively impacted.

The experimental validation only performs mutations on tokens produced by Python's lexical analyzer, and therefore produces far fewer mutations involving whitespace changes, such as the space between an identifier and an operator, then could be produced if whitespace tokens were considered.

Runtime Dependencies

Python software may have dependencies at runtime. One example are libraries that work with database engines such as MySQL and that require not only MySQL libraries and development files, but also require MySQL to actually be running when the library is loaded with Python's `import` statement. Such requirements complicates the use of real-world software in validation experiments. Furthermore, the order in which libraries are loaded is occasionally important for successful execution: if they are loaded in the wrong order they may break.

Validation used only source files from open-source software that could actually run during the experiments. This is a significantly higher bar that source code must meet than merely compiling. One example that was encountered was that some Python modules depend on a configuration module, which must be written by the user, in order to be imported. This configuration model defines run-time behaviour such as database authentication. and passwords, which can not be distributed with the module.

The environment in which mutated Python code was executed included some initial imports to set up the global scope before running the mutant code. These imports were required because a few Python files would not execute unless another file was imported first from the same project.

One Error

Python only produces a single syntax error at most, and this syntax error is usually without diagnostics. In contrast, most other compilers, such as C and Java compilers, can produce many syntax errors for a single failed compilation. The reporting of multiple errors employs techniques such as those described by Kim and Choe (2001) and Corchuelo et al. (2002).

To be consistent with Python's behaviour this study reports only one result from UnnaturalCode.py to the user. UnnaturalCode.py is capable of producing as many results as there are contiguous 20-token sequences in the source file. However, given that Python only reports at most one syntax or other error

at a time, while `javac` reported 50 or more, UnnaturalCode.py is limited to report only one result so that it can be more easily used and compared with Python.

Python, could, perhaps be served well by implementing several features to mitigate the difficulties listed above. First, Python could add a mode similar to Perl's `use strict` that forces all variables to be declared with their scope. Second, Python could add variables, similar to Java's `final` variables, which can not be changed once set. Third, Python could prevent the run-time mutation of types, especially built-in and standard library types. The Python project has recently, as of 2014, announced future improvements to Python typing, such as type hinting, which will address this difficulty. Fourth, Python could employ the standard extensions of basic parser technology to produce better parser diagnostics, error recovery, and multiple-error messages in a similar fashion to C, C++, Java and many other languages, as recommended by Cannon (2005). Sippu and Soisalon-Soininen (1990) describes many of these techniques.

These suggestions and proposals should not interfere with the operation of most Python programs, and require minimal changes to those whose operation is affected. For example, software may need to be adjusted so that it uses a new type instead of overwriting a type from Python's standard library.

4.9 Threats to Validity

When comparing these results to the results obtained in UnnaturalCode Java, the validity of the comparison is threatened by several factors. First, the metric used in Campbell et al. (2014), MRR, is not directly comparable or convertible to the metric used in this chapter: the precision of the only result. The conversion of Python's precision at 1 to an MRR in Table 4.8 is biased against Python because it produces only one result. Second, the Python lexer produces different token types than Java's lexer. For example, Python's lexer does not produce whitespace tokens. This implies that the distribution of

semantic meanings of the mutations generated for Python code differs from the distribution of semantic meanings of the mutations performed for Java code. Third, Campbell et al. (2014) did not use mutations that passed compilation: rather, any mutation that compiled was discarded. However, those mutations were included here. Table 4.3 shows the performance of UnnaturalCode.py on Python programs that are known to be broken.

Python provides no mechanism to perform static analysis such as type checking, scope checking, and constant immutability checking. Thus, the experimental evaluation of UnnaturalCode.py provides results at both extremes: results assuming that every mutant would fail the checks in table 4.2, and results assuming that every mutant that executed successfully would successfully pass the checks in Table 4.3.

As an analysis of random-edit mutations in Python source files, the validity of these results is threatened mainly by the possibility that the uniform distribution of the mutations made in the evaluation do not reflect the distribution of mistakes made by humans when authoring Python software. In order to address this concern, both whole-token and single-character mutation experiments were performed. Whole-token mutations represent changes to program structure and single-character mutations represent typographical errors that a human may make.

4.10 Future Work

There are many other software engineering tools available for Python and their performance or some aspects of their performance may be characterized by using the random-edit-mutation testing tools presented in this chapter. These include tools such as `pyflakes`, and `pylint`. UnnaturalCode.py could be integrated into such tools as another method of critiquing code. A scientific comparison of the use of random-edit-mutation tool as opposed to a tool with fewer, specialized mutations such as Pester (Moore, 2000–2013), to characterize the coverage and effectiveness of test suites would also be useful.

Whether the mutations used in this chapter or the mutations used in other systems more accurately represent typos that are introduced by human software developers is an open question.

An extension of UnnaturalCode.py intended to provide feedback on possible errors while an author types would be useful. This feedback could then be compared to other systems that give similar feedback such as the Eclipse integrated development environment. Experimental results using Python 3.0 and other Python implementations would also be useful in order to track the changes over time and between projects.

4.11 Conclusions

Python, and other dynamically typed, interactive, scripting languages, pose many challenges to the Software Engineering experimenter. Due to Python's lack of static analysis and its complex lexical structure, *Python is a poor choice for generating and analyzing of source code of unknown quality.* The Python interpreter can not be asked if the identifiers in a Python program are valid, a question that the compilers for statically-typed languages answer easily.

This chapter provides a tool, and some techniques, for addressing these challenges. This tool may also be used when experimenting with genetic programming or other types of automatic code generation. These techniques may be applied when languages with a robust compiler is not available or desired. If a robust compiler is available, it should be employed. Because such a compiler can be used as an oracle for basic program validity including syntactic validity and identifier validity.

UnnaturalCode.py can augment Python's own error messages by suggesting a location for the developer to check. Often that is the location of a mistake, as shown in the "UC" column of Table 4.3. Using UnnaturalCode.py with Python improves the chances of locating a mutant piece, or mistaken piece, of code. Therefore UnnaturalCode.py can be a valuable tool for debugging language implementations such as Python because syntax errors appear

unnatural to language models trained with working source code. The n-gram language models can help find syntax errors when the model is built from a corpus of code known to be correct. Additionally, UnnaturalCode.py is able to locate some semantic errors, such as type errors induced by typos.

Python is a language that raises challenges for mutant-based experimentation: for instance, Python does not report as faulty 25% of programs with a single missing token. Python's use as an educational language needs further study and perhaps suitable supporting tools. Thus, when performing mutation-based experiments with "scripting" languages such as Python, researchers must be aware of the issues discussed in this chapter. Typical code errors that would be caught quickly and automatically by a compiler for a language such as Java can be difficult to automatically discover and report in Python.

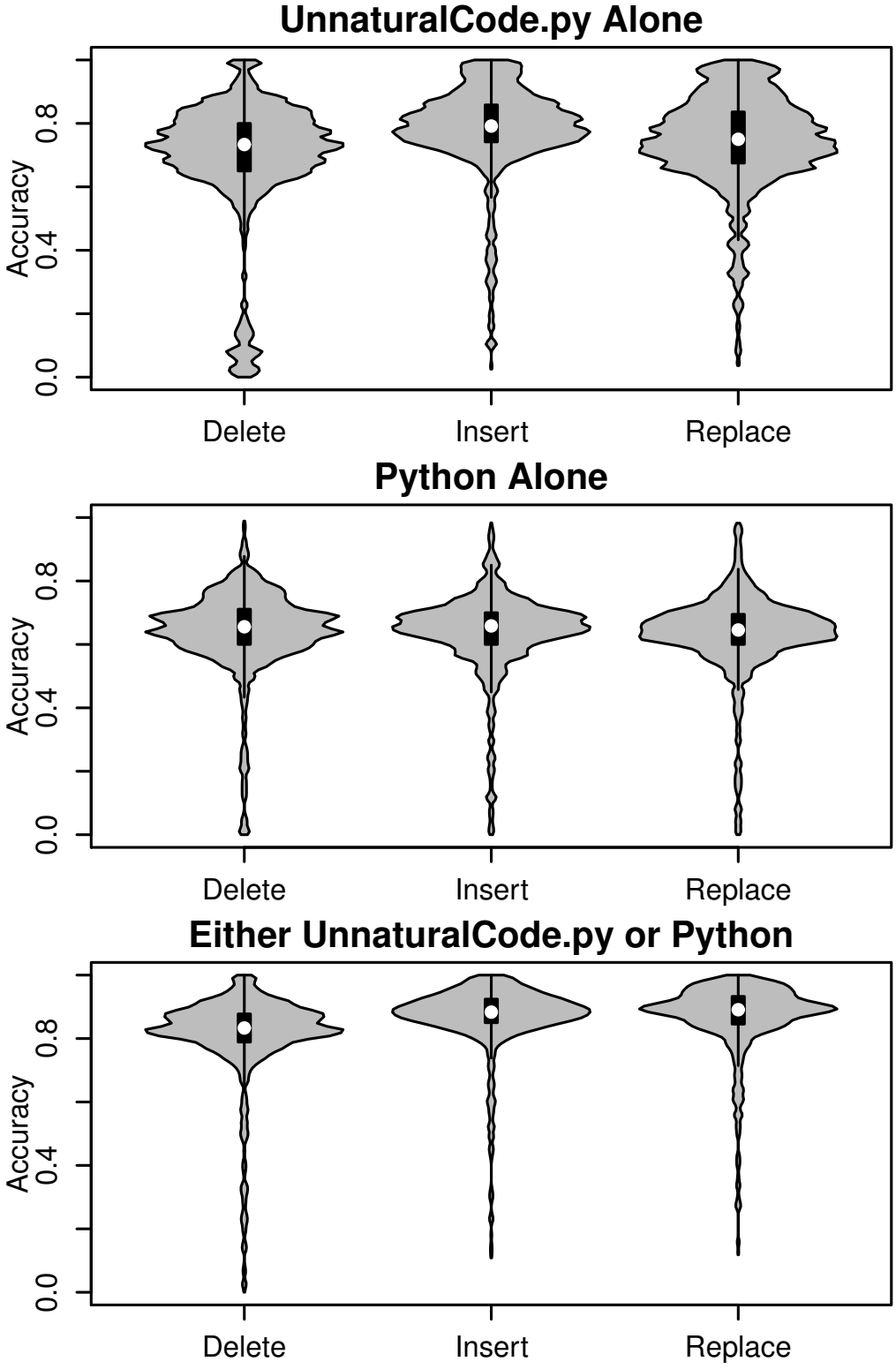

Figure 4.1: Independent and combined performance of UnnaturalCode.py and Python Width indicates the relative number of source files on which the precision level is achieved.

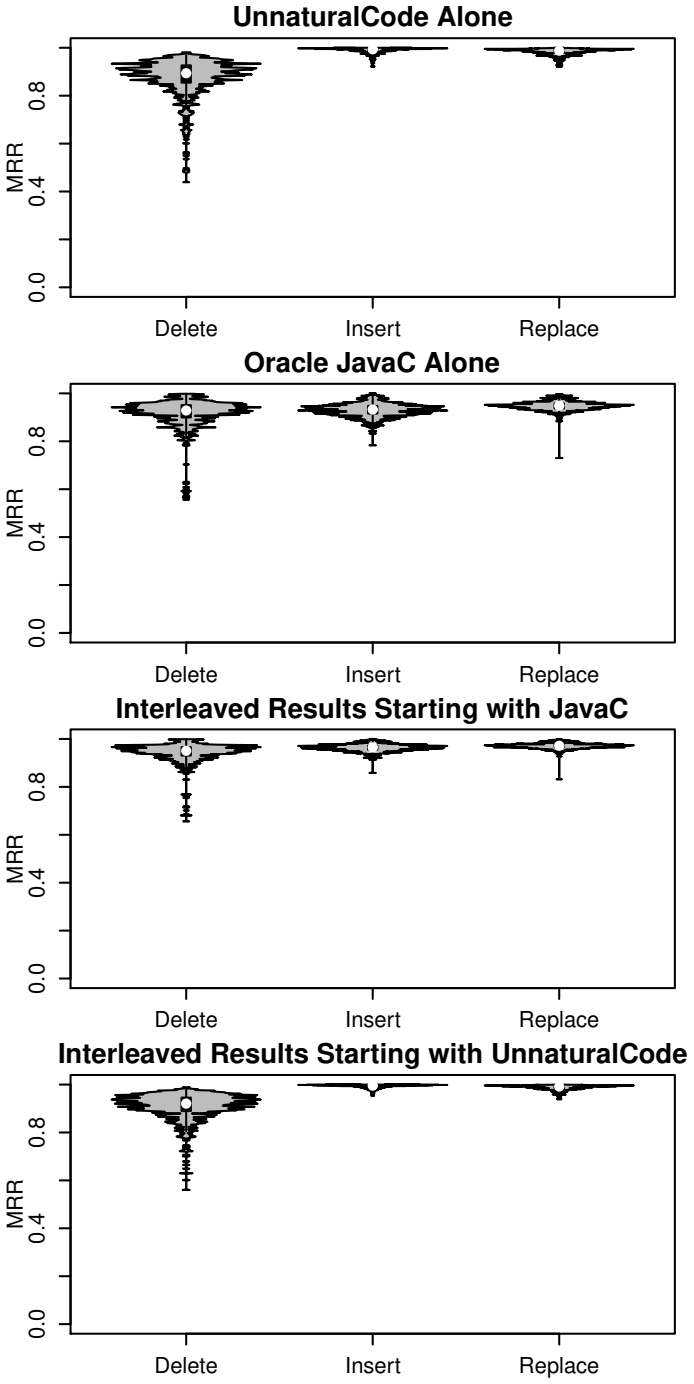

Figure 4.2: Independent and combined performance of UnnaturalCode Java prototype and JavaC, from Campbell et al. (2014). Width indicates the relative number of source files on which the MRR score is achieved.

Chapter 5

From Online Location to Offline Correction of Typographical Errors in Code

5.1 Preface

5.1.1 Acknowledgement

This chapter is based on work presented in the proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) as "Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors" (Santos et al., 2018). The experiments in that paper pertaining to n-gram language models were performed by myself. The experiments in that paper pertaining to LSTM neural network models were performed by the primary author, Eddie Antonio Santos. The paper was written by Eddie Antonio Santos and myself. Editing and supervision was provided by Abram Hindle and José Nelson Amaral. Dhvani Patel assisted in data processing and coding. The tables in this chapter are taken verbatim from that work.

5.1.2 Relationship with "Syntax and Sensibility"

Santos et al. (2018) focuses on the comparison of n-gram and LSTM (long short-term memory neural network) language models for finding and correcting syntax errors in Java code. Both models are trained on correct, syntactically

valid, source code. Then the models are queried for both the most likely location of a typo and the most probable fixes for that typo, based on their representation of correct source code.

This chapter focuses on n-gram model results, as well as providing details about the preprocessing and algorithms used. Instead of focusing on a comparison between LSTM and n-gram models as in the paper, this chapter focuses on the implementation and evaluation of n-gram models for automatically repairing source code. LSTM results from the paper are present in the tables, but there is no further discussion of the LSTM methodology.

5.1.3 Significance

At the time of writing (March 2021), Santos et al. (2018) had 36 citations (according to Google Scholar), including works describing other approaches to helping programmers work with compiler error messages (Ahmed et al., 2019). Some of those citations, were published in major journals and conferences, such as Chen et al. (2019), which already has 29 citations of its own after appearing in the IEEE Transactions on Software Engineering. Santos et al. (2018) is also listed in Monperrus (2020) Many of those 31 citations include extensions to the neural network work performed by Eddie Antonio Santos and presented in the paper. Thus, the n-gram work presented in this chapter remains state-of-the-art for typo location and fix suggestion generation with fast, on-line algorithms. I discuss the advantages that fast, on-line algorithms have over methodologies that require slow, off-line neural network training in Section 5.6.2.

5.1.4 Terminology

In this chapter, I will refer to the 10-gram model, coupled with the algorithms listed in this chapter as UnnaturalCode. However, UnnaturalCode as discussed in this chapter is a much more mature and refined version of UnnaturalCode than was discussed in previous chapters.

In this chapter, "typo" is used as defined in Section 2.1, while in chapters 3 and 4, "mistake" and "syntax error" were also used for the same purpose.

5.2 Introduction

This chapter is focused on evaluating a practical application of UnnaturalCode and comparing it to a similar tool based on a more advanced LSTM (long short-term memory) neural network. Additionally, this chapter focuses on providing UnnaturalCode in a form that would not require updating the corpus as done in previous chapters. Instead, the idea is to provide UnnaturalCode as if it were already trained, with a general purpose, fixed corpus based on known-good Java code. This chapter is also the first chapter to use real human-generated data for evaluation rather than applying mutations to known-good code.

5.2.1 Motivation

The motivation for the research presented in this chapter is largely the same as the motivation for the research presented in chapters 1, 3, and 4. However the same motivations pushed me to go beyond helping programmers find their typos and to move on to helping programmers fix their typos. This chapter is a natural progression of the prior work.

My peers had criticized the research done in earlier chapters was that the 10-gram model used was simple and uninteresting, so I also wanted to compare the 10-gram model to a more complex neural network model. I was motivated by the question: was I leaving typo locations on the table by using such a simple model?

Another common criticism of the research done in earlier chapters was that the evaluation was performed using synthetic typos in the form of code that was known to be good and then mutated. Thus, I also wanted to respond to those criticisms by performing an evaluation that was based on typos created by human programmers.

Santos et al. (2018) begins with this motivational example:

> Imagine a novice programmer writing this simple program for the first time, being greeted with three error messages, all of which include strange jargon such as `error: illegal start of type`. The compiler identified the problem as being *at least* on line 7 when the mistake is four lines up, on line 3. However, an experienced programmer could look at the source code, ponder, and exclaim: "Ah! There is a missing open brace (`{`) at the end of line 3!" Such mistakes involving unbalanced braces are the most frequent errors among new programmers (Brown and Altadmri, 2014), yet the compiler offers little help in resolving them.

The example of the novice programmer shows exactly knowing where a problem is may not be enough information to fix it. If they are not familiar with the language they may not be able to easily recognize a missing delimiter. They may not yet have an understanding of when braces should or should not be used. A compiler certainly understands braces and what they mean when used *correctly*, A compiler has understand every facet of the language to in order to do its job. However, compilers sometimes fail to produce intelligible errors and repair suggestions when faced with an input that does not meet the language's specification.

5.2.2 Prior Work

In 2014, the research in Chapter 4 was concluded, and Section 4.4 was written. Since then, research has continued on both the application of n-gram models language models to source code and the improvement of compilation errors resulting from typos. This chapter presents research that is a part of that ongoing work. Thus, the prior work mentioned in Chapter 3 and 4 remains relevant to this chapter. This section focuses on prior work since 2014 that is relevant to n-gram language models as well as typo location and repair. For a discussion of prior work relevant to the LSTM neural network research and results mentioned in this chapter, which was performed and evaluated by Eddie Antonio Santos, please consult (Santos et al., 2018) and (Santos, 2018).

The results in this chapter are produced by evaluating with code and typos collected in the Blackbox repository of student's activity (Brown et al., 2014), while using the BlueJ Java IDE (Integrated Development Environment) (Kölling and contributors, 2016; Kölling et al., 2003). The Blackbox repository is used for other research projects in Altadmri and Brown (2015) and Brown and Altadmri (2014). In a five-year retrospective, Brown et al. (2018) lists this work (along with Eddie Antonio Santos's neural-network) as the only work to apply machine-learning of any sort to the Blackbox data set.

5.3 Methodology

The UnnaturalCode methodology for this chapter's iteration of UnnaturalCode involves the following steps:

1. Provide UnnaturalCode with a ready-made general corpus of known-good Java code.
2. Wait for compilation of some code to fail.
3. Tokenize the input file for which compilation failed.
4. Generate windows of tokens from the tokenized input file.
5. Use the 10-gram model to estimate the cross entropy between each window and the corpus.
6. Interpolate each token's contribution to total cross entropy from the cross entropy of each window.
7. Rank tokens by their contribution to the total cross entropy.
8. Generate a list of possible repairs for the top-ranked tokens.
9. Compute the improvement (decrease) in cross entropy between a window centered on the repair before and after applying the repair and the corpus.

10. Rank repairs by how much they improve (decrease) cross entropy.

11. Present the top-ranked repairs to the programmer.

5.3.1 Tokenization

For the experiments presented in this chapter, two different tokenization schemes were used. In one, each token was taken directly from the token stream produced by the `javac` lexical analyzer from Java 8 SE. This is referred to as "concrete" in this chapter. The "concrete" model is trained using these exact tokens.

In the second tokenization scheme, which is referred to as "abstract" in this chapter, all identifiers are replaced with a single common token, `Identifier` and all literals are replaced with a token that represents the type of literal such as `String`. Table 5.1 gives examples of the tokens and how they are treated.

Table 5.1: Token kinds according to the Java SE 8 Specification Gosling et al. (2015), and whether we abstracted them or used them verbatim. From Santos et al. (2018).

Token kind	Action	Examples
Keyword	Verbatim	`if`, `else`, `for`, `class`, `strictfp`, `int`, `char`, `const`, `goto`, …
Keyword literal	Verbatim	`true`, `false`, `null`
Separators	Verbatim	`(`, `)`,`{`, `}`,`[`, `]`,`;`, `,`, `.`, `...`, `@`, `::`
Operators	Verbatim	`+`, `=`, `::`, `>>>=`, `->`, …
Identifier	Abstracted	`AbstractSingletonFactory`, `$ecret`, `buñuelo`
Numeric literal	Abstracted	`4_2L`, `C0FFEE`, `0755` `0b101010`, `.3e-02d`, `0xFFp+12f`, `' '`
String literal	Abstracted	`"hello, world"`

5.3.2 Locating syntax errors

In the exact same manner as the typo location techniques proposed in chapter 3 and 4, the technique evaluated in this chapter depends on a compiler or interpreter to determine whether input code contains errors or not. In the evaluation used for this chapter, the Java 8 SE compiler is used to determine

whether code causes compilation errors. The technique is presented under the assumption that the user has a Java compiler available to check code first, before any of the algorithms in this chapter are used. This workflow is described in detail in Section 3.5.

The n-gram language model uses up to 10-grams, the same as the language models used in previous chapters. However, the window length used for the validation presented in this chapter is 21 tokens. 21 token windows were chosen to remain consistent with the LSTM model for comparison in Santos et al. (2018). Previously, Chapters 3 and 4 used a window length of 10 tokens.

Querying the language model 21 "grams" at a time rather than 10 "grams" at a time results in the total cross-entropy of a 21-token sequence being reported, rather than the cross-entropy for a 10-token sequence being reported. However, the model used to compute the cross-entropy is still the same 10-gram language model. The entropy of a window $w = (w_1, \ldots, w_{21})$ is computed with the following formula:

$$\begin{aligned} H_{bits}(w, q) = &- \log_2 \left[q_{\text{modKN}}\left(w_1\right)\right] \\ &- \log_2 \left[q_{\text{modKN}}\left(w_2 \mid w_1\right)\right] \\ &- \log_2 \left[q_{\text{modKN}}\left(w_3 \mid w_2, w_1\right)\right] \\ &\vdots \\ &- \log_2 \left[q_{\text{modKN}}\left(w_{10} \mid w_9, \ldots, w_1\right)\right] \\ &- \log_2 \left[q_{\text{modKN}}\left(w_{11} \mid w_{10}, \ldots, w_2\right)\right] \\ &\vdots \\ &- \log_2 \left[q_{\text{modKN}}\left(w_{20} \mid w_{19}, \ldots, w_{11}\right)\right] \\ &- \log_2 \left[q_{\text{modKN}}\left(w_{21} \mid w_{20}, \ldots, w_{12}\right)\right] \end{aligned} \tag{5.1}$$

In Equation 5.1, q_{modKN} is the smoothed estimate of the conditional probability according to the model q. If the entire proceeding sequence of up to 10 tokens if that sequence has not been seen before the estimate will be computed using modified Kneser-Ney smoothing (James, 2000).

In order to keep the window length consistent, the file is padded at beginning with 20 "<s>" and at the end with 20 "</s>" tokens. Thus, each window contains 21 tokens, while at the same time, each token exists within 21 windows.

As in Algorithm 1, the n-gram model is queried for the cross entropy one window at a time, and can only produce the cross entropy for the entire window. It cannot produce the cross entropy for a single token. In order to find tokens that may be erroneous, it is necessary to find tokens that contribute the most entropy to the windows they exist within. In order to recover that information, each token from the source code is assigned a score, which is the average cross-entropy of the 21 windows that each token is a part of.

Finally, tokens are ranked from highest score to lowest.

Algorithm 1 Scoring each token's contribution to the cross-entropy

```
 1: function SCORE(tokens, window, q)
Ensure: tokens is a sequence of tokens
Ensure: window is a positive integer
Ensure: q is a 10-gram model
 2:     padding ← window − 1
 3:     header ← REPEAT(<s>, padding)
 4:     footer ← REPEAT(</s>, padding)
 5:     tokens ← CONCATENATE(header, tokens, footer)
 6:     l ← LENGTH(tokens)
 7:     scores ← a sequence of zeros
 8:     for i ← 0 to l − window inclusive do
 9:         entropy ← H_bits(tokens[i to i + padding], q)  ▷ all ranges inclusive
10:         for j ← i to i + padding inclusive do
11:             scores[j] ← scores[j] + entropy/window
12:         end for
13:     end for
14:     return scores
15: end function
```

5.3.3 Fixing syntax errors

Algorithm 2 shows how UnnaturalCode searches a limited space for a change to the code that will repair it. Using the same metric, the cross entropy between a window and the model. However, this time the windows being tested are

windows containing modified source code: each one is the result of applying a insertion, deletion, or mutation at a specific location. In order to limit the search space, only the locations of the *depth* tokens with the highest estimated contribution to the total cross-entropy are considered.

For each of the *depth* locations that are considered, Algorithm 2 generates a number of suggestions. Since there is only one way to delete a token, this provides the first suggestion: delete the token at the considered location. Then, for each token that has been seen by the model at least $minf$ times anywhere in the corpus, it generates a suggestion to insert the previously seen token after the token at the considered location and a suggestion to replace the token at the considered location with the previously seen token.

Most of the suggestions generated by this procedure will not produce enough of an improvement in the cross-entropy, if any. In order to save processing time, the algorithm uses *depth* and $minf$ as a simple heuristic to limit the number of windows with possible suggestions generated that would then need to be evaluated and ranked. Even though the task of computing the cross-entropy for a window is practically instantaneous, there are an astronomical number of possible suggestions that could be evaluated. Thus, this space needs to be limited.

In order to limit the search space of possible suggestions, the algorithm considers insertions, deletions, or replacements only for the *depth* windows with the highest entropies. The *depth* limit is based on the assumption that the typo is very unlikely to have produced a window with low cross-entropy. Additionally, the algorithm applies the assumption that tokens, such as literals, that appear very few times, (especially those that appear only once) are unlikely to be a good suggestion for a valid insertion or replacement token. Thus, insertion and replacement suggestions that would suggest inserting or replacing a very infrequently used token are skipped.

Each suggestion is given a score: the reduction (improvement) of the cross-entropy score for the window that the suggestion modifies. Finally, the suggestions are ordered by their score from greatest to least, and presented to

the user. With the assumption that the user will only consider the first few suggestions, they will only consider the suggestions that improve (reduce) the total cross-entropy of the file the most. The first few suggestions are the suggestions that seem the most likely to the model and would result in the least surprising code.

Algorithm 2 Searching for a change to the code that will repair it

```
 1: function FIX(oldTokens, window, depth, uniques, minf)
Ensure: oldTokens is a sequence of oldTokens
Ensure: window is a positive integer
Ensure: depth is a positive integer
Ensure: uniques is a list of all the unique oldTokens in the corpus, along
    with their frequency in the corpus
Ensure: minf is a positive integer
 2:     scores ← SCORE(oldTokens, window)
 3:     suggestions is an empty set of changes to the oldTokens along with
            their scores
 4:     for token ← each of the top-depth scoring oldTokens do
 5:         suggestions ←
                suggestions + TRY DELETE(token, oldTokens, window)
 6:         for unique ← each of the oldTokens in uniques do
 7:             if unique has frequency at least minf in the corpus then
 8:                 suggestions ← suggestions +
                        TRY INSERT(token, oldTokens, window, unique)
 9:                 suggestions ← suggestions +
                        TRY REPLACE(token, oldTokens, window, unique)
10:             end if
11:         end for
12:     end for
13:     return suggestions ordered by score from greatest to least
14: end function
```

5.4 Evaluation

UnnaturalCode, the 10-gram language model combined with the algorithms in the previous section, was evaluated on the following task: Providing UnnaturalCode with a general, fixed corpus for a given programming language as a ready-to-go tool that requires no training or on-line updating of the corpus to

Algorithm 3 Scoring potential repairs to the source code

1: **function** TRY DELETE($token, oldTokens, window$)
2: $side \leftarrow (window - 1)\,/2$
3: $center \leftarrow$ index of $token$ in $oldTokens$
4: $oldEntropy \leftarrow H_{bits}\,(oldTokens[center - side$ **to** $center + side], q)$
▷ all ranges inclusive
5: $newTokens \leftarrow oldTokens$ with $token$ removed at the appropriate location
6: $newEntropy \leftarrow H_{bits}\,(newTokens[center - side$ **to** $center + side], q)$
7: $score \leftarrow oldEntropy - newEntropy$
8: **return** $(score, newTokens)$
9: **end function**

10: **function** TRY INSERT($token, oldTokens, window, what$)
11: $side \leftarrow (window - 1)\,/2$
12: $center \leftarrow$ index of $token$ in $oldTokens$
13: $oldEntropy \leftarrow H_{bits}\,(oldTokens[center - side$ **to** $center + side], q)$
▷ all ranges inclusive
14: $newTokens \leftarrow oldTokens$ with $what$ inserted after $token$
15: $newEntropy \leftarrow H_{bits}\,(newTokens[center - side$ **to** $center + side], q)$
16: $score \leftarrow oldEntropy - newEntropy$
17: **return** $(score, newTokens)$
18: **end function**

19: **function** TRY REPLACE($token, oldTokens, window, what$)
20: $side \leftarrow (window - 1)\,/2$
21: $center \leftarrow$ index of $token$ in $oldTokens$
22: $oldEntropy \leftarrow H_{bits}\,(oldTokens[center - side$ **to** $center + side], q)$
▷ all ranges inclusive
23: $newTokens \leftarrow oldTokens$ with $token$ replaced by $what$
24: $newEntropy \leftarrow H_{bits}\,(newTokens[center - side$ **to** $center + side], q)$
25: $score \leftarrow oldEntropy - newEntropy$
26: **return** $(score, newTokens)$
27: **end function**

help programmers locate and suggest repairs for typos. Thus, I will answer the following research questions:

RQ1 Can UnnaturalCode, using a general, fixed corpus, locate human-made typos?

RQ2 Can UnnaturalCode, using a general, fixed corpus, suggest repairs for human-made typos that allow the code to compile successfully?

RQ3 Can UnnaturalCode, using a general, fixed corpus, suggest repairs for human-made typos that are the same repairs that the human who made the typo would use to repair their own code?

RQ4 How does using abstracted literal and identifier tokens impact UnnaturalCode performance?

RQ5 How does UnnaturalCode's performance compare to a similar tool using a LSTM neural network?

5.4.1 Novice Mistakes

The evaluation is run against actual novice code collected by the Blackbox project (Brown et al., 2014). This differs from the evaluation used in previous chapters, which used random mutations. The Blackbox project uses the BlueJ Java Integrated Development Environment (IDE) (Kölling et al., 2003) to record edit events, user actions that the novice programmers take while they are using BlueJ to develop, compile, run, and debug code.

The Blackbox data set was mined to find typos that were genuinely created by human programmers. The data also contains information about how the programmers fixed those typos. These are referred to as *true* fixes, which distinguishes them from fixes that do make the code compilable but are not what the actual novice programmer used to fix their code. Fixes that make the code compilable but are not the actual fix that the human programmer used are referred to as *valid* fixes.

Table 5.2: Edit distance of collected syntax errors. From Santos et al. (2018).

Edit Distance	Instances	Percentage (%)
0	10 562	0.62
1	984 471	57.39
2	248 388	14.48
3	93 931	5.48
4	54 932	3.20
5 or more	323 028	18.83
Total	1 715 312	

Table 5.3: Summary of single-token syntax errors. From Santos et al. (2018).

Edit Operation	Instances	Percentage (%)
Insertion	223 948	22.75
Substitution	77 846	7.91
Deletion	682 677	69.34

In order to evaluate the methodology described by this chapter, revisions of source code were selected. In particular, revisions where the code ”before” the revision caused a compilation error and the code ”after” the revision caused none. Additionally, only revisions where a single token was changed by the programmer to fix the compilation error were selected. Selecting only single-edit revisions removes ambiguity from the evaluation.

Table 5.2 shows how many tokens needed to be inserted, deleted, or replaced (the edit distance) for a revision to fix the previous revision and allow the revised code to compile without error. Some revisions fixed the syntax of escape sequences within long string literals. Since string literals are considered a single token, and often their full contents cannot be used in the model, these revisions are listed as having an edit distance of 0.

Table 5.3 shows, for the single-edit revisions used in this evaluation, the prevalence of each edit action. In Table 5.3, “deletion” indicates that the erroneous revision was missing tokens that were inserted in the next revision, while “insertion” indicates that the erroneous revision contained extra tokens that were removed in the fixed revision. “Substitution” means that a token was

Table 5.4: Number of tokens between partitions. From Santos et al. (2018).

	Mean	S.D.	Median	Min	Max
Train	8.16 M	596 019.30	8.06 M	7.45 M	9.00 M

Table 5.5: Abstract vs concrete model: Number of *unique* tokens between partitions

Partition	1	2	3	4	5
Concrete	229027	229027	208728	208707	226604
Abstract	110	109	110	109	109

replaced by a different token, either of the same, or of a different kind. Notably, missing tokens that need to inserted in order to get source code compiling accounted for almost 70% of the novice single-token typos. Substitutions, using the wrong token or token kind, were far more rare, accounting for only 8% of the single-edit revisions.

5.4.2 Big Java

Since the task I was evaluating UnnaturalCode for was one where the corpus was supplied pre-formed and not updated while the tool was being used, a corpus for the evaluation was obtained from popular Java projects on GitHub. Table 5.4 shows summary statistics of how many tokens were assigned to the mutually exclusive training data sets. All training data was mined from the 9993 most-popular Java projects on GitHub by downloading the latest revision of the default branch of that project's source code directly from GitHub. Each project was then randomly assigned to a training data set, and all of that project's source code files that compiled without error (about 99%) was added to that training data set. The training data consists of 2.3 million tokenized files.

Table 5.5 shows the stark difference in vocabulary size between the abstract and concrete corpuses. Concrete corpuses have a vocabulary that is on the order of two thousand times larger than abstract corpuses.

Table 5.6: MRRs of n-gram and LSTM models: performance from Santos et al. (2018)

Model	Qualification	1	2	3	4	5	All
10-gram Abstract	Location	.41	.42	.41	.40	.40	.41
	Valid Fix	.39	.39	.39	.38	.38	.39
	True Fix	.36	.36	.36	.35	.35	.36
10-gram Concrete	Location	.07	.07	.07	.08	.07	.07
	Valid Fix	.06	.06	.06	.07	.06	.06
	True Fix	.04	.04	.04	.04	.04	.04
LSTM 1	Location	.06	.05	.05	.05	.05	.05
(RMSProp)	Valid Fix	.06	.05	.05	.05	.05	.05
(no reshuffling)	True Fix	.05	.04	.04	.04	.04	.04
LSTM 2	Location	.52	.53	.53	.50	.50	.52
(Adam)	Valid Fix	.52	.53	.52	.49	.50	.51
(reshuffling)	True Fix	.46	.46	.46	.44	.44	.46

No Blackbox data was used in the corpus. Conversely, all evaluation data came from Blackbox novice typos, and no GitHub data was used for evaluating model performance. Thus, the corpus and evaluation data sets are all completely independent and mutually exclusive.

5.5 Results

Table 5.6 shows the performance of the 10-gram model with identifiers and literals abstracted, and verbatim, as well as the results obtained in Santos et al. (2018) for a LSTM model. This shows that the 10-gram model in combination with the algorithms presented in this chapter is able to locate and repair the test programs at least one third of the time. However it is not able to make repairs with correct identifiers and literals.

In Table 5.6, the columns labelled 1 through 5 are independent partitions of the evaluation data set, each with its own separation between test and training (corpus) data. The column labelled "All" is the average of the MRRs obtained for each partition. In all cases, the individual partitions are within 0.02 MRR

of each other, so I am confident that the results in Table 5.6 form an accurate representation of the data set, tools, and validation methodology.

In Table 5.6, rows labelled "10-gram Abstract" contain the performance of the algorithms presented in this chapter with a 10-gram model on code with identifiers and literals abstracted. Rows labelled "10-gram Concrete" contain the performance of the algorithms presented in this chapter with a 10-gram model on code with identifiers and literals left as-is. Rows labelled "LSTM" contain the performance of similar algorithms using an LSTM model with identifiers and literals abstracted. The two groups of rows labelled "LSTM" show results from two different LSTM models with different hyperparameter and training algorithm tuning.

Table 5.6 quantifies performance using MRRs, which are described in detail in section 3.6.1. In short, MRR indicates how many results, on average, a user would have to consider from a ranked list of results before finding one that meets some criteria.

The MRRs in Table 5.6 indicate how many suggested locations or repairs, on average, a human programmer would have to consider from the ranked list produced by UnnaturalCode after a failed compilation for some acceptable result. Rows labelled "Location" contain MRRs using the criteria that an acceptable result must specify the exact location of the missing, extra, or incorrect token causing the compilation error. Rows labelled "Valid Fix" contain MRRs using criteria the an acceptable result must propose a change to the source code that allows it to compile. In all cases, the proposed changes listed are the insertion, deletion, or modification of a single token. Rows labelled "True Fix" contain MRRs using criteria the an acceptable result must propose the same change to the source code that the human programmer used to repair the compilation error.

As an example, one could expect UnnaturalCode to automatically repair a missing curly brace one third of the time, while may not be able to repair a misspelled identifier even one twentieth of the time. The 10-gram Abstract results do not consider the spelling of identifiers at all.

In comparison, the LSTM model outperformed the 10-gram model, making repairs to abstracted code successfully almost half of the time.

The LSTM (Long Short-Term Memory) model, is a type of neural network. This model is explored in depth in Santos et al. (2018). The results for the LSTM model are included in this chapter as a comparison for the 10-gram model. The LSTM and 10-gram Abstract results presented in this chapter were produced by the same evaluation methodology at the same time with the same evaluation data using the same evaluation code.

5.5.1 Research Answers

For **RQ1**, the results show that yes, UnnaturalCode can sometimes (MRR of 0.41) locate human-made typos, even when using a general, fixed corpus, as long as those typos do not occur as misspellings within identifiers or literals.

For **RQ2**, the results show that yes, UnnaturalCode can sometimes suggest a repair (MRR of 0.39) that will allow the code to compile successfully, even when using a general, fixed corpus, as long as those typos do not occur as misspellings within identifiers or literals.

For **RQ3**, the results show that UnnaturalCode is almost as good at suggesting repairs that are the very same repair the human programmer would make as it is at suggesting repairs that simply allow the code to compile successfully (MRR goes down from 0.39 to 0.36).

For **RQ4**, the results show that when using a general, fixed corpus, using abstracted literal and identifier tokens significantly improves UnnaturalCode's ability to locate and suggest repairs for typos (MRR goes up form 0.07-0.04 to 0.36-0.41).

For **RQ5**, the results show that UnnaturalCode's ability to locate and suggest repairs for typos is similar to that of a similar LSTM neural-network-based tool (MRR of 0.36-0.41 compared to 0.46-0.51 for the LSTM tool).

5.5.2 Comparison with LSTM model

The LSTM model and its training program had many structural options and parameters that needed to be carefully selected and tuned. Tuning the LSTM model to the performance level obtained in Table 5.6 required a huge amount of work. 985 different configurations of the LSTM model's hyperparameters and training algorithms were evaluated (Santos et al., 2018).

In contrast, the 10-gram model had only two choices: the size of n, which was chosen in this case to be 10, and the type of smoothing, for which I chose modified Kneser-Ney, as in previous chapters. 10 was chosen for the size of n because it is relatively large. Typical values for n in literature employing n-gram models is around 3. However, $n = 10$ did not cause any memory or disk usage issues during initial experiments for Chapter 3. No other sizes of n were ever evaluated. No other smoothing methods were ever evaluated either formally or informally.

An off-the-shelf 10-gram model coupled with a fast, shallow search algorithm can achieve performance comparable to a sophisticated, highly-tuned neural network for the task of locating and fixing typos.

5.6 Discussion

5.6.1 Single-edit Repairs

As described in subsection 5.4.1, we only evaluated code that could be repaired with a single edit. This means that the evaluation results in Table 5.6 do not represent UnnaturalCode's ability, using any of the three models, to locate and repair code with multiple seperate problems.

More than half of the files that failed to compile in one revision and compiled successfully in the next were fixed with a change to a single edit to a single token, as shown in Table 5.2. If the Blackbox data set is representative of source code written by novices, our evaluation applies to their typos most of the time.

It is reasonable to assume that performance on code with multiple, seperate problems would change dramatically depending on evaluation criteria. For example, consider a piece of code that has two seperate problems. Under the expectation that UnnaturalCode should be able to locate and repair both problems performance would be much lower, because it can only repair a single problem some of the time. However, under the expectation that UnnaturalCode should be able to locate and repair at least one problem. Performance would be higher with multiple seperate problems, because it would have two or more chances at getting a single repair correct.

Consider a human programmer using UnnaturalCode as a tool to help them locate problems and make repairs to their own code. The human programmer may choose to repair one problem at a time, which is a reasonable approach. Thus, the second expectation may be the most appropriate. With additional experiments, the Blackbox data set could be used to quantify these assumptions.

5.6.2 Constrained Performance

As shown in Table 5.6, the 10-gram model performed worse than similar models in previous chapters. In this chapter, compared to the previous chapters, there were two main differences to the evaluation methodology. First, there was a hard split between testing and training data. Second, typos made by human students were used, rather than automatically generated mutations.

However, based on the results in Chapter 3, which involves testing a 10-gram model against code it has not seen, such as the results presented in figure 3.6 have already shown that using a 10-gram model to locate typos in Java code that the model has never seen before will yield poor results. Thus, relatively poor results, when compared to earlier chapters, in this chapter are not unexpected. Furthermore, there is evidence that the decrease in performance is due to splitting the data into exclusive training and testing sets.

As shown in Table 5.6 using concrete (non-abstracted) tokens performed very poorly. This poor performance comes from using the technique in a set-

ting where there is a firm division between the training and test sets. The training and test sets also have different vocabulary, especially for the "10-gram Concrete" experiment. As described in previous chapters, the technique can work in a setting where this division does not exist, so it is rather artificial. It should also be noted that the LSTM results from Santos et al. (2018) do not include a concrete version because the one-hot encoding used in the LSTM model is not capable of representing concrete tokens without becoming unreasonably large and slow to train.

Without being able to use previous iterations of the same codebase or related code as training data, the 10-gram model cannot work with identifiers. It has never seen the identifiers being used in the broken code before, due to the nature of splitting the data between a training set and a validation set. Thus, most identifiers in the code are novel to the model, and create high entropy tokens throughout the code. These high entropy tokens are then flagged by algorithm 1 and prioritized by 2 as locations in the source code where a repair could be made, however, these locations are usually not the location of the problem. Instead, the locations prioritized by 2 are simply the locations of identifiers that are unfamiliar to the 10-gram model. This essentially negates the 10-gram model's ability to locate defects. Since the model can no longer locate defects, it fails to suggest fixes for them as well. As shown in previous chapters, this performance degradation does not happen when the model is trained on compilable versions of the same code before the typo is introduced.

Another possible cause of reduced performance comes from the fact that the models used a corpus compiled from popular Java projects on GitHub. The code in popular Java projects on GitHub is likely produced by multiple experienced programmers working together, rather than a single student working alone. This may lead the model to represent how code from large, popular Java projects is written rather than how Java code is written by students.

5.6.3 What is a Correct Location

Algorithm 1 was not used in Chapters 3 and 4 for locating typo tokens because in those earlier experiments, location accuracy down to a single token was not required. For those earlier evaluations, the location of a single line or 10-token window was considered good enough, as the ranked locations were intended for direct output to the human programmer. For this chapter, in order to attempt to repair source code, the location of each specific, individual token in the source code needed to be ranked.

The evaluation performed in this chapter is not directly comparable to the evaluations performed in previous chapters. Importantly, it cannot be used to validate the mutation testing performed in previous chapters. A direct comparison between the evaluation using mutation and the evaluation using human-generated typos cannot be made.

5.6.4 Comparison with Previous Chapters

Even the results from Table 3.2 with a corpus built from Apache Lucene 4.0.0 and typos mutated from Apache Ant 1.8.4 are not comparable with the 10-gram concrete results obtained in this chapter. The results in Table 3.2 are with respect to locating a 10-gram window containing the code-breaking mutation, while the results in Table 5.6 are with respect the exact location of a single token.

It is reasonable to assume that the much narrower locations used in this chapter's evaluation are the cause of the MRRs listed for 10-gram concrete location, giving an MRR of 0.07, being much lower than the MRRs obtained with a corpus from Apache Lucene 4.0.0 and typos mutated from Apache Ant 1.8.4 giving an MRR between 0.20 and 0.36. In more human terms, this is a search tool providing the exact location of the typo or missing token as the 14th search result rather than an approximate location as the 5th search result.

There are several other possible explanations for the change in performance for the comparison described above. The first is that the Ant and Lucene projects, both being Java projects from one single organization, may have sim-

ilar coding style, coding conventions, and even shared code. The second may be that human typos are more difficult for the 10-gram tool to locate. The third may be that changes to the tool itself, as I rewrote and evolved UnnaturalCode's and MITLM's code over the years caused a decrease in performance.

Unfortunately, it is impossible to determine the exact cause of the apparent change in performance without additional experiments to evaluate each proposed cause listed above.

For the reasons discussed in this section, the evaluation methodology presented in this chapter also diverges significantly from evaluations in the previous two chapters. Comparing results from this chapter and the previous two chapters may have very limited validity.

5.6.5 General Limitations of this Evaluation

Unfortunately, because the evaluation presented in this chapter was designed to meet several requirements, it is also limited in its power to evaluate UnnaturalCode as it was intended to be used.

The evaluation in this chapter was designed to address common concerns raised by peer reviewers who reviewed the work done in Chapters 3 and 4. Peer reviewers were frequently concerned by the use of similar test and training sets, though of course, for the n-gram model this was entirely by design. Reviewers were also frequently concerned by whether or not the random mutations used in previous chapters accurately represented human-made typos for the purposes of evaluation.

Human-made typos were used and their repairs were used in this chapter's evaluation rather than mutations of known-good code. This constrains the amount of training and testing data available. The results may also be biased towards typos made by students rather than programmers at other levels of expertise. It is also biased toward typos made by users of the research IDE that was used to collect the data. Many of the typos in the evaluation data set may have been prevented by a different IDE.

As discussed in subsection 5.6.2, the test and training data sets were strictly divided in the familiar way used to evaluate algorithms for *static, offline* inference. However, the setting UnnaturalCode is meant to be used in, a development environment, combined with the fact that the 10-gram model can be updated quickly, means that UnnaturalCode can be used for *dynamic, online* inference.

This evaluation was designed to satisfy peer reviewers who had often asked for a hard division between test and training data sets. Additionally, this evaluation was designed to allow the direct comparison of the 10-gram model with the LSTM model. The LSTM model could not be updated quickly, which prevents it from being used online.

Additionally, since the evaluations performed for previous chapters, the 10-gram model used (MITLM) had also received a number of bug fixes that may affect performance. The code and algorithms that attempt to locate typos using the 10-gram model have also been rewritten and refined. The goal of these changes was to make the software more careful and precise, while also expanding the types of evaluations that could be performed. Thus, the results in previous chapters may diverge from the results in this chapter because of bugs fixes, algorithm changes, and code rewrites.

5.6.6 The Benefits of Cheap Models

While the LSTM model had better performance, getting that level of performance required tuning and multiple days to train the model with a high performance GPU, in 2017-2018. In comparison, the 10-gram model can be initialized and updated with additional data extremely quickly, typically in just a fraction of a second, and without the assistance of specialized hardware such as a GPU. In fact, training speed for the 10-gram model is primarily limited by available I/O bandwidth required to read the training data and write the model to disk.

For comparison, Santos et al. (2018) states of the LSTM neural-network models: “ Each model individually took between 2½ and 11½ days to train;

up to six models were trained simultaneously on two nVidia® GeForce® GTX 1070 GPUs. Section VII-A discusses how files were chosen for the training, validation, and testing sets." The same training sets were processed into a usable n-gram model in less than 10 seconds each: about 100 000 times faster, without the assistance of GPU acceleration. The time required to produce suggestions was similar for each type of model: 2-3 seconds.

The speed and ease with which the n-gram model makes updating the n-gram model *cheap*. The updating the model with new information is so inexpensive (in terms of time and hardware requirements) that updates can easily happen on-line and immediately when any new information becomes available, making UnnaturalCode usable in the workflow presented in Section 3.5.

Evaluating UnnaturalCode for its original intended workflow gives UnnaturalCode a huge boost. It is possible that this effect extends to other tools and techniques that rely on a corpus of source code. If a technique or tool is intended to provide a quick response at some stage of the software development process, and it uses some corpus or training data based on plentiful software artifacts, it may be possible to leverage a workflow similar to UnnaturalCode's.

This is true for techniques and tools requiring any task to be performed as long as that task can be performed with a model-driven algorithm. This includes techniques and tools that require classification, clustering, regression, anomaly detection, prediction, information retrieval, and search (search-based software engineering). As long as the model being used can be updated or retrained quickly enough, it can be updated or retrained with every commit, every compilation, or every change to a file. It may even be possible to update a model in real-time with every keystroke or action in an IDE. These *cheap models* bring with them several advantages.

With current hardware cheap models generally excludes neural network models and complex probabilistic graphical models (PGMs, for example, Latent Dirichlet Allocation) with slow inference schemes (for example, Gibbs sampling). Both neural networks and PGMs that require slow inference schemes

approximate a maximum value for some function with a large number of degrees of freedom by iterating. In the case of neural networks the computation cost of obtaining good results has has lead to booming business for companies (such as nVidia) that provide hardware to accelerate the process. Such hardware is becoming more common, and future advancements in computer hardware and neural network training may allow neural network models to be updated or retrained quickly enough to be used workflows with frequent model updates like the original UnnaturalCode workflow from Chapter 3.

In contrast, however, generating an n-gram model merely involves counting the number of times each sequence of tokens is seen and storing those counts in a look-up table. Even the advanced n-gram model software used, MITLM, does this. (However, the look-up tables computed by MITLM are specialized and organized cleverly for maximum performance while using minimal disk-space.) Computing the look-up tables needed to estimate the n-gram model can be done in less than a second, even for large projects with hundreds of thousands of lines of code old hardware.

One advantage of using cheap models is that models that are specific to a project, organization, or developer can be produced. Such models may outperform a generalized model trained on a massive corpus of a large number of projects. Of course, such a specialized model would also underperform when used with a different project, organization, or developer. However, because they are cheap to retrain, the model can just be thrown out and replaced with a new, specialized model.

By focusing on a single project, a model can focus on a limited vocabulary of terms, identifiers, and structures that are employed by that project. Focusing allows the model to represent terms, identifiers, and structures much more precisely in a way that would be considered over-training for a more generalized model.

A second advantage of using cheap models is that the model can "keep up" and "follow along" with the progress of the software development process, adapting each time it is updated or retrained. For example, as soon as a new

class appears in a Java project, the model can incorporate the identifiers, the "vocabulary," associated with the new class. It may be possible for the model to stay up to date regardless of whether it is being used for code completion, bug prediction, or any other model-driven SE task.

In summary, n-gram language models can be used to help human programmers find the locations of and suggest repairs to typos in source code that cause compilation to fail. n-gram language models are competitive with more advanced neural-network based models on this task when both are used in an off-line setting.

5.7 Conclusion

5.7.1 Automated Repair

For faults that can be detected by compilation and repaired with one single-token edit, a 10-gram model combined with a fast search algorithm can perform the repair automatically a significant fraction of the time.

However, the results presented in this chapter show that using a 10-gram model, combined with the algorithms described, can be used to quickly perform automated repair for some simple typos that prevent program compilation. In fact, this technique is so fast on modern hardware that it adds negligible time when compared to the compilation itself.

UnnaturalCode can provide this service to students, amateur, volunteer, and professional programmers with almost no overhead. The primary cost to using UnnaturalCode to automatically repair typos is set-up, since computational overhead is negligible.

Chapter 6

Statistical Properties of Crash Reports

6.1 Preface

6.1.1 Acknowledgement

This chapter was originally distributed as a "pre-print" (Campbell et al., 2016b). I performed the data collection. I also performed data analysis for most of presented results and figures. However, some of the analysis was specified by me, but performed by and plotted by my co-author, Eddie Antonio Santos. I most of the text, with some of the text also being composed by Eddie Antonio Santos. Dr. Abram Hindle helped with concept formation, editorial work, and supervised the research. The research presented in this chapter was funded by a MITACS Accelerate grant and BioWare, LLC.

6.1.2 Significance

Despite being distributed without peer review, the work presented in chapter has been referenced by one of its co-authors in Hindle (2019). People working in industry seemed more interested in the results than those in academia: I have received a number personal comments on how valuable this work would be if there were more of it available to industry that focused on different types and collections of software, different languages, etc. I intend to publish a shortened summary of these results in a practitioner-oriented venue.

6.2 Introduction

A software crash is any unintended abrupt termination of a program, often as the result of a programming error (Khomh et al., 2011). Software crashes may happen in deployed code, away from the developer's machine. When this happens, some systems create a *crash report*, summarizing the state of the program and its environment when it crashed. Telemetric crash reports are a major source of data about software quality that can be collected to identify bugs in software so that they can be fixed and the software can be improved. Because crash reports are so important, software companies such as Google (Google Inc., 2016), Canonical (Canonical Ltd., 2004), Mozilla (Mozilla Corporation, 2012), Apple (*Technical Note TN2123: CrashReporter* 2016), and Microsoft (Glerum et al., 2009) have *automated crash reporting systems*. These systems send crash reports from end users' machines to a centralized database, letting developers know what software is crashing and helping them figure out why. You may have experienced one of these systems if you have ever seen a dialog box that popped up asking if you wanted to send feedback when an application crashed (Figure 6.2).

Most crash repositories rely on automated systems (Mozilla Corporation, 2012) that deduplicate and cluster the crash reports, so that hopefully each crash bucket (cluster) has crashes relating only to one fault in the program. Crashes must be deduplicated automatically, as the quantity of crash reports being uploaded is vast. Campbell et al. (2016a) indicated that Mozilla received 2,189,786 crash reports in a single week alone. There is a substantial body of prior work (Bartz et al., 2008; Campbell et al., 2016a; Dang et al., 2012a; Dhaliwal et al., 2011; Glerum et al., 2009; Kim et al., 2011; Modani et al., 2007; Seo and Kim, 2012a; Wang et al., 2013) that has tackled automated crash report deduplication and triage; however none of these works focused on the properties of crashes and buckets or how those properties are distributed. Unfortunately the true accuracy (measured in *precision* and *recall*) of these

automated systems is unknown, because they have not been validated with a gold set of true duplicates.

Because of the big data nature of the problem, sources of manually deduplicated bug reports are rare. For this reason, we analyzed the Ubuntu Launchpad data set, which contains manually-deduplicated buckets of crashes—crashes that a human determined to be caused by the same underlying software fault. The data set also contains crashes that may have been neglected, receiving little or no attention. The Ubuntu data set contains crashes from 1,921 different applications, programs, and libraries made available on Ubuntu. This type of crash repository data mining and fundamental descriptive statistics has not been performed in the literature previously, to our knowledge.

Due to the manual labor involved in manually bucketing crashes, datasets like the Launchpad crash repository are both rare and small. Thus, we wish to extract as much useful information from the dataset as possible.

Understanding crash repositories and characterizing the properties of crashes that can be found within is important not only to deduplication and triage. For example, large crash repositories could be used to provide feedback on how crashes are collected, how software is designed, or what hardware is problematic. One can leverage crash repositories by instrumenting crashes in a way to aide debugging (Liblit et al., 2005). Another question that can be answered by analyzing crash repositories is how often recursion was involved in a crash. We can also find out what actions cause a large number of crashes. Intuitively, functions on the top of stacktraces are viewed as the most important features for deduplication, and we can use the Ubuntu data set to investigate those functions.

The Ubuntu data set is a valuable resource because it contains thousands of buckets of crash reports that were hand-marked as duplicates by Ubuntu developers and volunteers. The Ubuntu project has limited human resources to carry out this task, however, leaving many crash reports neglected and not yet deduplicated. Unfortunately it is difficult to discern crash reports that are truly unique from ones that were merely never given enough attention to be

assigned a duplicate. Thus, the Ubuntu data set does not contain a gold set of unique crashes.

Fortunately, the Ubuntu data set contains crashes from 1,921 different applications, programs, and libraries made available on Ubuntu. This includes a wide variety of software such as, GUI applications, command line tools, specialized libraries, standard libraries, compilers, server software, video games, scientific software, and more.

We attempted to characterize the data set statistically to inform crash report bucketing techniques and the validation methodologies for those techniques. We hope that these statistical characterizations will be helpful to the developers of crash bucketing tools and the engineers deciding how to triage and handle crash reports. In addition, we want to understand the Launchpad crash repository to understand the impact that validating crash report bucketing tools with this data might have.

While there are a large number of aspects of the data set that could be studied we chose 10 aspects in the form of research questions (RQ) for this chapter. RQ1 to RQ3 focus on how crashes are organized into groups in the data set. RQ4 to RQ5 focus on categorical properties of crash reports. RQ6 to RQ8 focus on the size of stacktraces, recursion and names. RQ9 and RQ10 focus on common functions and libraries that crash.

6.3 Terminology

To understand the domain of crash report deduplication, we provide definitions of the terms used in this chapter.

A *crash* is any unintended, abrupt termination of a program. These are often due to *bugs*, or erroneous programming caused by *human errors*. Human errors lead to *faults* in programs, faults can lead to *failures*, and failures can include *crashes*. We use bugs as an umbrella term to cover human error, faults, and failures, while crashes are the result of bugs. One common crashing bug

Context data

ProblemType: Crash
DistroRelease: Ubuntu 11.04
Package: rhythmbox 0.13.3-0ubuntu6
ProcVersionSignature: Ubuntu 2.6.38-8.42-generic 2.6.38.2
Uname: Linux 2.6.38-8-generic x86_64
Architecture: amd64
Date: Wed Jul 6 06:50:47 2011
ExecutablePath: /usr/bin/rhythmbox
Signal: 11 (SEGV)
SourcePackage: rhythmbox

Stack Trace

#0 0x00007f08c4128740 in ?? ()
#1 0x00007f08dd022a56 in g_hash_table_lookup_node () at ghash.c:313
#2 g_hash_table_remove_internal () at ghash.c:1093
#3 0x00007f08e1ae059c in remove_child () at rhythmdb-tree.c:1514
#4 0x00007f08e1ae0631 in remove_entry_from_album () at rhythmdb-tree.c:1535
#5 0x00007f08e1ae4675 in rhythmdb_tree_entry_set () at rhythmdb-tree.c:1608
#6 0x00007f08e1acf8b6 in rhythmdb_entry_set_internal () at rhythmdb.c:3345
#7 0x00007f08b5f98fc1 in update_tracks () at rb-audiocd-source.c:1178
#8 0x00007f08b5f9919a in update_tracks_string () at rb-audiocd-source.c:1191
#9 0x00007f08b5f9924c in update_album_cb () at rb-audiocd-source.c:1212
#10 0x00007f08e0ca8578 in _gtk_marshal_BOOLEAN__BOXED () at gtkmarshalers.c:86

[37 stack frames omitted...]

#48 0x00007f08dd0349f2 in g_main_loop_run () at gmain.c:3299
#49 0x00007f08e0ca5af7 in IA__gtk_main () at gtkmain.c:1257
#50 0x0000000000404f4c in main () at main.c:336

Stack Frame

Depth	Address		Function name	
`#1`	`0x00007f08dd022a56`	in	`g_hash_table_lookup_node`	`()`

	Filename	Line Number
at	`glib/ghash.c`	`313`

Figure 6.1: An example crash report (Schaaf, 2012): its contextual data, its stacktrace, and a dissection of one of its stack frames. Note that some information was removed for space.

occurs when a program tries to access memory that is not allocated to it. This is called a segmentation fault on Linux and Unix-based systems.

Crash reports (Figure 6.1) can be generated when a program crashes. Such reports typically contain information about what was going on in the program when it crashed such as the stacktrace, file and line number of code that was executing when the program crashed, and what was in memory at the time of the crash. They also typically include information about the system the program was running on such as what type of computer it was running on, what other programs were running, and what version the program and any libraries it used.

A *stacktrace* (Figure 6.1, middle) is a snapshot of the execution state of a program. A stacktrace is a sequence of *stack frames* (Figure 6.1, bottom) that describe the current function, on the *top of the stack*, followed by the function that called the current function—the second function on the stack—and so on. The entry point of the program is the *bottom of the stack*. Included in a stack frame is usually, the address of the function, the name of the function, and the source file that defined the function. Sometimes, a stacktrace also includes the line number in the source file that defined the function, or the line number that was executing at that point in time. Additional information such as arguments and their values at the time each function is called may also be included.

A *crash stacktrace* is the stacktrace captured at the time that a crash occurred. In this case, the top of the stack indicates the function that encountered the crash, but this function does not necessarily contain the bug that *induced* the crash. The contents and formatting of crash stacktraces varies wildly depending on what language the software was written in, what platform it was running on, what tools were invoked to generate the crash stacktrace, and what libraries the software was using.

Bug reports (sometimes called issue reports) are user- or developer-submitted reports, usually in a standard format, that detail a problem that the user or developer experiences with a piece of software.

Crash buckets are groups (or clusters) of crash reports, in our case, in the form of bug reports, that have been manually or automatically determined to represent crashes caused by the same underlying problem.

In addition to the stacktraces, *contextual data* may be included such as: what version the software and other pieces of software on the system are, the system architecture and OS version, settings the user may have modified, and information about hardware installed on the system such as sound and video cards.

Stacktraces may contain *recursion*, which occurs when a function calls itself. Stacktraces may also contain *mutual recursion*, when multiple functions call each other in a sequence with the last calling the first. In a stacktrace with recursion, there are several instances of the same function.

Unbounded recursion is when a function calls itself endlessly without finding a terminating condition. In C and C++ programs, each function call allocates stack space.[1] Thus, unbounded recursion will allocate stack space endlessly until the operating system refuses to allocate more pages to the program, crashing the process with a segmentation violation (`SEGV` signal as described in Section 6.6.4).

Applications, programs, and libraries are treated as *packages*, or pieces of software that can be installed onto Ubuntu systems, using the Ubuntu system's package manager. These packages come from source packages that typically consist of a single piece of software from a single source repository, plus bug fixes and customizations made for Ubuntu.

6.4 Prior Work

There have been a few prior works that have investigated the properties of crash report repositories. These works primarily seek to determine causes of crashes, in contrast to this this chapter that seeks to describe and characterize

[1] A compiler may be able to apply the tail-call elimination optimization to recycle the same stack frame, however this is an optional feature provided by some compilers (*Options That Control Optimization—GCC online documentation* 2016).

the nature of the crashes from one particular crash repository. Much more prior work has created tools to deduplicate crash reports for the ultimate purpose of aiding developers in fixing common crashes in production. Our work is far more empirical, seeking to understand the properties and structure of crash reports, and is similar in intent to that of Herraiz et al. (2007) who sought to characterize distributions of version control system repositories to study software growth.

6.4.1 Studying crash report repositories

Kechagia and Spinellis (2014) mined crashes from Android applications written in the Java programming language. The authors recorded the exceptions that crashed the software compared to the exceptions that were said to occur according to the Android API documentation and found that many of the crash causing exceptions were not mentioned at all in the documentation.

Ganapathi and Patterson (2005) collected crashes from local research machines running Windows XP SP1. Their analysis revealed that web browsers crashed most frequently of all types of applications, even more so than the more widely-used document preparation software. The software that did crash was due to users manual termination due to an application hang. The authors remarked that shared libraries (`.dll` files on Windows) invoked by multiple applications accounted for about 15% of all application crashes.

This was followed up by Ganapathi et al. (2006), which analyzed Windows XP *kernel* crash data collected from machines on the Berkeley Open Infrastructure for Network Computing. They measured several aspects such as kernel crashes per user, and system uptime before a crash. They concluded that many kernel crashes are the result of unreliable device drivers; 10 device driver vendors were responsible for over 75% of kernel crashes out of the 110 vendors attested in the crash corpus.

Gómez et al. (2015) created MoTiF to generate *in vivo* crash test suites that create the shortest sequence of actions to reproduce a crash experienced

by a user. They used crowdsourced test suites to bucket crashes in order to give developers a sense of what needs to be fixed.

Like Campbell et al. (2016a), this chapter focuses on one particular crash repository—the Ubuntu Launchpad data set. Unlike the prior work, this chapter places a greater emphasis on describing the statistical tendency of crashes. Our data set is different from the ones studied in the prior work in a few important ways. We collected crashes from Linux machines, so some of the research questions in this work are Unix-specific, as opposed to Windows-specific as with Ganapathi's two prior works (Ganapathi et al., 2006; Ganapathi and Patterson, 2005). As well, the crashes came from lower level languages; despite the presence of exceptions in C++, ultimately what we captured were the Unix signals that crashed the applications.

6.4.2 Stacktrace crash report deduplication

In order to prevent as many crashes as possible, it is necessary to find out how many users are experiencing what kinds of crashes, categorize those crashes, prioritize them, and triage them. Deduplication is necessary to group and count similar crashes that are hopefully caused by the same fault in the software. Many papers use stacktraces—either exclusively, or combined with contextual data—to deduplicate crash reports. Brodie et al. (2005b) introduced the concept of bug localization by comparing stacktraces and finding the "best" sequence of consecutive function names to create a signature for a crash. This was followed up by Brodie et al. (2005a), wherein the authors describe several heuristics for preprocessing stacktrace data, including stop word removal and recursive call removal. Bartz et al. (2008) created a call stack similarity classifier based on learned weighted edit distance. Modani et al. (2007) proposed several stacktrace similarity metrics including edit distance, longest common subsequence, and top-of-the-stack matching. Dhaliwal et al. (2011) described using average weighted Levenshtein distance between stack frames to measure the similarity between crashes. Kim et al. (2011) deduplicated already bucketed crashes by constructing a directed graph using stack frames

as nodes, and their relation to one-another as directed edges, and then applied a graph similarity metric. Seo and Kim (Seo and Kim, 2012a) studied crash stacktraces in order to select previously bucketed crashes that may reoccur, and are thus worth the triaging and debugging effort to fix. Dang et al. (2012a) used agglomerative hierarchical clustering on crash stacktraces for deduplication. Wang et al. (2013) used crash stacktraces to train a Bayesian belief network to find files that are likely the cause of a crash. Note that many deduplicators (Brodie et al., 2005a; Dang et al., 2012a; Modani et al., 2007) who used stacktrace edit-distance found it necessary to remove recursion in the preprocessing stages. Lerch and Mezini (Lerch and Mezini, 2013) addressed deduplication by using tf–idf to determine the similarity between crash stacktraces in order to deduplicate *bug* reports as they are being written. Campbell et al. (2016a) followed this up by directly addressing *crash* report deduplication by comparing various tokenization methods and the parameter of similarity threshold to bucket crashes.

6.5 Data

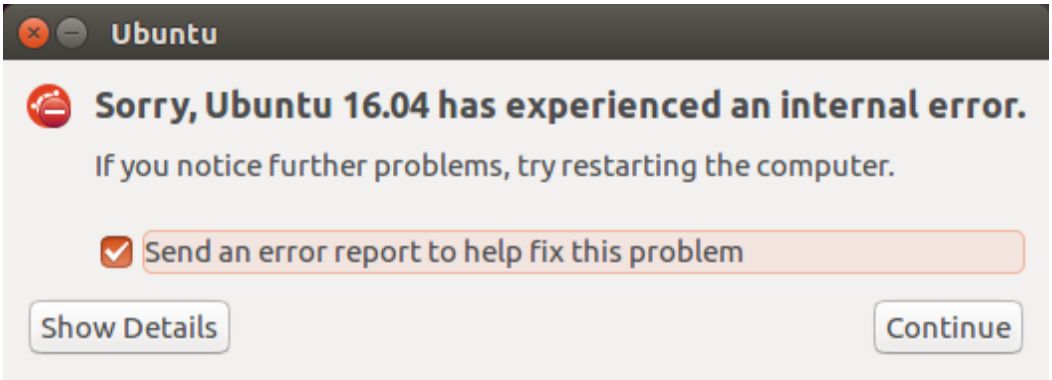

Figure 6.2: The dialog that appears when an Ubuntu application crashes. A user can opt-out of sending the crash report to Canonical.

This chapter uses the Ubuntu Launchpad data set that consists of 40,592 crash reports that are taken from bug reports for 1,921 packages. In Ubuntu Launchpad, crash reports are submitted as bug reports by a user, but with tooling that helps fill out the contextual data automatically. The tooling

that collects the information from the crash report to the bug report also tags the bug report so that it is possible to separate out bug reports that were completely manually generated from those that were generated with the assistance of the crash reporting tool `apport`.

6.5.1 How do crashes make it into *Launchpad*?

On standard installs of Ubuntu Desktop and Ubuntu Server, the Apport system collects the stacktrace and metadata, such as the CPU architecture and the signal that crashed the program. According to the Ubuntu Wiki: "If any process in the system dies due to a signal that is commonly referred to as a 'crash', […] the Apport back-end is automatically invoked" (Ubuntu Wiki contributors, 2012). Apport does this by installing itself as the program that receives the *core dump* by setting `/proc/sys/kernel/core_pattern` (*Piping core dumps to a program—core(5) Linux User's Manual* 2015). The Linux kernel invokes Apport with the core dump, which contains the full state of the program at the point of the crash. The core dump is loaded into `gdb` to extract a string representation of the crash stacktrace (Figure 6.1, middle). Apport presents the user with a window indicating that their app has crashed (Figure 6.2). At this point, the user may choose to opt-out of sending the crash report. If the user does not opt-out they are prompted to log in to Ubuntu Launchpad, and add their comments to a form that is pre-filled with all of the data automatically collected by the Apport system.

6.5.2 Why *Launchpad*?

We chose the Launchpad data set for several reasons: the data set is freely available to researchers,[2] so our observations are reproducible. Similarly, most of the crashes occurred in open source software, allowing researchers to find the source code online and analyze any interesting faults (as we show in Section 6.6.7). Since the crashes come from a myriad of applications and packages on desktop machines, we can analyze crashes that were caused by shared

[2] `https://pizza.cs.ualberta.ca/lp_big.json.xz`

libraries—in contrast with Mozilla's crash database that only contains crashes from Firefox. Most remarkably, crashes that were marked as duplicates on Launchpad were labelled *manually* by volunteers and employees of Canonical. This creates somewhat of a gold set with which researchers can evaluate the precision and recall of their automated crash deduplication algorithms (Campbell et al., 2016a). This set of human-curated crash buckets are invaluable for evaluating crash deduplication methods.

Since most of the software in Ubuntu is open source, most crashes can be debugged if they were not caused by system instability, cosmic rays (Hwang et al., 2012), or other random phenomena. Even though some of the crashes are more than 6 years old, the version of the code that crashed is usually still available.

6.5.3 How were the crashes obtained?

The corpus was downloaded using a modified version of Bicho (Robles et al., 2011) that is available online.[3] This process took a considerable amount of time (more than a month) due to the fact that the Ubuntu Launchpad API throttles requests. Additionally, much of the data required for the crash reports are uploaded as attachments that must be downloaded separately. These were downloaded using `wget`. 126,609 issues were downloaded, of which 44,464 contained stacktraces from C or C++ programs. Of those, 3,872 were thrown out because they were unparsable, leaving 40,592 crash reports. These crash reports come exclusively from software that is compiled to a standard Linux binary and can be debugged with GDB, the GNU debugger. This limits the data set to crashes from software written in C, C++ or other similarly compiled languages.

6.5.4 What do the crashes look like?

Each crash report consists of semi-structured text information. An example is shown in Figure 6.1. The text was parsed with a Python script written by the

[3] `https://github.com/hazelybell/Bicho`

authors.[4] The initial text is automatically generated by Apport, but the user is free to modify anything they want. This results in mostly machine-readable crashes. Unfortunately, some crashes are unparsable or contain unparsable metadata.

Additionally, because the data set contains crashes from 2009 until 2015, the various software that produced the crash reports, the Launchpad system, and the standard format of reports changed over time. Some reports, when parsed contain the wrong information in the wrong place. For example, if there is a missing newline in the crash report data, information about which package a crash occurred in might mistakenly appear in the field meant to describe the signal that crashed the application. Our parser cannot correct these types of problems in the data. However, this type of problem is rare, occurring in less than 1% of crashes.

Most crashes contain more than one stacktrace. This is because there are stacktraces of varying types available for a single crash. For example, for many crashes, both a normal stacktrace is available along with stacktraces for any other threads that might be running simultaneously in the application when it crashed. We only use the single stacktrace (of the crashing thread) in our experiments.

Additionally, several versions of the same stacktrace of the same crash are often available. This can be caused by information that was missing on the user's computer being automatically filled in later by the Ubuntu Apport system. We attempt to use the stacktrace with the additional information by default, if it is available.

6.6 Research Questions, Methodology and Results

We answer 10 research questions about the Launchpad crash repository. The first 3 answer how crashes are organized within the crash repository:

[4] `https://github.com/naturalness/partycrasher/blob/master/partycrasher/launchpad_crash.py`

RQ1. How are crashes distributed among applications?

RQ2. How are crashes distributed among buckets?

RQ3. How long do crash buckets last?

The latter 7 research questions are focused on the properties of individual crashes and how they are distributed:

RQ4. What Unix signals are crashes associated with?

RQ5. What CPU architectures experienced the crashes?

RQ6. How long are crash stacktraces?

RQ7. How are crashes associated with recursion?

RQ8. How long are function names in crashes?

RQ9. What are the most common functions in crashes?

RQ10. What are the most common crashing libraries?

6.6.1 RQ1. How are crashes distributed among applications?

In order to fix the causes of crashes, it is first necessary to find out what applications, programs, or libraries the crashes occur in. If a few pieces of software crash far more often, it may make sense to allocate developer time or other resources by focusing on them. Or, if crashes are spread out among many pieces of software, it might indicate that widely-used libraries are a common cause.

In order to determine which applications crashed we looked at each crash report's `SourcePackage` metadata field, or if that was not available, the crash report's `Package` field. Ubuntu applications are often broken up into multiple packages, but they will usually all share a single `SourcePackage`. 44 crash reports were missing this information (or it was unparsable) so we looked at the remaining 40,548 crash reports.

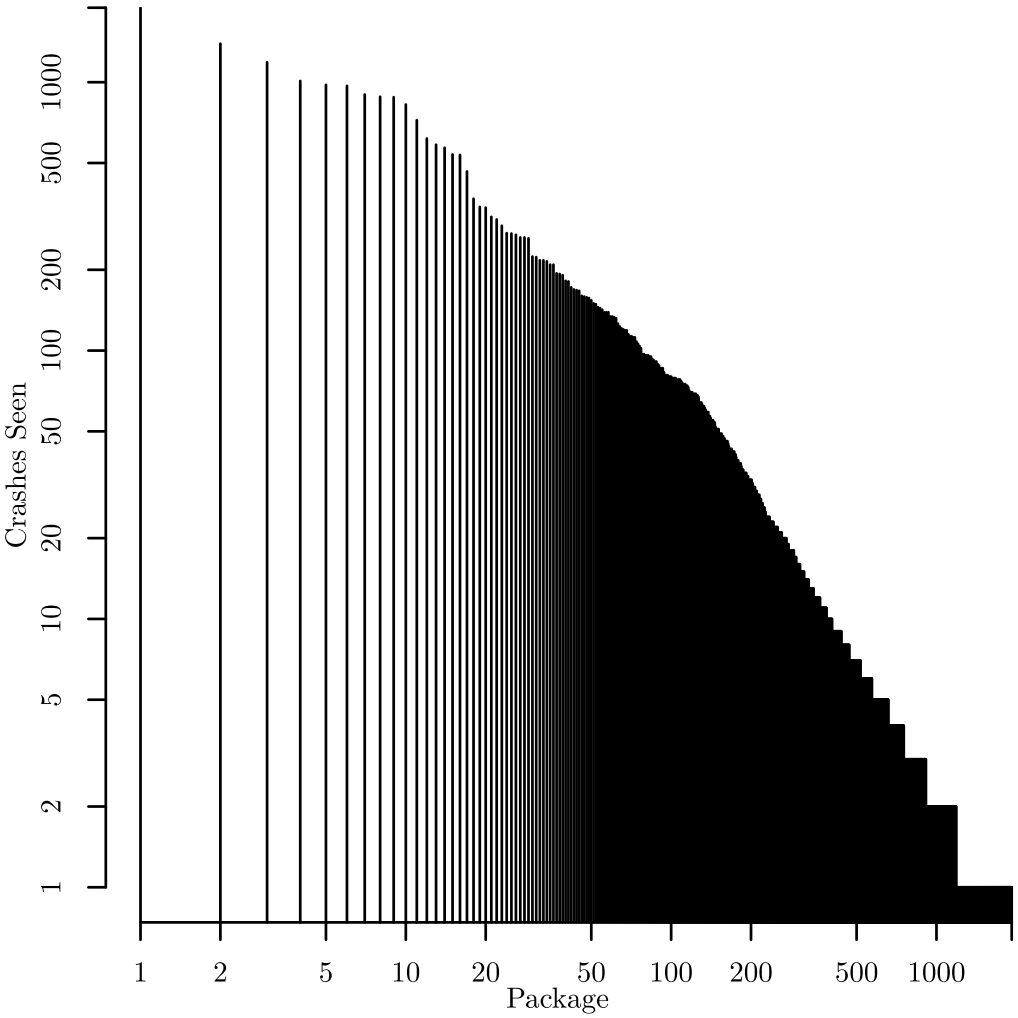

Figure 6.3: Crashes per package. Both axes are logarithmic.

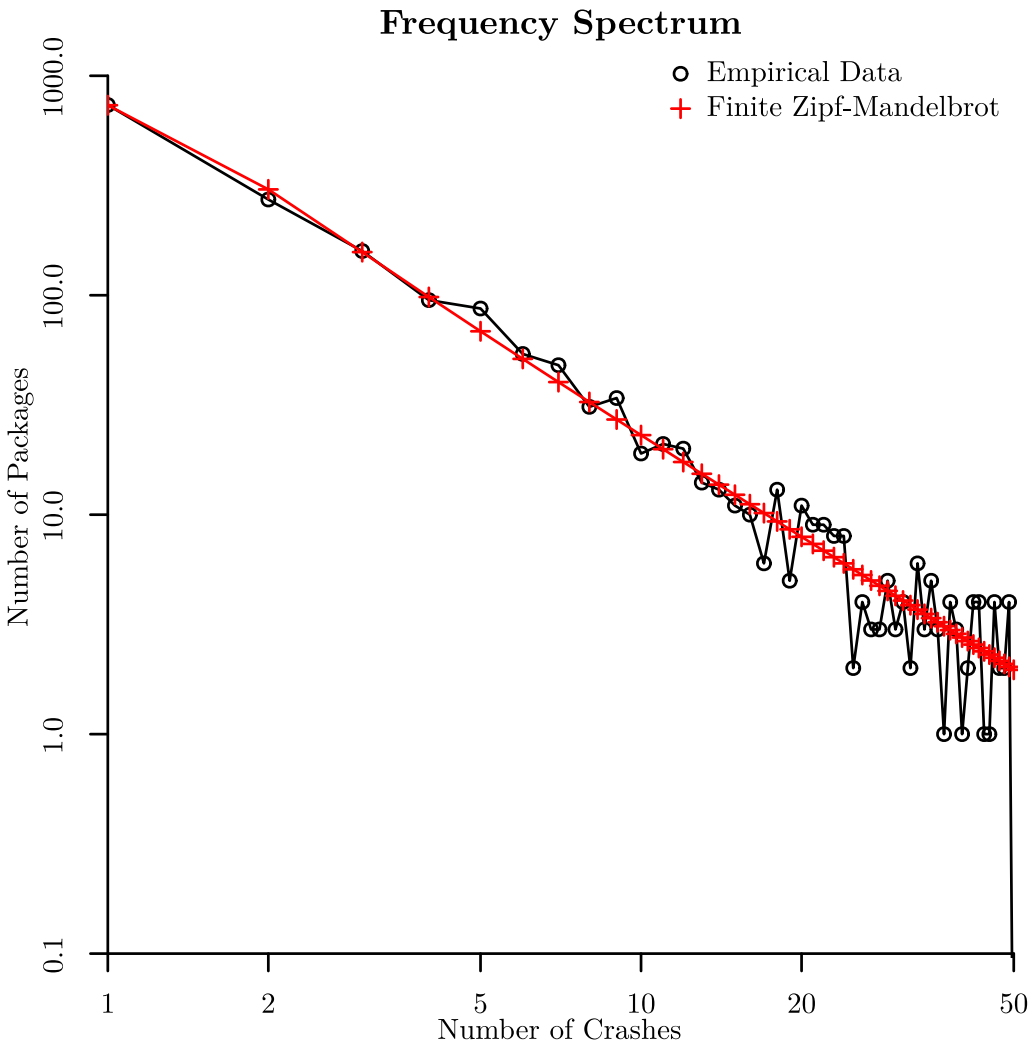

Figure 6.4: Frequency spectrum of crashes by package, indicating there are many packages with only one crash report, and a few packages with many crash reports. Both axes are logarithmic. The leftmost point on the plot indicates there are 736 packages with only a single crash.

Minimum	Maximum	Median	Mean	Std. Dev.	Skewness	Kurtosis
1	1895	2	21.1	94.8	10.8	150.4

Table 6.1: Descriptive statistics for crashes per package.

The package with the most crashes was Nautilus with 1,895 crash reports. In total, 1,921 packages had crash reports. A log-log plot of the distribution of crash reports is shown in Figure 6.3. This plot shows the packages with the most crashes on the left and the fewest crashes on the right. Additionally, a log-log frequency spectrum plot of the distribution is shown in Figure 6.4. The frequency spectrum shows how many packages have 1 crash, how many packages have 2 crashes, and so on.

Figure 6.3 indicates that the data might follow a broken power-law (Zipf's law) distribution because it appears to have several linear appearing decreasing segments when plotted with both axes being logarithmic. The plot is very similar in shape to plots of word frequency in English-language texts, such as the plot of the word frequency in English Wikipedia text by Grishchenko (Grishchenko, 2006). This indicates that the distribution of crashes per application has a piecewise power law distribution.

We fit a finite Zipf-Mandelbrot model to the frequency spectrum using the `zipfR` R package (Evert and Baroni, 2007). In Figure 6.4, the red line indicates the model distribution and the black line indicates the empirical data. We used the χ^2 goodness-of-fit test to determine how well the model fit the data. The test determined a p-value of 0.17. However this p-value is not small enough to reject the hypothesis that the data came from the model distribution.

We can conclude that the distribution of crashes among packages follows a power-law distribution. Therefore, a few applications will cause a large number of crashes, and there will also be a large number of applications with only a few crashes. Crash deduplication systems must be able to handle deduplicating both crashes from software that crashes rarely and software that crashes frequently.

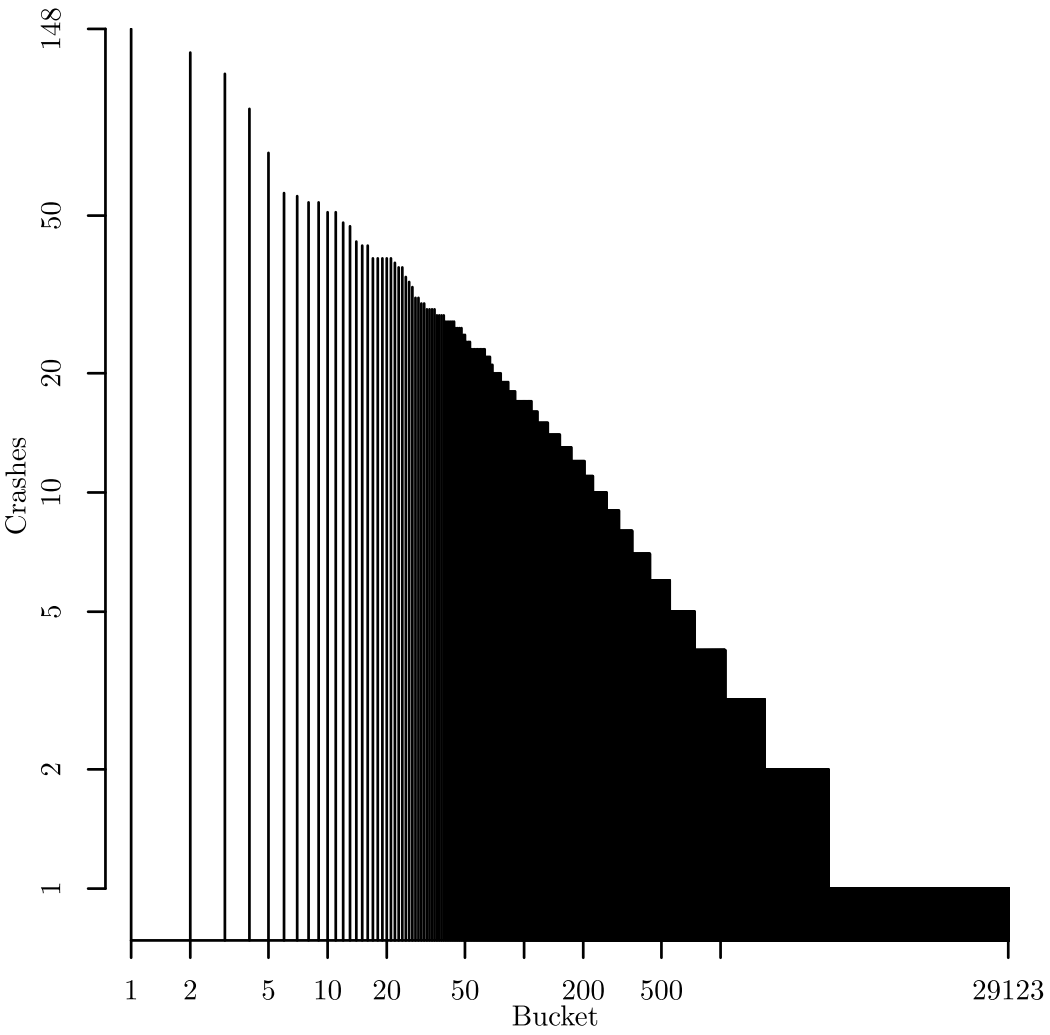

Figure 6.5: Crashes per bucket. Both axes are logarithmic.

6.6.2 RQ2. How are crashes distributed among buckets?

Minimum	Maximum	Median	Mean	Std. Dev.	Skewness	Kurtosis
1	148	1	1.4	2.5	24.0	989.4

Table 6.2: Descriptive statistics for crashes per bucket.

Understanding the distribution of crashes in buckets is critical to effective crash report deduplication. It might also be useful for predicting how many major buckets (groups of crashes that are caused by a single underlying bug) will appear based on how many new crashes appeared. Thus it may be use-

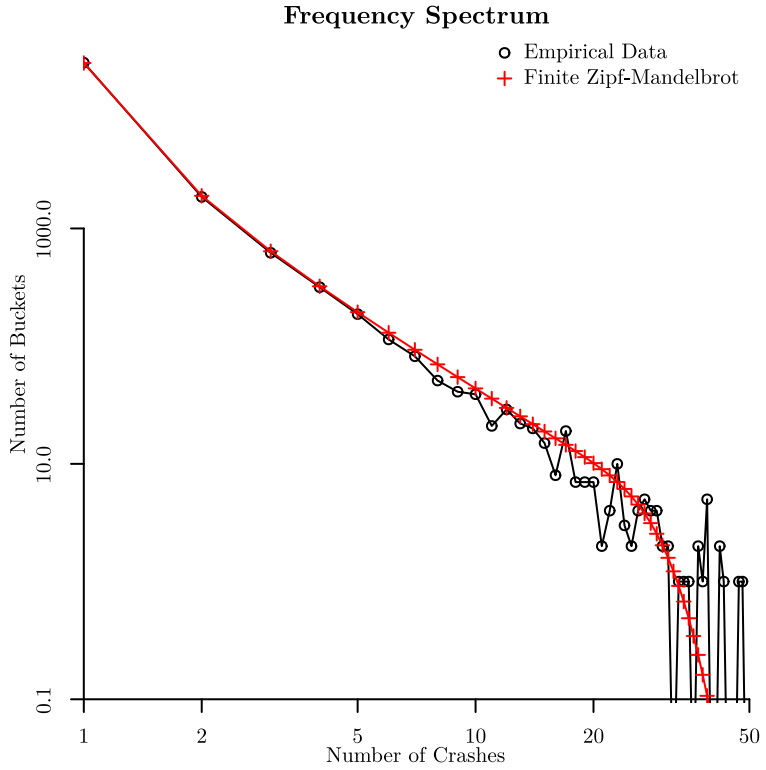

Figure 6.6: Frequency spectrum of crashes by bucket, indicating there are many buckets with only one crash report, and a few buckets with many crash reports. Both axes are logarithmic. The leftmost point indicates there are 25,594 buckets with only a single crash.

ful for predicting how many new crash-causing faults have appeared in the software once the number of new crashes is known.

We downloaded all of the crash reports available at the end of 2015. Our modified version of Bicho also downloaded information about which crash each crash was marked as a duplicate of, if any. This information is available from the Launchpad API (Launchpad contributors, 2016) along with the crashes. It is provided by the API as a field indicating the bug report each bug report is a duplicate of (crashes are stored as bug reports in the same system). We used this information to group crashes into buckets. Then we sorted the buckets by size and plotted the result.

Figure 6.5 shows the distribution of crashes among buckets. The largest bucket had 148 crashes. The crashes concern a library package called `gtk-sharp` that is used as a GUI toolkit for many applications in Ubuntu. These packages include CD burning applications, photo management applications, and others, all of which crashed due to this library bug.

In addition, there are 1,853 buckets with only 2 crashes in them, and there are 25,594 buckets with only a single crash. Buckets with only a single crash were not used in Campbell et al. (2016a) because it is not known if they are truly unique or simply have not been evaluated by Ubuntu developers and volunteers enough to be assigned to a larger bucket. In this chapter, we consider crashes with no known duplicates as being in a bucket by themselves.

Buckets with only a single crash in them account for more than half of the crashes. There are only 14,998 crashes in the data set that are in buckets with other crashes.

We fit a finite Zipf-Mandelbrot model to the frequency spectrum using the `zipfR` R package (Evert and Baroni, 2007). In Figure 6.6 the red line indicates the model and the black line indicates the empirical data. We used the χ^2 goodness-of-fit test to determine how well the model distribution shown fit the data. The test produced a p-value less than 10^{-13}, indicating that the data did not come from the fit distribution. However, Figure 6.6 clearly shows

that the model distribution is able to capture the general shape of most of the empirical distribution.

Much like RQ1, we can conclude that a many crashes will be lonely, that is, being the only crash in a bucket, with no duplicates, while some buckets will be significantly larger, containing a large number of crashes. Crash deduplication systems must be able to correctly place crashes into both small buckets and very large buckets. This means that any deduplication technique that tends toward some average size of bucket will perform poorly. Most importantly, this data set indicates developers can address large amounts of the crashes that users report by focusing on only a few crash buckets.

6.6.3 RQ3. How long do crash buckets last?

Min.	Max.	Median	Mean	Std. Dev.	Skewness	Kurtosis
0	1.28×10^8	0	6.3×10^6	2.5	9.8	124.0

Table 6.3: Descriptive statistics for lifetime in seconds.

A bucket is first created when a user experiences a crash and reports it. After that, other users might experience the same crash, and those crashes will be added to the bucket. Eventually, the code causing the crashes could be fixed, replaced, or become irrelevant. However, even if it is fixed, it is still possible that later changes undo the fix and more crashes are reported. Ideally, all of these crashes are in the same bucket.

It is important to understand how long crash buckets last in order to deduplicate them, simulate their creation over time, and manage crash repositories. For example when deciding on data retention policies, if we found that buckets have a very limited lifespan it would indicate that crashes could be removed after some time, limiting the amount of work that processes working with crashes, such as databases, would need to do.

We analyzed the lifespan of buckets in order to determine whether there are any patterns in bucket lifespan. The lifespan is the time between when the first and last crashes were posted to Launchpad, for each bucket. Of

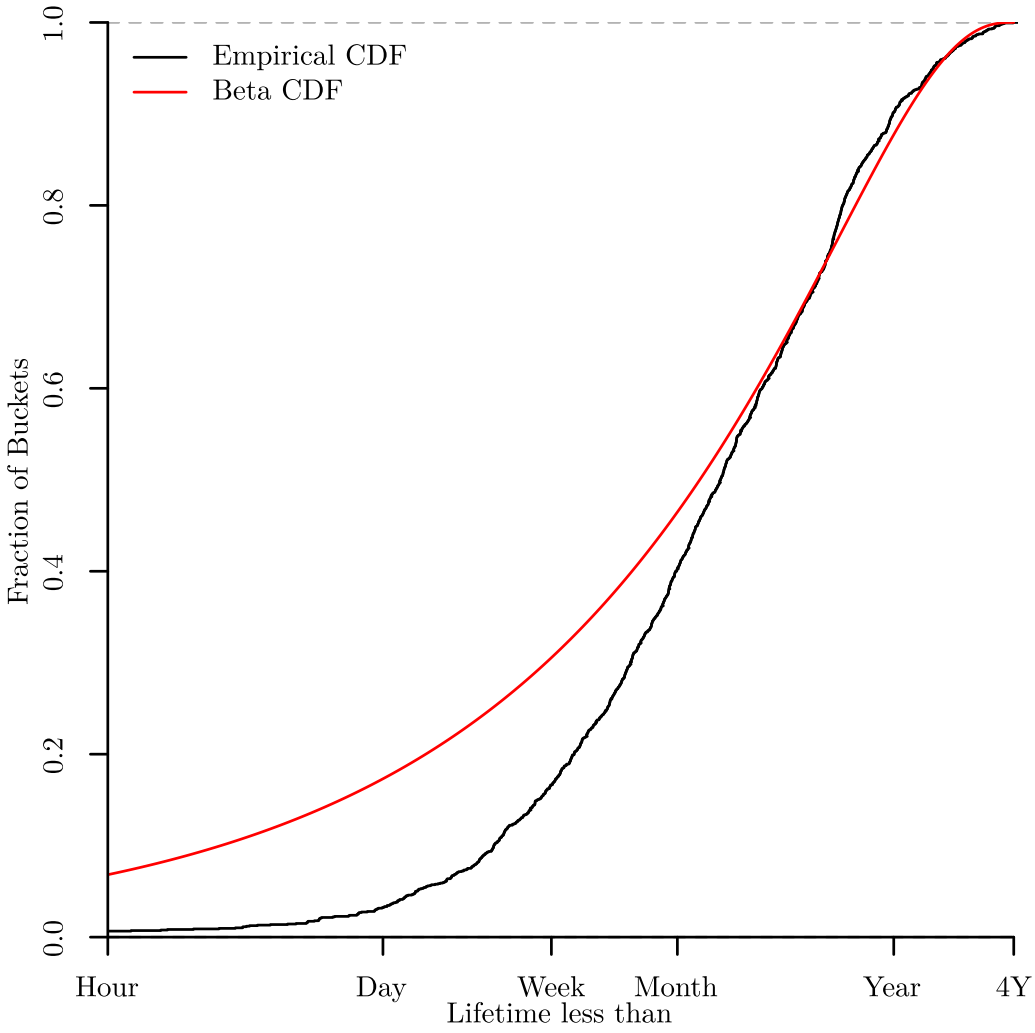

Figure 6.7: Empirical and model cumulative distribution functions of bucket lifespan. Only buckets with at least 2 crashes are shown.

course, there could have been crashes added to each bucket after the data was collected (and could be in the future) so it is impossible to know definitively how long each bucket lasts. We used a Cullen and Frey graph produced by the `fitdistrplus` R package (Delignette-Muller, Dutang, et al., 2015) to determine what distribution to attempt to fit to the data.

Figure 6.7 shows the lifespan of each bucket. The longest-lived bucket lasted for over 4 years, while there are several buckets containing multiple crashes that lasted less than an hour.

The Cullen and Frey graph indicated that the skewness and kurtosis of the data matched that of a beta distribution. An example beta distribution is show in Figure 6.7. The distribution was fit using the method of moments. When fitting a distribution and plotting, only buckets with two or more crashes were used. A method of moments fit matches the mean and the variance of theoretical distribution to the empirical distribution. The fit beta distribution shown in Figure 6.7 has parameters $a = 0.30$ and $b = 2.88$.

We used the Kolmogorov-Smirnov, Cramer-von Mises, and Anderson-Darling goodness-of-fit tests to determine how well the model fit the data. The tests all produced p-values less than 10^{-6}, indicating that the data did not come from a pure beta distribution of the estimated parameters. However, Figure 6.7 clearly shows that the beta distribution is able to capture the general shape of the empirical distribution. It is possible that the empirical distribution is a sum of two distributions or the result of multiple processes that a single beta distribution is not able to capture completely.

The beta distribution predicts that more buckets will have lifetimes less than a month than are actually observed. This could be the result of only considering buckets with two or more crashes in them.

There is little evidence to support a strong correlation between bucket size and bucket lifetime. The Pearson's (linear) correlation is 0.17, which is a weak correlation, but it is statistically significant, with a p-value of $2 \cdot 10^{-12}$. Similarly, the Spearman's (rank) correlation is 0.23, also statistically significant with a p-value of $2 \cdot 10^{-16}$. Despite the weakness of the correlation,

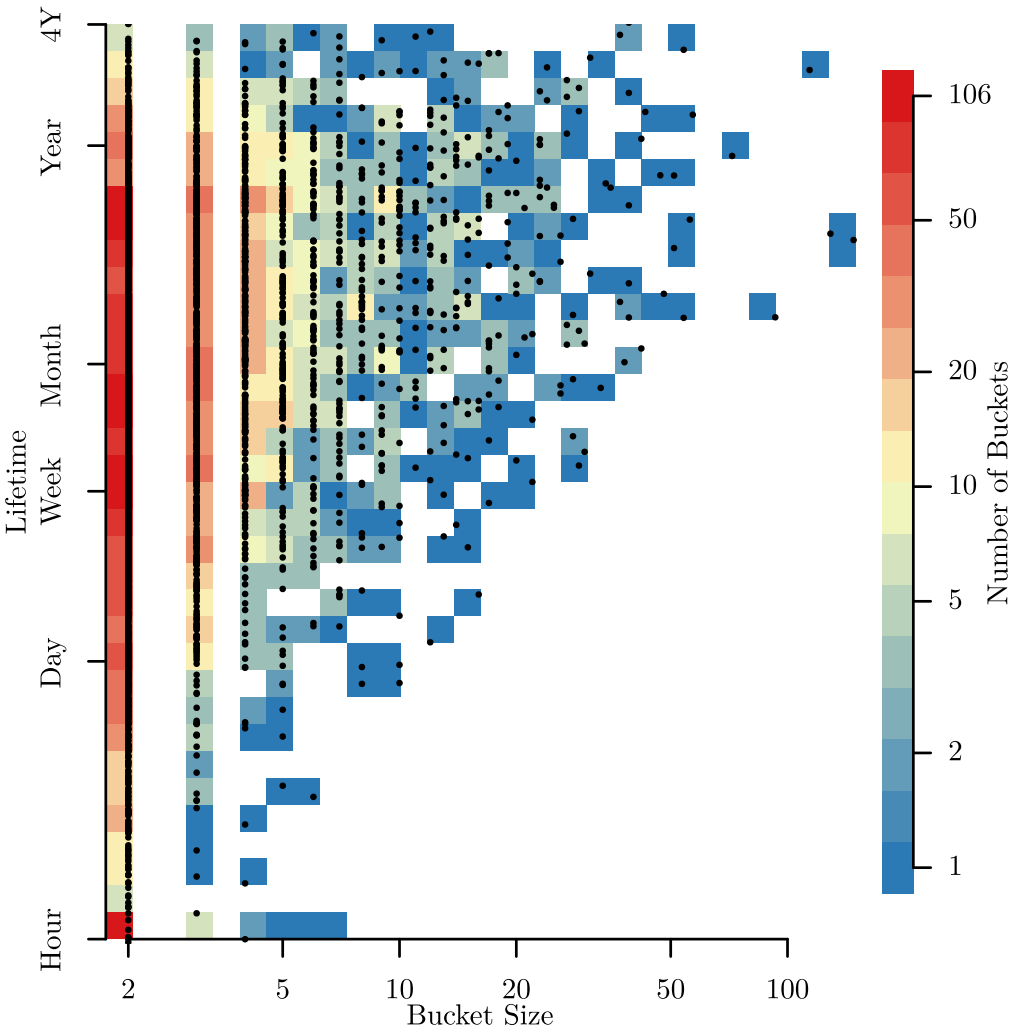

Figure 6.8: Lifespan of each bucket. Only buckets with at least two crashes are shown.

it is present. Our initial hypothesis was that larger buckets would have a longer lifespan, because with more crashes in a bucket, the bucket has more opportunities to have crashes that are spaced further apart in time. This relationship is shown in Figure 6.8.

However, we also expect that there is a competing effect: the more crashes a single bug causes, the more likely it is to be fixed by drawing the attention of Ubuntu developers or volunteers. The bucket with the largest lifespan, lasting from 2007-08-13 until 2011-09-12 (∼4 years, 30 days), had 39 crashes. However, the bucket with the second largest lifespan lasted from 2009-10-03 until 2013-10-16 (∼4 years, 12 days) and only contained two crashes. This is possibly due to a bug that was present in the code for a long time. In fact, the bug report for this crash is still open at the time of writing (2016-11-07).

The age of a crash may be artificially shortened due to the Launchpad maintainers deeming a crash report to be "unreproducible." Crashes (which are treated as a type of bug in the Launchpad system) are marked as "unreproducible" if the report does not contain enough information to replicate the conditions under which the crash occurred. A crash may be considered unreproducible even if it contains a stacktrace, context about the machine, and a comment from the user who experienced the crash. The issue will be closed with a status of "Invalid" (Zerone, 2009).

The empirical data shows that the lifespan of buckets is highly variable, and that it would be incorrect to assume that it is limited. If the data set contained 60 years of data rather than 6, crashes with lifetimes even longer than 4 years would probably be observed. Thus, crash repositories should keep data for as long as possible, because some old crashes will be experienced again. In addition, crash deduplication systems must be able to find duplicates of any age.

Duplicate crash reports can also arrive in fairly rapid succession. In this data set there are buckets with more than one crash, that lasts less than an hour. This indicates that crash deduplication systems must be able to form buckets quickly.

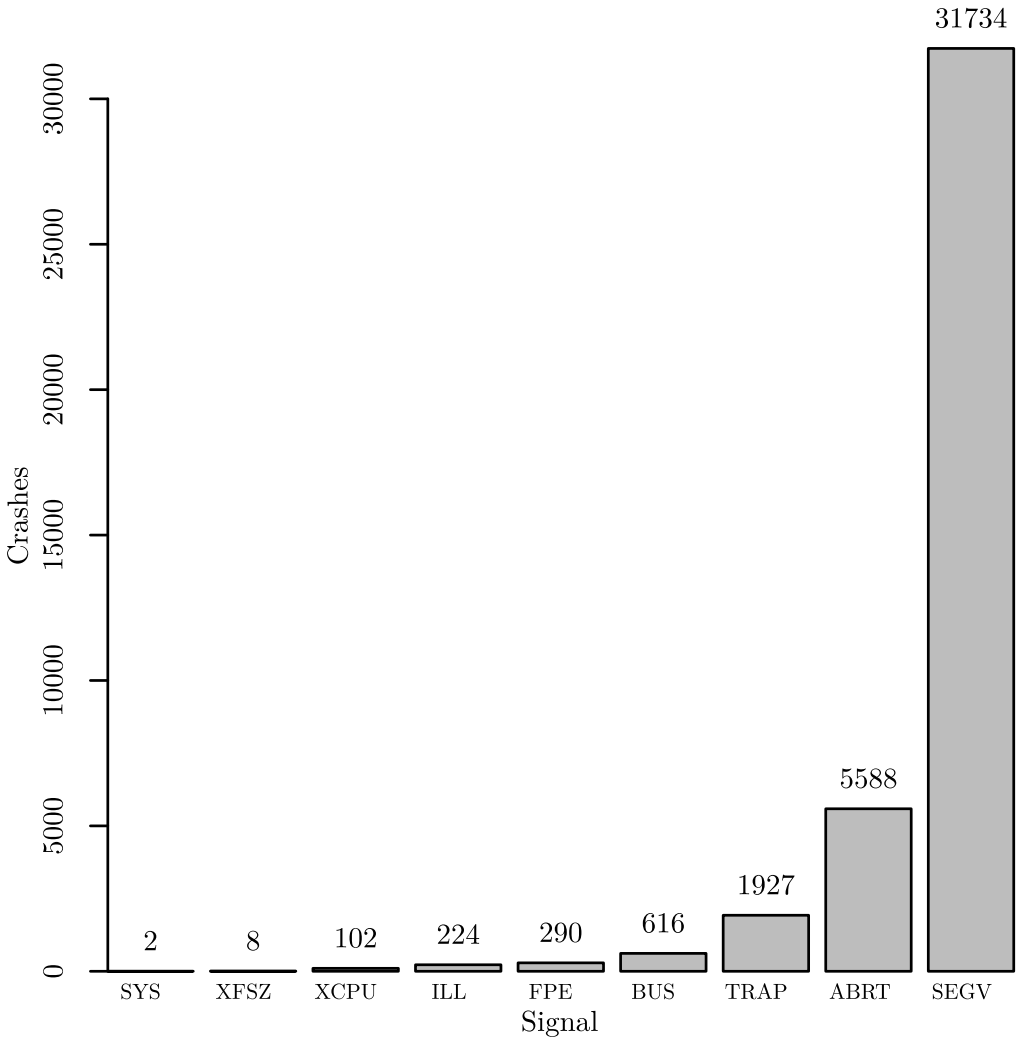

Figure 6.9: Signals causing crashes.

If the deduplication system only looked for similar crashes in an archive, and the archive was only updated once an hour, it might miss duplicate crashes.

6.6.4 RQ4. What Unix signals are crashes associated with?

Signals are sent to processes running on Linux when they crash. They can indicate what kind of crash the process experienced. Sometimes they are sent by the Linux kernel when it detects a crash, and sometimes they are sent by the program itself or by other programs on the system when they detect some

fault condition in order to crash the process. These are similar to exceptions in managed languages like Java, where there has been prior work (Kechagia and Spinellis, 2014) to understand that exceptions crash programs.

Because the signal indicates what general type of crash occurred, 2 crashes with different signals are unlikely to be caused by the same underlying bug. Therefore the signal is very useful for crash deduplication.

In order to investigate the ways in which applications crash, we used the crash metadata that usually contains a field indicating what signal a program crashed with. An example can be seen in Figure 6.1.

Figure 6.9 shows the amount of crashes associated with each signal in the data set. Due to the semi-structured text nature of the data, 101 crashes either did not have signals or the signals were not parsable.

The most common signal, `SEGV` (a *segmentation violation*), indicates that memory has been accessed in an illegal way. This can happen when a program tries to access memory that does not belong to it, has yet to be allocated, or has been deallocated. It can also happen if the program tries to write to read-only memory. `SEGV` can also occur when a program uses up all of the available stack space due to unbounded recursion (Section 6.6.7). It is sent to the program by the Linux kernel.

`ABRT` was the second most common signal and is typically caused by the program sending `ABRT` to itself. This can happen, for example, when a program attempts to deallocate memory that has already been deallocated. In this case, the fault is not detected by the OS kernel, but rather, by a library in use by the program.

The third most common signal causing crashes was `TRAP`. This signal is typically used to invoke a C debugger such as `gdb` (Eldredge, 2008). However, when a program receives `TRAP` and is not being debugged, the default action is to terminate the program. Thus it is used to either exit the program or pause the debugger when some fault condition is detected (Walters, 2011).

`BUS` was the fourth most common signal. Originally it was intended to indicate that there was something wrong with the hardware memory bus, but

this usage is not common anymore. This is because serious hardware faults are now handled by the kernel and typically crash the entire system. `BUS` signals do still occur in other situations, such as when a program tries to read data past the end of a file (Linux man-pages project, 2014).

The fifth most common signal, `FPE` indicates a floating-point math error, such as division by zero. It was followed by `XCPU` and `XFSZ`, both of which indicate that a program has used more than its allowed CPU time and that a program has made a file that is larger than the maximum allowed size, respectively. The least commonly seen signal was `SYS` that is described as unused.

From this information we can conclude that C and C++ software running on Ubuntu mostly crashes due to illegal memory access. This may indicate that there could be fewer crashes if memory-safe languages were used for these applications instead.

6.6.5 RQ5. What CPU architectures experienced the crashes?

When using a crash repository to evaluate deduplication methods, it is important to understand how the computer architecture will affect those findings. For example, Campbell et al. (2016a) found that the addresses of functions was a possible method of deduplicating crashes using an earlier version of the Launchpad data set used in this analysis. However, we can see that the data set contains two types of addresses: 32-bit and 64-bit. A 32-bit address will never match a 64-bit address.

We used the "Architecture" metadata field provided in most Ubuntu crash reports to examine how often crashes were reported on different CPU architectures. An example can be seen in Figure 6.1. Due to the semi-structured text nature of the data, 55 crashes either did not have architecture information or the architecture information was not parsable.

Figure 6.10 shows the amount of crashes occurring in each CPU architecture. The crashes in the data set consist primarily of the two PC architec-

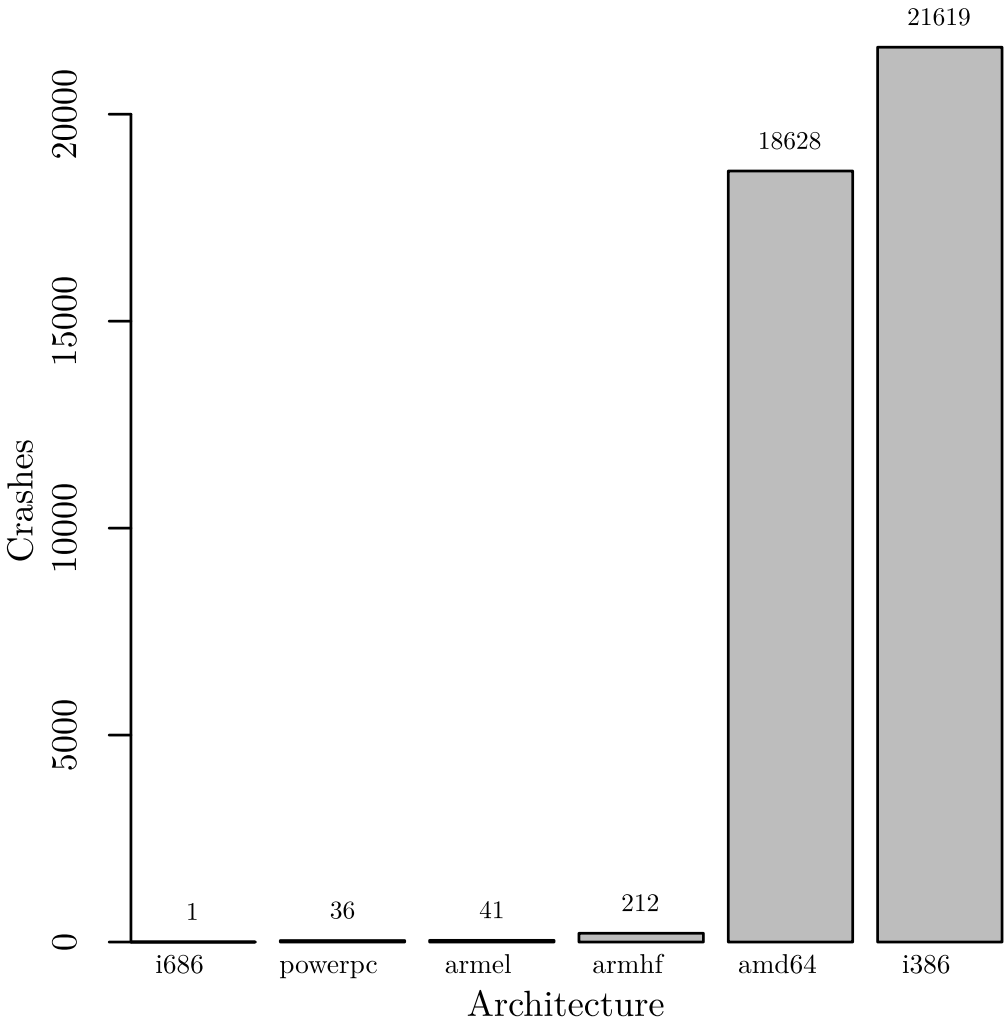

Figure 6.10: Crashes by architecture.

tures, `i386` and `amd64`. The most popular is the 32-bit `i386` architecture, even though it is becoming less common. The second most popular is the `amd64` architecture, which is the 64-bit version of the `i386` architecture. `amd64` is also commonly referred to as x86_64 in other places, and does not necessarily indicate that the CPU in use is an AMD.

Less popular architectures are `armhf` and the older `armel`. Both `armhf` and `armel` are ARM architectures, which differ on how they support floating-point math. The least popular architecture is `ppc`, indicating PowerPC CPUs. There is one crash labelled `i686`, but this is likely due to the person who submitted the crash mistakenly editing `i386` before they submitted it to Launchpad.

Because crash repositories can contain crashes from several hardware architectures, any system that works with crash reports—whether for storage, data mining, or deduplication—needs to function regardless of changes in metadata caused by differences between architectures. These difference include handling both 32-bit and 64-bit addresses.

6.6.6 RQ6. How long are crash stacktraces?

Minimum	Maximum	Median	Mean	Std. Dev.	Skewness	Kurtosis
1	2654	16	42.7	201.8	9.5	91.6

Table 6.4: Descriptive statistics for stack length.

Because stacktraces are a major component of a crash report, the longer a stacktrace is, the more data a crash report contains. That data may then be used as a source of features for a crash report deduplicator. For example, if all crashes only had one function in its stacktrace, it would be impossible to distinguish crashes from each other based on the stacktrace alone. A factor that may affect the length of stacktraces is *recursion* (Section 6.6.7).

Using our crash report extraction scripts (described in Section 6.5), we were able to count the amount of stack frames to determine the stacktrace length for each crash. Then we fit a geometric distribution to the empirical data.

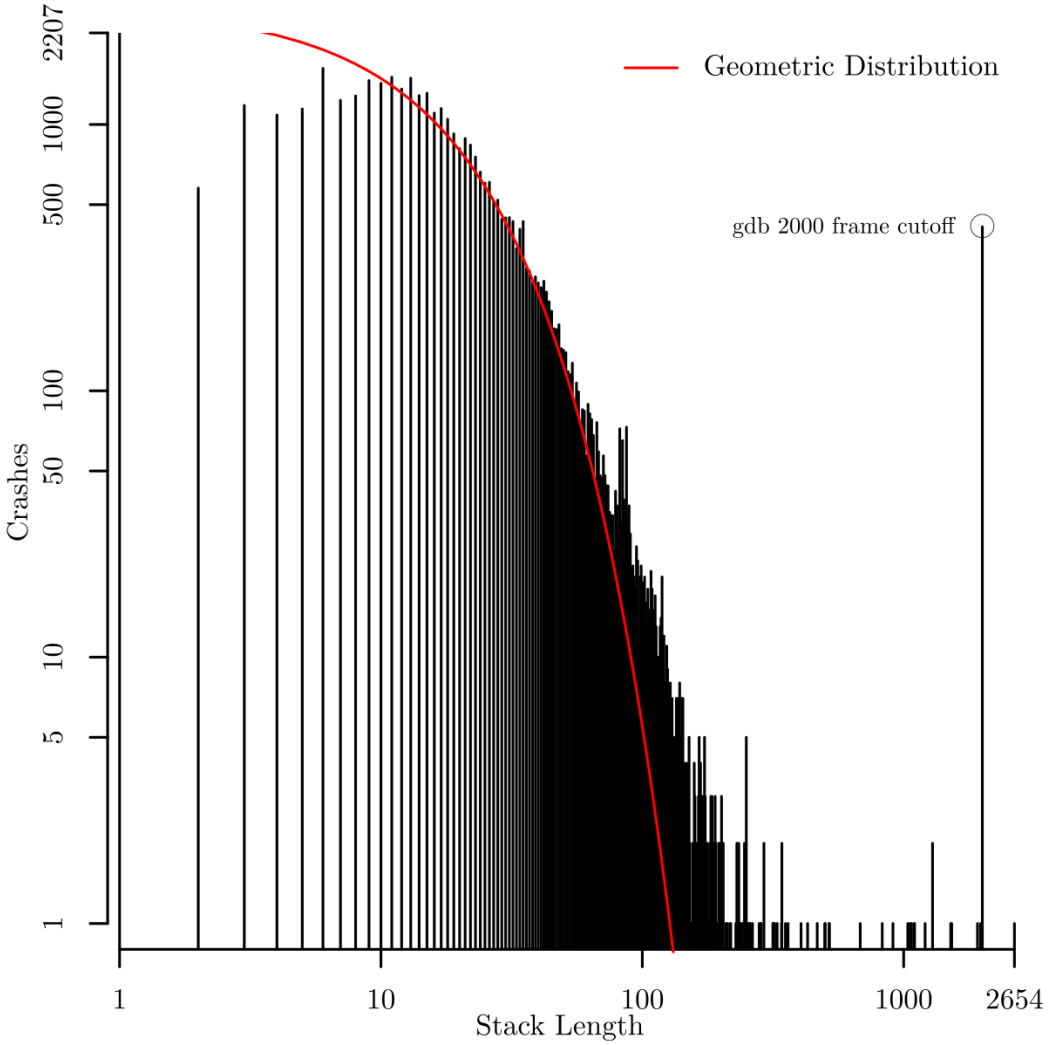

Figure 6.11: Stacktrace lengths.

Figure 6.11 shows a log-log plot of the lengths of stacks found in crashes. The peak at stack length 2000 is caused by a hard limit coded into `gdb` that prevents it from generating traces with more than 2000 frames. There were 413 such stacktraces truncated at 2000 frames. The code that performs this truncation is defined in `tracefile.c` in the `gdb` source with `#define MAX_TRACE_UPLOAD 2000` (Qi et al., 2014). The truncated stacktraces always include the top of the stack (the point of the crash), but discard the bottom of the stack (the entry point of the program). There was only one crash that exceeded the truncation limit with 2654 stack frames, which was an internal error from the Mono C# runtime (Ford, 2008).

In Figure 6.11, the red line indicates the model and the black bars indicate the empirical data. The model distribution is geometric distributed with parameter $p = 0.060$. We used the χ^2 goodness-of-fit test to determine how well the model fit the data. The test determined a p-value of less than 10^{-13}, which indicates that the data did not come from a geometric distribution with the parameters listed. However, it appears to capture the shape of the data with stack lengths between 10 and 50 quite well.

The fact that some of the data seems to follow a distribution that appears similar in shape to a geometric distribution, indicates that there may be some fixed chance of any function calling (or not calling) another function, and making the stack one frame longer. However, the small p-value, and the tail that extends past the fit geometric distribution indicates that this is only a part of the process that determines stack length.

The truncation behaviour in `gdb` is problematic for a number of reasons. First, this discarded information may be crucial for debugging, because the point that triggered the crash may be missing—even though it is essential for debugging the root cause of the crash. Second, it is incomplete information that may produce incorrect results in a naïve tool, relying on the fact that the bottom of the stack may contain the entry point of the program.

Future crash reporting systems should not truncate the stack if possible, to avoid losing information about what the software was doing before it began to

crash. In addition, crash deduplication systems need to be able to determine if stacks are similar regardless of stack length.

6.6.7 RQ7. How are crashes associated with recursion?

When synthesizing crash reports, it is important to know whether to model recursion, both bounded and unbounded; when creating crash report deduplication algorithms, it is useful to know whether unbounded recursion can be used as a feature to cluster crashes together.

We found all cases of single-function recursion or *trivial recursion*. To find cases of trivial recursion, we considered all stacktraces wherein at least 2 consecutive stack frames contained the same function name. Then, we counted the number of consecutive stack frames with the same function to find the recursion length. Note that each stacktrace with recursion may contain more than one instance of recursion; thus, we counted the instances of recursion contained within one stacktrace. We considered any trivial recursion as being *unbounded* if the recursive function was at the top-of-stack and the stack length exceeded `gdb`'s hard-coded limit, described in Section 6.6.6.

Minimum	Maximum	Median	Mean	Std. Dev.	Skewness	Kurtosis
2	2000	2.0	4.8	72.3	26.9	732.1

Table 6.5: Descriptive statistics for recursion lengths.

Summary descriptive statistics for recursion lengths are provided in Table 6.5. We plotted the length of all instances of trivial recursion in Figure 6.12, bottom. There appears to be at least two different distributions at play: one wherein there is shallow recursion, never exceeding 25 stack frames deep; and one wherein there is deep, perhaps unbounded recursion. There is an additional clump of recursion lengths around 1000 stack frames deep.

5,947 crash reports had trivial recursion associated. In those reports, 28,172 instances of recursion for 2 or more frames were identified. Thus, many crash reports had multiple instances of recursion within their stacktraces. Of the 413 truncated stacktraces (Section 6.6.6), 123 of these stacktraces exhibited

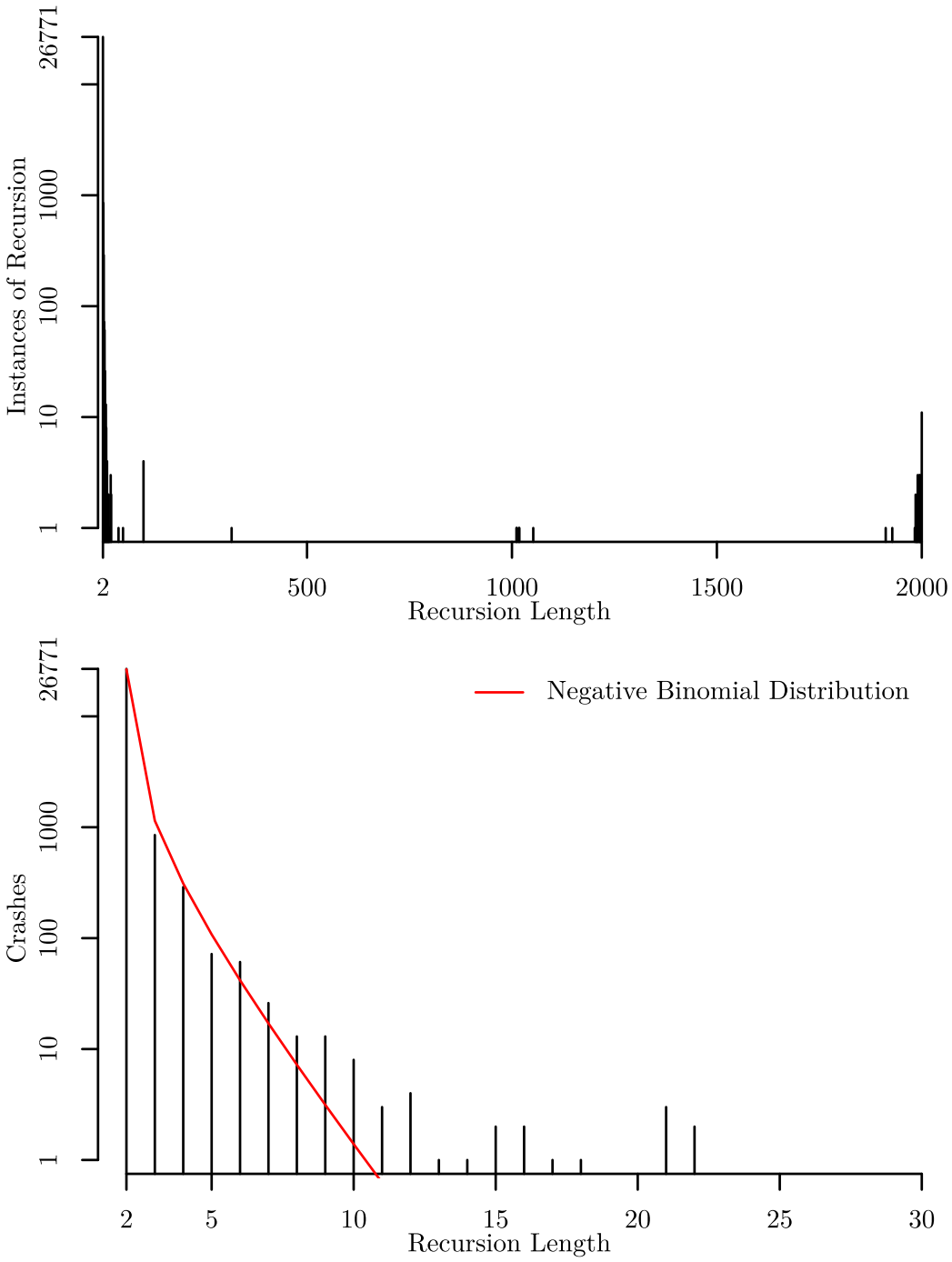

Figure 6.12: Recursion length. The top plot shows lengths 2 to 2000 and bottom right plot shows lengths 2 to 30 of the same data. A function is considered recursive if there are least two consecutive stack frames with the same function name, hence the minimum length of recursion is 2.

unbounded trivial recursion, leaving 290 truncated stacktraces unaccounted for.

We fit a negative binomial distribution to recursion lengths less than 20 stack frames. The fit distribution is shown in red in Figure 6.12. An offset of 2 was applied to the distribution, because recursion must have a length of at least 2. The μ (mean) parameter was 0.086 and the dispersion parameter (size) was 0.047. The fit distribution had a p-value of 0.0014 according to the χ^2 test, indicating that the data likely did not come from the fit distribution. However, based on Figure 6.12, the fit distribution captures the general shape of the empirical data for recursion lengths less than 8.

A crash in Rhythmbox exhibited unbounded trivial recursion (Zerone, 2009). All 2000 reported stack frames represented the same function: `rb_removable_media_source_should_paste_no_duplicate()`; thus, the entry point to this C program, `main()`, and all ancestors to the function call that caused the unbounded recursion were discarded from the stacktrace that was ultimately uploaded to Launchpad. This function was defined within a Rhythmbox plugin, Rhythmbox-Spotify-Plugin (McCann et al., 2008). The bug is interesting, because there is no explicit call to the function; rather, a function pointer is assigned to a member of a `struct` (Matthew et al., 2009), and a pointer to that `struct` is eventually passed to the crashing function itself. This function innocently calls the function pointer, implicitly calling itself (McCann et al., 2008).

For the 15 percent of crash reports that had trivial recursion, we can not necessarily say whether or not the recursion caused the crash, except in the case of the 123 stacktraces with trivial unbounded recursion. However, we can conclude that recursion occurred in a crash in a significant fraction of crash reports, and that it is capable of causing a crash. Synthetic crash report data must contain examples of both bounded recursion and unbounded recursion. Crash deduplication systems should be able to use information from both recursive functions and non-recursive functions.

6.6.8 RQ8. How long are function names in crashes?

Minimum	Maximum	Median	Mean	Std. Dev.	Skewness	Kurtosis
0	331	3	2.9	5.7	38.2	1987.6

Table 6.6: Descriptive statistics for function name lengths.

Function names are an important source of information when debugging, categorizing, or deduplicating crash reports because they indicate what the software was doing when it crashed. The longer (and hopefully more descriptive) a function name is, the more information it contains. This is supported by prior work on the quality and efficacy of identifier names (Lawrie et al., 2007a; Lawrie et al., 2007b). However, in this study a single "function name" may include multiple identifiers, such as its declaring class, and any C++ template parameters. In order to extract that information, function names must be tokenized.

In order to evaluate function name lengths in words, we used the CamelCase tokenizer used in Campbell et al. (2016a) for crash report deduplication. We re-implemented the tokenizer in Python (originally, it was implemented in Java) and then applied it to every function name in the data set. The CamelCase tokenizer splits words into tokens on symbols and at lowercase-to-uppercase transitions. It is a regular expression suggested by the ElasticSearch documentation (Gormley, 2015).

Sigurd et al. (2004) fit gamma distributions to word lengths in syllables and sentence lengths in words in various natural languages. Because words in a function name serve a similar purpose to words in a sentence, and often contain natural language vocabulary words (Lawrie et al., 2007a; Lawrie et al., 2007b), we attempted to fit a gamma distribution to the tokenized function name lengths as well.

However, as shown in Figure 6.13, the Gamma distribution does not appear to be a particularly good fit for function name lengths in tokens. When only unique tokens are considered, the tail of the distribution extends past where it is predicted by the gamma distribution. The gamma distribution fit for every

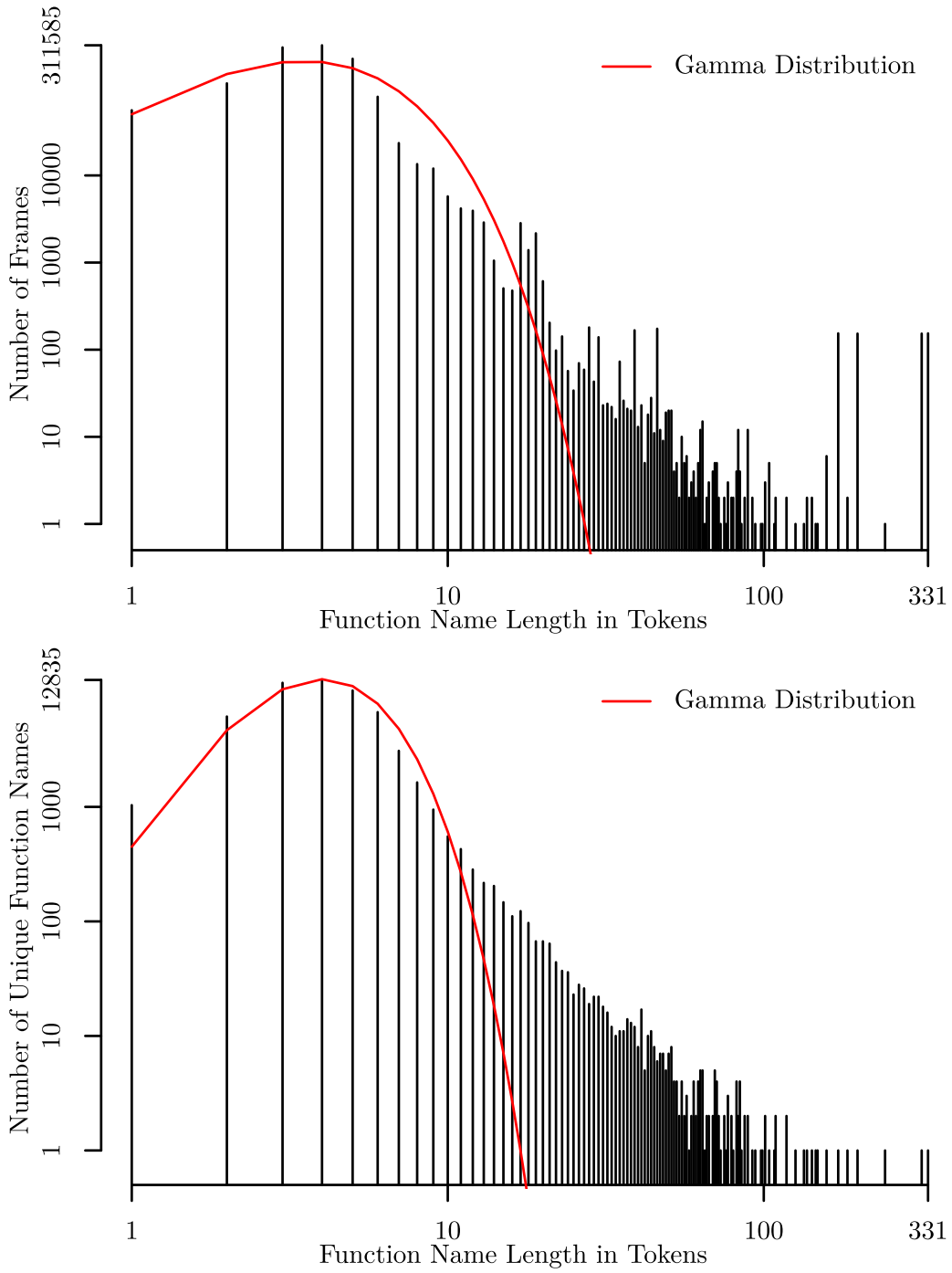

Figure 6.13: Function name lengths in tokens as produced by the CamelCase tokenizer.

#	Count	Function name
1	15477	`main`
2	14625	`g_main_context_dispatch`
3	12433	`g_main_loop_run`
4	8501	`__libc_start_main`
5	7654	`g_signal_emit_valist`
6	7526	`g_closure_invoke`
7	7204	`g_signal_emit`
8	7197	`g_main_context_iterate`
9	5836	`gtk_main`
10	5662	`g_main_dispatch`

Table 6.7: The top 10 most common function names in crash stacktraces. Each function name is counted at most once per crash.

function name seen in every frame had parameters $\alpha = 3.6$ and $\beta = 0.74$. The gamma distribution fit for unique function names had parameters $\alpha = 6.3$ and $\beta = 1.3$. p-values were computed using the Kolmogorov-Smirnov, Cramer-von Mises, and Anderson Darling tests for both distributions, but they were all less than 10^{-16}, indicating the data did not come from the fit distributions. The fit distributions only appear to capture the shape of the data for function names less than around 10 tokens.

Occasionally, function names can become very long when using C++ templates as a metaprogramming device. One example contains function names with over 300 tokens in over 2000 characters using the CamelCase tokenizer (Kehne, 2013).

From this data we can conclude that function names can range from very short, single-token names to very long, multiple identifier names involving C++ templates. Thus, crash repositories and crash deduplication systems must be able to handle function names that range from short to very long. Synthetic crash repositories should contain function names of a wide variety of lengths.

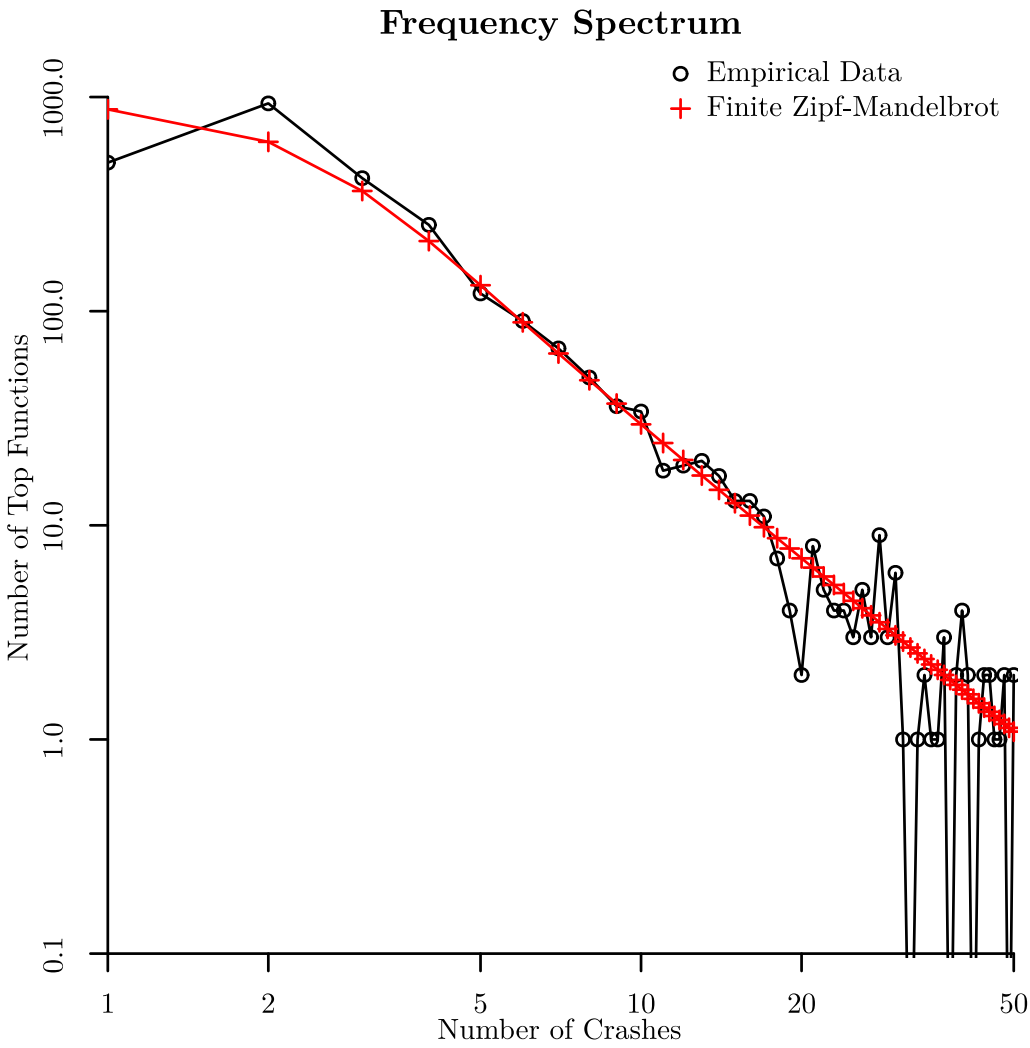

Figure 6.14: Frequency spectrum of crashes by the function on top of the stacktrace, indicating there are many functions that only appear on top of the stacktrace once, and a few functions that appear on top of the stacktrace in many crash reports. Both axes are logarithmic.

#	Count	Function name	Most common signal		
			Name	Count	%
1	3390	`__kernel_vsyscall`	ABRT	2637	77.8%
2	1442	`__GI_raise`	ABRT	1421	98.5%
3	1440	`g_logv`	TRAP	1428	99.2%
4	1237	`raise`	ABRT	946	76.5%
5	544	`g_type_check_instance_cast`	SEGV	543	99.8%
6	299	`strlen`	SEGV	286	95.7%
7	245	`free`	SEGV	244	99.6%
8	230	`pthread_mutex_lock`	SEGV	230	100.0%
9	220	`malloc_consolidate`	SEGV	219	99.5%
10	205	`g_slice_alloc`	SEGV	205	100.0%

Table 6.8: The top 10 most common function names at the top of crash stacktraces. The most common signal associated with the function is given. An explanation of what each signal means is provided in Section 6.6.4.

6.6.9 RQ9. What are the most common crashing functions?

An important piece of information about the cause of a crash is what the program is doing at the time of the crash. The functions present in the stack at the time of the crash indicate what the software was doing when it crashed. This is usually indicated by the function on the top of the stack at the time of a crash. It is also considered to be the most important part of the stack in many papers on crash deduplication (Glerum et al., 2009; Modani et al., 2007; Wang et al., 2013). Thus, by examining the functions on top of the stack, we can determine how reliable the top of the stack is for determining the cause of a crash.

To collect function names, we iterated through every crash in the data set, and collected every *unique* function name present in the crash stacktrace. Not every stack frame had a function name. Function names are often missing due to missing debugging symbols that `gdb` relies on to produce this information. Once collected, we counted the number of crashes each function name appeared in.

Only 36,511 out of the 40,592 crashes (89.9%) had any parsable function names at all. Of these, 27,406 crashes had parsable function names on the top of the stack (67.5% of all crashes). Thus, the results presented here are a subset of the total corpus. The results of the most common unique function names within a stacktrace are presented in Table 6.7, and the most common functions at the top of the stack, along with their crashing signal, are presented in Table 6.8.

Figure 6.14 shows the frequency spectrum of the top functions. We fit a finite Zipf-Mandelbrot model to the frequency spectrum using the `zipfR` R package (Evert and Baroni, 2007). In Figure 6.14, the red line indicates the model distribution and the black line indicates the empirical data. We used the χ^2 goodness-of-fit test to determine how well the model fit the data. The test determined a p-value of less than 10^{-75}. However, the shape of the model distribution seems to follow the empirical distribution quite closely, with the exception of functions that only appear in a single crash (the leftmost points plotted in Figure 6.14). This could indicate some other process is affecting the number of functions that are only seen once.

Despite being the entry point for most C and C++ programs (thus, most likely to be the top function present in stacktraces), `main()` is present in only 15,477 crashes (42.4% of crashes with function names). Part of this may be explained by the stack truncation reported in Section 6.6.7.

Because the most common function on the top of the stack merely indicates that some signal was raised, but not the name of the signal raised, it alone is often a poor indicator of what the software is truly doing at the time of the crash. This is why we included its most common crashing signal. In particular, the function `__kernel_vsyscall` is a generic way to invoke any system call in Linux, and as such, its most common signal, `ABRT`, does not account for 22.2% of all its crashes. The second, third, and fourth most common functions on top of the stack (`__GI_raise`, `g_logv`, and `raise`, respectively) merely indicate that the program is crashing itself intentionally after detecting some fault condition. This is supported by ABRT (intentionally aborts the program)

and TRAP (causes the debugger to break here, or crash if not attached to a debugger) being the most common signals for these crashes. The rest of the top ten are caused by the top signal overall `SEGV`, or segmentation violations (Section 6.9). These are often caused due to passing an invalid pointer or a `NULL` pointer to some function argument, causing the function at the top of the stack to access memory in some invalid manner. For example, the fifth most common crasher (`g_type_check_instance_cast`) indicates that an invalid pointer was passed into a glib function. The sixth most common function on top of the stack does not give a clear indication of the bug: either a bad pointer was passed to `strlen`—a C function that computes the length of a string—or the string was not null-terminated, perhaps due to being invalid or corrupted data.

The top of a stacktrace is often not a good indication of what exactly the fault is. Thus, crash deduplicators must not rely on the top of the stacktrace. Instead, they should examine the entire stack for clues, and contextual data like the crashing signal for clues.

The top of the stack is populated with intentional crashes, aborts, and syscalls, as well as calls to the standard library with inappropriate data. Again we see that improper memory management affects many crashes rather than resource depletion.

6.6.10 RQ10. What are the most common crashing libraries?

While crashes are often specific to a particular product, similar faults or misuses of a library can occur across the clients of a library. Even worse, if a library has a fault it could induce crashes across numerous client products. Thus we ask the question what are the most common crashing libraries. Are they just misuses of libc or is GUI client code causing instability? Are device drivers at play or concurrency? In this section we look for evidence of libraries whose use can cause or correlate with instability across many products.

#	Count	Library name
1	2950	`libc`
2	1854	`libglib-2`
3	469	`libpthread`
4	376	`libgobject-2`
5	341	`libgtk-x11-2`
6	309	`libGL`
7	268	`ld-linux`
8	152	`libQtCore`
9	143	`libdbus-1`
10	141	`libQtGui`

Table 6.9: The top 10 most common library names at the top of crash stacktraces.

In order to determine the libraries that crashed the most often, we examined the top of the stacktrace. For some crash reports, frames have the library file listed in the crash report. For others, the source file is listed, or there is no information about the origin of the function. We only considered stacktraces that had the library file specified, because it could be parsed the most reliably. 11,569 out of 40,592 crashes had a library file specified for the function at the top of the stack.

The 10 most common crashing libraries are listed in Table 6.9. The most common library in which crashes occurred was `libc`, the C standard library. Yet libraries about concurrency, `libpthread`, GUI libraries, `libgtk-x11-2`, `libQtGui`, `libQtCore`, and even 3D graphics, `libGL`, are ranked top crashers.

Using our methodology we could only determine what library functions software crashed inside. However, a library function may crash because it was supplied invalid arguments by another part of the software. Therefore, we cannot necessarily conclude that the libraries that crash the most often are the most buggy. For example, the library that crashed the most often, `libc` is probably the most well-tested C library because it is used by every C program on Linux. Thus, it is not likely that the crashes in `libc` are caused by faults in `libc`, but rather by software using `libc`. `libGL` also signals that graphical device drivers could also be at fault for instability.

The libraries and frameworks that tend to induce crashes are both popular and infrastructural. Many crashes seem to be the result of passing invalid parameters (invalid pointers) to library and framework code. Ubuntu's UI are dominated by GTK and QT based programs so it is not surprising to see these functions appear in stacktraces. We are left wondering if managed code environments such as C# or Java exhibit the same properties.

6.7 Threats to validity

6.7.1 Threats to construct validity

Construct validity is threatened by the fact that parsing the semi-structured textual crash reports is not completely reliable. As stated, 3,872 reports (9 percent) of the collected reports were thrown out because they could not be parsed. Additionally, the crashes that could be parsed were not always perfectly reliable. For example, a missing newline character can cause data from one field to appear in another field. However, we believe these issues to be very rare, affecting only a handful of crashes.

6.7.2 Threats to internal validity

Internal validity is threatened by the validity of the Ubuntu Launchpad data set itself. We trusted that the manually tagged duplicates were true duplicates and that all true duplicates were found and clustered appropriately. This affects our conclusions about the distribution of crashes among buckets (Section 6.6.2) and the lifespan of crash buckets (Section 6.6.3).

Even though we only studied crashes for which we could parse stacktraces, not all stack frames contained crucial information such as function names, line numbers, and filenames. As such, this fact may affect our conclusions depending on function names (Section 6.6.7, Section 6.6.9, and Section 6.6.8).

A threat to internal validity is alluded to in Figure 6.2 is that end-users have the ability to *opt-out* of submitting a crash report. As such, this biases the data that is ultimately available for analysis—both in this chapter, and for

other purposes, such as crash report deduplication. In addition, the crashes that *do* appear in Launchpad require that the users have a Launchpad account, as the bug reports are intended to be uploaded by more technically-minded users. This further biases the data we have available for study. This affects our conclusions, as we cannot reason about crash reports that were never uploaded.

6.7.3 Threats to external validity

The crashes from the Launchpad data set came exclusively from Ubuntu-based distributions of Linux, focusing on personal computing and server applications (such as Rhythmbox, a music player, and Nautilus, a graphical file manager). As such, our conclusions may not be applicable to other platforms and operating systems such as macOS and Windows, different crash report repositories, or even different distributions of Linux.

Since the stacktraces we collected exclusively come from software debuggable by `gdb`, most of the crashes come from C and C++ software. The results here may not generalize to software written in other programming languages such as Java or Python.

Because the Launchpad data set consists of crashes from many different applications, the results in this chapter may not generalize to crash repositories that contain only crashes from a single application such as Mozilla's (Mozilla Corporation, 2012).

6.8 Conclusions

We analyzed 40,592 crash reports for 1,921 packages from the Ubuntu Launchpad crash repository. From this data set, we found that various properties of crash reports tend to have empirical distributions that, at least partially, resemble standard statistical distributions. However, crash reports are complicated, and no single standard distribution fit the entire range of crash reports we observed.

When grouped by package or by bucket, crashes exhibited power-law distributions. There are a few groups with many crashes, and many groups with few crashes. This fact should be taken into account when deciding that two crashes are similar, and when determining which crashes should receive developer attention. Crash deduplication systems must be able to handle this wide variety. The data set also shows that it is not possible to rely on buckets having a limited life span. Thus, crash repositories and deduplication techniques must allow crashes to be added to buckets regardless of their age.

Crashes in this data set, which consists of programs written in naively compiled languages, such as C and C++, are heavily impacted by the hardware architecture. Crashes from two different architectures may be difficult to compare despite being caused by the same underlying software fault. In addition, the most common fault type for the crashes in the data set was illegal memory access (`SEGV`). This indicates that preventing illegal memory access may be a valuable approach to preventing crashes.

Crash stacktraces and function names can range from short to very long. Crash repositories and crash reporting tools should be able to handle any length of stacktrace. Crash deduplication techniques must be able to handle any length of stacktrace and a wide range of lengths of function names, despite the fact that long stacktraces and long function names provide much more information that deduplication techniques can use than short function names and short stacktraces. Additionally, deduplication techniques should handle crashes with no recursion, bounded recursion, and unbounded recursion.

Popular infrastructural, GUI, and standard libraries and functions appeared in stacktraces. Graphical libraries such as `libGL` and concurrency libraries such as `libpthread` occurred as well. Essentially complicated and popular infrastructure libraries and functions appeared frequently in crashes. While these could be common errors that clients are susceptible to, it might be worthwhile for library maintainers to understand how their code becomes involved in these crashes. Perhaps library maintainers could provide checks for the more common crashes?

Overall, this data indicates that there are many crashes that have similar characteristics, and many crashes that have unique and exceptional characteristics. Crash repositories, deduplicators, or other systems cannot rely on crashes following common patterns or expressing limited ranges, because every property has significant outliers. For example, crash repositories cannot rely on buckets having a consistent size, applications causing similar numbers of crashes, or buckets having limited lifespans. Crash deduplicators cannot rely on crashes having functions with addresses or names of similar length, or stacktraces of similar lengths, and they should examine the entire stack, not just the first function. Recursion can cause problems by making the stack too long or by hiding the bottom of the stack if it gets truncated.

Future work includes analyzing other crash repositories to determine if crashes from those repositories follows similar distributions to the crashes in the Ubuntu Launchpad data set. In addition, studying crashes in languages such as Python, which crash due to exceptions rather than signals, and are unlikely to experience a `SEGV` signal, might give better insight into why software crashes.

The lack of larger data sets, and data sets with known good examples of unique (not duplicate) crash reports motivates the need to create data sets where such knowledge exists. A major reason to understand the distributions of crash report data is for creating synthetic crash report data sets. Unfortunately, synthesizing crash report data sets is still work for the future. For synthetic data sets to be helpful, we require a deep understanding of what real crash repositories contain. This must include understanding properties of real crashes and their statistical distributions within real crash repositories. Because of the scarcity of data sets that contain gold sets of human-deduplicated crashes that can be used to study crash report deduplication techniques, we feel it would be beneficial to produce large data sets of artificial automatically synthesized crash reports.

Chapter 7

Clustering Automatically Generated Software Crash Reports

7.1 Preface

7.1.1 Acknowledgement

This chapter originally appeared as a report in the Proceedings of the 13th Working Conference on Mining Software Repositories, 2016, as "The Unreasonable Effectiveness of Traditional Information Retrieval in Crash Report clustering."

I, Hazel Victoria Campbell, wrote most of the chapter, conducted most of the experiments, collected the data used, produced results and analysis and created the figures. Eddie Antonio Santos assisted with developing the software described in the chapter and used in the experiments. Dr. Abram Hindle wrote did some writing and editorial work on the chapter and supervised the research. The research presented in this chapter was funded by a MITACS Accelerate grant and BioWare, LLC.

7.1.2 Significance

The work presented in this chapter was the first time that textbook Information Retrieval (IR) techniques had been applied to the problem of crash-report bucketing (clustering). Much like UnnaturalCode, the technique presented in

this chapter is a significant departure from traditional bucketing techniques. By using a "bag-of-words" approach with TF-IDF representation, and aggressive tokenization, much of the information that is relied upon by other bucketing techniques is lost. However, the ability to handle information flexibly, such as "metadata" that may not be present in every crash, automatically, is introduced by this work.

The importance of practicality is a major focus of the work presented in this chapter, an aspect of crash report bucketing that is often neglected in other techniques. Instead, much of the literature on crash report bucketing focuses solely on clustering accuracy. Often, bucketing accuracy in related works is gained at the expense of speed. In this chapter I argue why techniques using algorithms with quadratic time complexity (or worse) are simply not practical in industrial applications.

7.1.3 Impact

From its publication in 2016 until mid-2020, this chapter has earned a number of citations, including a citation in the top-tier journal Empirical Software Engineering, and the top-tier conference, Foundations of Software Engineering.

7.1.4 Extensions

Perhaps the most obvious improvement on the algorithm presented in this chapter is to use more of the information that PartyCrasher discards, such as function call order. Moroo et al. (2017) presents a re-ranking based technique to allow function call order to inform the organization of the final buckets (clusters of crash reports). Moroo et al. (2017) not only extends the work presented in this chapter with a new technique based upon it, but provides another comparison of bucketing "performance," in both speed and accuracy, the author's technique, and another piece of prior work, "ReBucket," described in Dang et al. (2012b).

7.2 Introduction

Ada is a senior software engineer at Lovelace Inc., a large software development company. Lovelace has just shipped the latest version of their software to hundreds of thousands of users. A short while later, as Ada is transitioning her team to other projects, she gets a call from the quality-assurance team (QA) saying that the software she just shipped has a crashing bug affecting two-thirds of all users. Worse yet, Ada and her team can not replicate the crash. What would really be helpful is if every time that crash was encountered by a user, Lovelace would automatically receive a *crash report* (Seo and Kim, 2012a), with some *context* information about what machine encountered the crash, and a *stacktrace* (Seo and Kim, 2012a) from each thread. Developers consider stacktraces to be an indispensable tool for debugging crashed programs—a crash report with even one stacktrace will help fix the bug significantly faster than if there were had no stacktraces available at all (Schröter et al., 2010).

Luckily for Ada, Lovelace Inc. has gone through the monumental effort of setting up an automated crash reporting system, much like Mozilla's Crash Error Reports (Mozilla Corporation, 2012), Microsoft's WER (Glerum et al., 2009), or Apple's Crash Reporter (*Technical Note TN2123: CrashReporter* 2016). Despite the cost associated with setting up such a system, Ada and her team find the reports it provides are invaluable for collecting telemetric crash data (Ahmed et al., 2014).

Unfortunately, for an organization as large as Lovelace Inc., with so many users, even a few small bugs can result in an unfathomable amount of crash reports. As an example, in the first week of 2016 alone, Mozilla received 2 189 786 crash reports, or about 217 crashes every minute on average.[1] How many of crash reports are actually relevant to the bug Ada is trying to fix?

[1] `https://crash-stats.mozilla.com/api/SuperSearch/?date=>%3d2016-01-01&date=<%3d2016-01-08` The total number of crashes will slowly increase over time and then eventually drop to zero due to Mozilla's data collection and retention policies.

The sheer amount of crash reports present in Lovelace's crash reporting system is simply too much for one developer, or even a team of developers, to deal with by hand. Even if Ada spent only one second evaluating a single crash report, she would still only be able to address 1/3 of Lovelace's crash reports received during one day of work. Obviously, an automated system is needed to associate related crash reports together, relevant to this one bug, neatly in one place. All Ada would have to do is to select a few stacktraces from this *crash bucket* (Glerum et al., 2009), and get on with debugging her application. Since this hypothetical bucket has all crash stacktraces caused by the same bug, Ada could analyze any number of stacktraces and pinpoint exactly where the fault is and how to fix it.

The questions that this chapter seeks to answer are:

RQ1: What are effective, industrial-scale methods of crash report bucketing?

RQ2: How can these methods be tuned to increase precision or recall?

This chapter will evaluate existing techniques relevant to crash report bucketing, and propose a new technique that attempts to handle this fire hose of crash reports with industrially relevant upper bounds ($O(\log n)$ per report, where n is number of crash reports). In order to validate new techniques some of the many techniques described in the literature are evaluated and compared in this chapter. The results of the evaluation shows that techniques based on the standard information retrieval statistic, *term frequency* $\times$ *inverse document frequency* (TF-IDF), do better than others, despite the fact these techniques discard information about what is on the top of the stack and the order of the frames on the stack.

7.2.1 Contributions

This chapter presents PartyCrasher, a technique that buckets crash reports. It extends the work done by Lerch and Mezini (2013) to the field of crash report clusteringand show that despite its simplicity, it is quite effective. This chapter contributes:

1. a criterion for industrial-scale crash report clusteringtechniques;

2. replication of some existing methods of clustering(such as Wang et al. (2013) and Lerch and Mezini (2013)) and evaluations of these methods on open source crash reports, providing evidence of how well each technique performs at crash report bucketing;

3. implementation of these methods in an open source crash bucketing framework;

4. evaluation based on the automated crashes collected by the Ubuntu project's Apport tool, the only such evaluation at the time of writing;

5. a bug report clusteringmethod that outperforms other methods when contextual information is included along with the stacktrace.

7.2.2 What makes a crash bucketing technique useful for industrial scale crash reports?

The volume, velocity, variety, and veracity (uncertainty) of crash reports makes crash report bucketing a big-data problem. Any solution needs to address concerns of big-data systems especially if it is to provide developers and stakeholders with *value* (Marr, 2015). Algorithms that run in $O(n^2)$ are unfeasible for the increasingly large amount of crash reports that need to be bucketed. Therefore, an absolute upper-bound of $O(n \log n)$ is chosen for evaluated algorithms.

The methods evaluated in this chapter were methods found in the literature, or methods that the authors felt possibly had promise. Methods that were evaluated in this chapter were restricted to those that met the following criteria. The criteria were chosen to match the industrial scenario as described in the introduction.

1. Each method must scale to industrial-scale crash report clusteringrequirements. Therefore, it must run in $O(n \log n)$ total time. Equivalently, each new, incoming crash must be able to be assigned a bucket in $O(\log n)$ time or better.

2. No method may delay the bucketing of an incoming crash report significantly, so that up-to-date near-real-time crash reports, summaries, and statistics are available to developers at all times. This requires the method to be *online*.

3. No method may require developer intervention once it is in operation, or require developers to manually categorize crashes into buckets. This requires the method to be *unsupervised*.

4. No method may require knowledge of the eventual total number of buckets or any of their properties beforehand. Each method must be able to increase the number of buckets only when crashes associated with new faults arrive due to changes in the software system for which crash reports are being collected. This requires the method to be *non-stationary*.

Several clusteringmethods are evaluated in this chapter. They can be categorized into two major categories. First, several methods based on selecting pre-defined parts of a stack to generate a *signature* were evaluated. The simplest of these methods is the `1Frame` method, that selects the name of the function on top of the stack as a signature. All crashes that have identical signatures are then assigned to a single bucket, identified by the signature used to create it.

Similarly, signature methods `2Frame` and `3Frame` concatenate the names of the two or three functions on top of the stack to produce a signature. `1Addr` selects the address of the function on top of the stack to generate a signature rather than the function name. `1File` selects the name of the source file in which the function on top of the stack is defined to generate a signature, and `1Mod` selects either the name of the file or the name of the library, depending on which is available. Figure 7.1 shows an example stacktrace and how the various signatures are extracted from it using these methods. All of the signature-based methods, as implemented, run in $O(n \log n)$ total time or $O(\log n)$ amortized time.

```
#1  0x00002b344498a150 in CairoOutputDev::setDefaultCTM () from
/usr/lib/libpoppler-glib.so.1
#2  0x00002b344ae2cefc in TextSelectionPainter::TextSelectionPainter () from
/usr/lib/libpoppler.so.1
#3  0x00002b344ae2cff0 in TextPage::drawSelection () from
/usr/lib/libpoppler.so.1
#4  0x00002b344498684a in poppler_page_render_selection () from
/usr/lib/libpoppler-glib.so.1
```

Method	Signature
1Frame	`CairoOutputDevsetDefaultCTM`
2Frame	`CairoOutputDev::setDefaultCTM` `TextSelectionPainter::TextSelectionPainter`

3Frame	`CairoOutputDev::setDefaultCTM`
	`TextSelectionPainter::TextSelectionPainter`
	`TextPage::drawSelection`

1Addr	`0x00002b344498a150`
1File	No Signature (no source file name given in the stack)
1Mod	`/usr/lib/libpoppler-glib.so.1`

Method	Tokenization

None	`#1 0x00002b344498a150 in`
	`CairoOutputDev::setDefaultCTM () from`
	`/usr/lib/libpoppler-glib.so.1`

Lerch	`0x00002b344498a150` `cairooutputdev` `setdefaultctm` `from` `libpoppler` `glib`

Space	`#1` `0x00002b344498a150` `in`
	`CairoOutputDev::setDefaultCTM` `()` `from`
	`/usr/lib/libpoppler-glib.so.1`

Camel	`1` `0` `x` `00002` `b` `344498` `a` `150` `in` `Cairo` `Output` `Dev` `set` `Default` `CTM` `from` `usr` `lib` `libpoppler` `glib` `so` `1`

Figure 7.1: An example stacktrace (top), its various signatures (middle), and various tokenizations of the top line of the stacktrace (bottom).

`1Addr` is not necessarily threatened by address space layout randomization (ASLR). ASLR is a technique employed by the operating system kernel and loader/dynamic linker to reduce the exploitability of security flaws in running software. However, the debugger which produces crash reports automatically undoes any randomization. Thus, stack traces in two different crash reports usually contain comparable addresses. If the debugger did not undo the randomization, `1Addr` would be completely unusable.

The second category of methods are those based on TF-IDF (Salton and McGill, 1983) and inverted indices, as implemented by the off-the-shelf information-retrieval software ElasticSearch 1.6 (Elasticsearch BV, 2016). TF-IDF is a way to normalize the importance or weight of a *token* proportional to its occurrence in a particular document (in our case, crash reports), and inversely proportional to its appearance in all documents. That means that common tokens that appear frequently in nearly *all* crash reports have little discriminative power compared to tokens that appear quite frequently in a small set of crash reports.

ElasticSearch uses a score based on TF-IDF. However, ElasticSearch uses its own additional weighting, smoothing, and distance metric, so it is not a pure TF-IDF implementation.

7.2.3 Background

Of course, the idea of crash bucketing is not new; Mozilla's system performs bucketing (Ahmed et al., 2014; Dhaliwal et al., 2011), as does WER (Glerum et al., 2009). Many approaches make the assumption that two crash reports are similar if their stacktraces are similar. Consequently, researchers (Bartz et al., 2008; Brodie et al., 2005a; Dang et al., 2012a; Dhaliwal et al., 2011; Glerum et al., 2009; Lerch and Mezini, 2013; Liu and Han, 2006; Modani et al., 2007; Wang et al., 2013; Wu et al., 2014) have proposed various methods of finding similar stacktraces, crash report similarity, crash report clustering, and crash report bucketing. In order to motivate the evaluation and design choices it is necessary to look at what already has been proposed.

Empirical evidence suggests that a function responsible for crash is often at or near the top of the crash stacktrace (Brodie et al., 2005a; Schröter et al., 2010; Wu et al., 2014). As such, many bucketing heuristics employ higher weighting for grouping functions near the top of the stack (Glerum et al., 2009; Modani et al., 2007; Wang et al., 2013). Many of these methods are similar to or extensions of the `1Frame` method, that assumes that the function name on the top of the stack is the most (or only) important piece of information for crash bucketing. However, at least one study refutes the effectiveness of truncating the stacktrace (Lerch and Mezini, 2013). The most influential discriminative factors seem to be function name (Lerch and Mezini, 2013) and module name (Bartz et al., 2008; Glerum et al., 2009).

Lerch and Mezini (2013) did not directly address crash report bucketing; they addressed *bug report* clusteringthrough stacktrace similarity. They clusteredbug reports that included stacktraces by comparing the traces with TF-IDF, which is usually applied to natural language text. Although crash bucketing was implicit in this approach to bug-report-clustering, the authors did not compare this technique against the other crash report clusteringtechniques. Unlike the signature-based methods, TF-IDF-based methods do not consider the order that frames appear on the stack. A function at the top of the stack is treated identically to a function at the bottom of the stack.

This chapter applies and evaluates the method in Lerch and Mezini (2013) of bug report clusteringto crash report clustering, both excluding *contextual* data from the crash report as suggested by Lerch and Mezini (2013) and including it. These methods are listed in the evaluation section as the `Lerch` method and the `LerchC` method, respectively. The automated crash reporting tools collected contextual data at the same time as the crash stacktrace. This chapter also evaluates variants of the `Lerch` and `LerchC` methods. `Space`, `SpaceC`, `Camel`, and `CamelC` were created for this evaluation based on tokenization techniques described by Elasticsearch BV (2016) and by including or excluding contextual information available in the crash reports. The variants replace the tokenization pattern used in `Lerch` and `LerchC` with a different

tokenization pattern. The name specifies the kind of tokenization–`Space` splits on whitespace only; `Camel` splits intelligently on CamelCasedComponents. If the name is followed by a `C`, the evaluation included the entire context of the stacktrace along with the stacktrace itself. Figure 7.1 shows how each method tokenizes a sample stack frame.

Modani et al. (2007) provides two techniques to improve performance of the various other algorithms. These techniques are inverted indexing and top-k indexing, both of which are evaluated in this chapter. Inverted indexing is employed to improve the performance of all of the TF-IDF-based methods including `Lerch` and `LerchC` (however Modani et al. (2007) did not use TF-IDF in their evaluation). The implementation is provided by ElasticSearch's version 1.6 (Elasticsearch BV, 2016) indexing system. Top-k indexing is employed to evaluate all of the methods that use the top portions of stacks, including `1Frame`, `2Frame`, `3Frame`, `1File`, etc.

7.2.4 Methods Not Appearing In This Report

Mozilla's clusteringtechnique, at the time of writing, as it is implemented in Socorro (*mozilla/socorro: Socorro is a server to accept and process Breakpad crash reports.* 2016) requires a large number of hand-written regular expressions to select, ignore, skip, or summarize various parts of the crash report. These must be maintained over time by Mozilla developers and volunteers in order to stay relevant to crashes as versions of Firefox are released. This technique typically uses one to three of the frames of the stack and likely has similar performance to `1Frame`, `2Frame`, and `3Frame`. Furthermore, the techniques employed by Mozilla are extremely specific to their major product, Firefox, while the evaluation data set contains crashes from 616 other systems.

Brodie et al. (2005a) presented an approach that normalizes the call stack to remove non-discriminative functions as well as flattening recursive functions, and compares stacks using weighted edit distance. Since pairwise stack matching would be infeasible on large data sets–having a minimum worst case run-time of $O(n^2)$–they index a hash of the top k function names at the top of

the stack and use a B+Tree look-up data structure. Several approaches since have used some stack similarity metric, and found that the most discriminative power is in the top-most stack frames—*i.e.*, the functions that are *closer* to the crash point.

Liu and Han (2006) grouped crashes together if they suggest the same fault location. The fault locations were found using a statistical debugging tool called SOBER (Liu et al., 2005), that, trained on failing and passing *execution traces* (based on instrumenting Boolean predicates in code (Liblit et al., 2005)), returns a ranked list of possible fault locations. Methods involving full instrumentation (Liu and Han, 2006) or static call graph analysis (Wu et al., 2014) are also deemed unfeasible, as they are not easy to incorporate into already existing software, and often incur pairwise comparisons to bucket regardless of instrumentation cost. Methods that already assume buckets such as Kim et al. (2011) and Wu et al. (2014) are disregarded as well.

Modani et al. (2007) propose several algorithms. The first algorithm employs edit distance, requiring $O(n^2)$ total time. The second and third algorithms are similar, employing longest common subsequences and longest common prefixes, respectively. The longest common subsequence problem is, in general, NP-hard in the number of sequences (corresponding to crashes for the purposes of this evaluation). The longest common prefix algorithm can be implemented sufficiently efficiently for the purposes of this evaluation, but was not evaluated here because it must produce at least as many buckets as the `1Frame` algorithm, that already creates too many buckets. Thus no Modani et al. (2007) comparison algorithms were used.

In addition to comparison algorithms that might be used for clusteringdirectly, Modani et al. (2007) also provide several algorithms for identifying frames that may be less useful in each stack and removing them from those stacks. These algorithms would then be combined with their other algorithms and are not evaluated in this chapter. One such algorithm removes frequent frames, such as `main()` that occur in many stacks. A similar effect is gained from TF-IDF, because the inverse document frequency reduces the weight of

terms that are found in many documents (crashes). These filtering techniques were not evaluated.

Bartz et al. (2008) also used edit distance on the stacktrace, but a weighted variant with weights learned from training data. Consequently, they were able to consider other data in the crash report aside from the stacktrace. The weights learned suggested some interesting findings: substituting a module in a call stack resulted in a much higher distance; as well, the call stack edit distance was found to be the highest-weighted factor, despite the consideration of other crash report data, confirming the intuition in the literature of the stacktrace's importance.

The methods based on edit distance [2] (Bartz et al., 2008; Brodie et al., 2005a; Modani et al., 2007) are disqualified due to their requirement of pairwise comparisons between stacktraces, with an upper-bound of $O\left(n^2\right)$.

Schröter et al. (2010) empirically studied developers' use of stacktraces in debugging and found that bugs are more likely to be fixed in the top 10 frames of their respective crash stacktrace, further confirming the surprising significance of the top-k stack frames in crash report bucketing, which is also corroborated more recently by Wu et al. (2014).

Glerum et al. (2009) describe the methods used by Microsoft's Windows Error Reporting (WER) service. Although they tout having over 500 heuristics for crash report bucketing—-many derived empirically—a large bulk of the bucketing is attributed to top-1 module offset; over 91% of bucketing is attributed to eight heuristics alone.

To avoid the $O(n^2)$ pairwise comparisons common to many of the previous approaches, Dhaliwal et al. (2011) proposed a weighted edit distance technique that creates *representative stacktraces*—a probability distribution based on all stacktraces seen within a bucket. Thus, instead of computing similarity against all stacktraces in a bucket, one would only use the weights derived from all stacktraces in the bucket simultaneously.

[2] They first use naive methods for indexing as well, that *is* evaluated here

The method described in Dhaliwal et al. (2011) is not included in the evaluation because it first subdivides buckets produced by the `1Frame` clusteringmethod, and requires $O\left(|B|^2\right)$ total time to run, where $|B|$ is the number of buckets. Its use of the `1Frame` method already produces a factor of 1.67 times too many buckets. Despite the optimization in Dhaliwal et al. (2011) that attempts to avoid $O\left(n^2\right)$ behaviour, it has $O\left(|B|^2\right)$ behaviour. Since the number of buckets increases over time, though at a slower rate, this method will eventually become computationally unfeasible if old data is not discarded.

Kim et al. (2011) constructed *Crash Graphs*, that are simply directed graphs using stack frames as nodes and their adjacency to other stack frames as edges. This also proved to be a useful crash visualization technique.

Dang et al. (2012a) created the *position independent model* that places more weight on stack frames closer to the top of the stack; and favours stacks whose matched functions are similarly spaced from each other. Purporting significantly higher accuracy than previous methods, this technique suffers from a proposed $O(n^3)$ clustering algorithm.

Wang et al. (2013) propose three different methods, that they refer to as rules. The first rule requires an incoming crash to be compared to every existing crash, requiring $O\left(n^2\right)$ time. The second rule compares only the top frame of every crash by considering two crashes related if the file names in the top frame of the crash are the same. This method is listed in the evaluation as the `1File` method. The third rule requires a set of common "frequent closed ordered sub-sets" of stack frames to be extracted from known "crash types" that are pre-categorized groups of crashes that have been bucketed using a separate method. The third rule requires $O\left(|B|^2\right)$ total time where $|B|$ is the number of buckets created by the other method. Specifically the authors use the method of comparing the top frame from each stack, that is evaluated in this chapter as the `1Frame` method. This method appears to create a number of buckets roughly proportional to the number of seen crashes, n. Thus, the third rule requires $O\left(n^2\right)$ total time, though with a low coefficient. The only method

from Wang et al. (2013) directly evaluated in this chapter is the method of comparing file names at the top of the stack.

Thus, there are many approaches for bucketing crash reports and crash report similarity, but some are less realistic or industrially applicable than others. Any new work in the field must attempt to compare itself against some of the prior techniques such as Lerch and Mezini (2013).

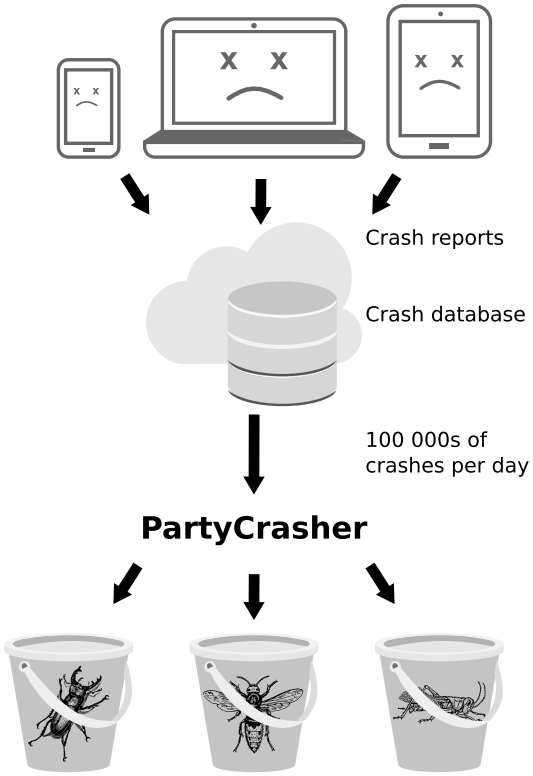

Figure 7.2: PartyCrasher within a development context

7.3 Methodology

First, the requirements for an industrial-scale automated crash clusteringsystem were characterized by looking at systems that are currently in use. Then, a variety of methods from the existing literature were evaluated for applicability to the task of automated crash report clustering. Several methods that met the requirements were selected. A general purpose Python framework in which any of the selected clusteringmethods could be supported and evaluated

was developed, and then used to evaluate all of the methods by simulating the process of automated crash reports arriving over time. Additionally, a data set that could be used as a gold set to judge the performance of such methods was obtained. The data set was then filtered to include only crash reports that had been clusteredby human developers and volunteers.

Various approaches of automatic crash report categorization (the exact problem that Ada is tasked with solving) is simulated. First, a crash report arrives with no information other than what was gathered by the automated reporting mechanisms on the user's machine. This report might include a description written by the user of what they were doing when the crash occurred. Figure 7.3 is an example of one of the crash reports used in the evaluation with a user-submitted description on the second line, metadata in the middle, and a stacktrace on the bottom.

7.3.1 Mining Crash Reports

The first step in the evaluation procedure is mining of crash reports from Ubuntu's bug repository, Launchpad (Canonical Ltd., 2004). This was done using a modified version of Bicho (Robles et al., 2011), a software repository mining tool.[3] Over the course of one month, Bicho was able to retrieve 126 609 issues from Launchpad, including 80 478 stacktraces in 44 465 issues. Some issues contain more than one stacktrace. For issues that contained more than one stacktrace, the first stacktrace posted to that issue was selected, yielding 44 465 issues with crash reports and stacktraces. The first stacktrace is selected because it is the one that arrives with the automated crash report, generated by the instrumentation on the user's machine.

Ubuntu crash reports were used for the evaluation because they are automatically generated and submitted but many of them have been manually clusteredby Ubuntu developers and volunteers. Other data sources, such as Mozilla's Crash Reports have already been clusteredby Mozilla's own automated system, not by humans.

[3] `https://github.com/hazelybell/Bicho/`

```
Binary package hint: evolution-exchange

I just start Evolution, wait about 2 minutes, and then evolution-exchange
crashed

ProblemType: Crash
Architecture: i386
CrashCounter: 1
Date: Tue Jul 17 10:09:50 2007
DistroRelease: Ubuntu 7.10
ExecutablePath: /usr/lib/evolution/2.12/evolution-exchange-storage
NonfreeKernelModules: vmnet vmmon
Package: evolution-exchange 2.11.5-0ubuntu1
PackageArchitecture: i386
ProcCmdline: /usr/lib/evolution/2.12/evolution-exchange-storage --oaf-activate-i
ProcCwd: /
ProcEnviron:
 PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
 LANG=en_US.UTF-8
 SHELL=/bin/bash
Signal: 11
SourcePackage: evolution-exchange
Title: evolution-exchange-storage crashed with SIGSEGV in soup_connection_discon
Uname: Linux encahl 2.6.20-15-generic #2 SMP Sun Apr 15 07:36:31 UTC 2007 i686 G
UserGroups: adm admin audio cdrom dialout dip floppy kqemu lpadmin netdev plugde

#0  0xb71e8d92 in soup_connection_disconnect () from /usr/lib/libsoup-2.2.so.8
#1  0xb71e8dfd in ?? () from /usr/lib/libsoup-2.2.so.8
#2  0x080e5a48 in ?? ()
#3  0xb6eaf678 in ?? () from /usr/lib/libgobject-2.0.so.0
#4  0xbfd613e8 in ?? ()
#5  0xb6e8b179 in g_cclosure_marshal_VOID__VOID ()
   from /usr/lib/libgobject-2.0.so.0
Backtrace stopped: frame did not save the PC
```

Figure 7.3: An example crash report, including stack.

Next, the issues were put into groups based on whether they were marked as duplicates of another issue, resulting in 30 664 groups of issues. These groups are referred to as "issue buckets" for the remainder of the chapter, to prevent confounding with groups of crash reports, that will be referred to as "crash buckets." This data set is available![4]

7.3.1.1 Stacktrace Extraction

Each issue and stacktrace obtained from Ubuntu is formatted as plain text, as shown in Figure 7.3. They were then parsed into JSON-formatted data with individual fields for each item, such as address, function name, and which library the function came from. Unfortunately, this formatting is not always consistent and may be unusable. For example, some stacktraces contain unintelligible binary data in place of the function name. This could be caused by memory corruption when the stacktrace was captured. 2216 crash reports and stacktraces were thrown out because their formatting could not be parsed, leaving 41 708 crash reports with stacktraces.

7.3.1.2 Crash Report and Stacktrace Data

Issues were then filtered to only those that had been clusteredby Ubuntu developers and other volunteers, yielding 15 293 issues with 15 293 stacktraces in 3824 issue buckets. These crash reports were submitted to Launchpad by the Apport tool.[5] They were collected over a one month period. Because Launchpad places restrictions on how often the Launchpad API can be used to request data, and each crash report required multiple requests, it required over 20 seconds to download each issue. The crash reports used in the evaluation span 617 different source packages, each of which represents a software system. The only commonalities between them are that they are all written in C, C++, or other languages that compile to binaries debuggable by a C debugger, and that they are installed and used on Ubuntu. The most frequently reported software

[4] `https://archive.org/details/bugkets-2016-01-30`
[5] `https://launchpad.net/apport`

system is Gnome[6], which has 2154 crash reports with stacktraces. This data set is large, comprehensive and covers a wide variety of projects.

7.3.2 Crash Bucket Brigade

In order to simulate the timely nature of the data, each report is added to a simulated crash report repository *one at a time*. This is done so that no method can access data "from the future" to choose a bucket to assign a crash report to. It is first assigned a bucket based on the crashes and buckets already in the simulated repository, then it is added to the repository as a member of that bucket.

7.3.3 Deciding when a Crash is not Like the Others

For methods based on Lerch and Mezini (2013), a naive clustering algorithm is used. It was not based on a clustering algorithm in the clustering literature. The clustering algorithm was not an attempt to approximate any known clustering algorithm or to be optimal under any specific metric.

The clustering is hierarchical, but neither bottom-up or top-down. Instead, only a single pass through the data is allowed. However, it is most similar to a single-linkage clustering algorithm (Murtagh and Contreras, 2012). Despite the similarity, the algorithm never computes cluster-cluster similarity for two clusters which both have more than one element. Two clusters with more than one element will never be merged, regardless of how similar they are.

The algorithm is also similar to a minimum spanning tree algorithm such as Prim's algorithm (Sedgewick and Wayne, 2011). However, vertices are not introduced in the order that minimizes some function. Vertices are introduced in the order in which they are generated by an external, unpredictable process. The first two vertices *in time* are always connected by an edge. The third vertex is always connected to one of the first two, and so on.

[6] `https://www.gnome.org/`

The the distance between two elements also changes over time due to the IDF (inverse document frequency) component in TF-IDF. Thus, the clustering produced depends heavily on the order in which elements are introduced.

Each crash is placed in the same bucket as the highest scoring crash returned by ElasticSearch search, unless that score does not meet a threshold, T. T determines how often, and when, an incoming crash report is assigned to a new bucket. A specific value for T was not described by Lerch and Mezini (2013), so a range of different values from 1.0 to 10.0 were evaluated. Higher values of T will cause the algorithm to create new buckets more often.

The threshold value applies to the *score* produced by the Lucene search engine inside ElasticSearch 1.6 (Elasticsearch BV, 2016). Details of this TF-IDF based scoring method are described within the ElasticSearch documentation.[7] The scoring algorithm is based on TF-IDF, but contains a few minor adjustments intended to make scores returned from different queries more comparable. The output of the scoring function is not directly comparable with cosine distance or any other formally defined distance metric. The score was designed by the creators of ElasticSearch and Lucene to rank results. A value of 1.0 is indicates a result more similar to the query than smaller values, such as 0.5, and more dissimilar than other values, such as 2.0.

7.3.4 Implementation

The complete implementation of the evaluation presented in this chapter is available in the open-source software PartyCrasher.[8] The implementation includes every clusteringmethod we claimed to evaluate above, a general-purpose clusteringframework, the programs used to mine and filter the data used for the evaluation, the programs that produced the evaluation results, the raw evaluation results, and the scripts used to plot them.

[7] `https://www.elastic.co/guide/en/elasticsearch/guide/1.x/practical-scoring-function.html`

[8] `https://github.com/naturalness/partycrasher`

7.3.5 Evaluation Metrics

Two families of evaluation metrics were used. These are the *BCubed* precision, recall, and F_1-score, and the Purity, Inverse Purity, and F_1-score. Both are suitable for characterizing the performance of online non-stationary clustering algorithms by comparing the clusters that evolve over time to clusters created by hand. A comparison of BCubed and Purity, along with several other metrics, and an argument for the advantages of BCubed over Purity is provided in Amigó et al. (2009). The mathematical formulae for both metrics can be found in Amigó et al. (2009). However, Purity also has an advantage over BCubed: specifically that it does not require $O(n^2)$ total time to compute whereas BCubed does.

If a method has a high BCubed precision, this means that there would be less chance of a developer finding unrelated crashes in the same bucket. This is important to prevent crashes caused by two unrelated bugs from sharing a bucket, possibly causing one bug to go unnoticed since usually a developer would not examine all of the crashes in a single bucket.

If a method has a high BCubed recall, this means that there would be less chance of all the crashes caused by a single bug to become separated into multiple buckets. Reducing the scattering of a single bug across multiple buckets is important as scattering interferes with statistics about frequently experienced bugs.

In contrast, Purity and Inverse Purity focus on finding the bucket in the experimental results that most closely matches the bucket in the gold set. Then the overlap between the two closest matching buckets is used to compute the Purity and Inverse Purity metrics, with high Purity indicating that most of the items in a bucket produced by one of the methods evaluated are also in the matching bucket in the gold set. High Inverse Purity indicates that most of the items in a bucket from the gold set are found in the matching bucket produced by the method being evaluated.

The Purity method does not, however, completely reflect the goals of the evaluation. Purity and Inverse Purity do not capture anything besides the

overlap between the gold-set buckets and the output buckets that overlap the most. The Purity metric was defined in Zhao and Karypis (2001) as:

$$\text{Purity} = \frac{1}{\left(\begin{smallmatrix}\text{total number}\\ \text{of documents}\end{smallmatrix}\right)} \sum_{b \in \text{clusters}} \left[\max_{g \in \text{classes}} \left(\begin{smallmatrix}\text{number of documents in class } g\\ \text{that were assigned cluster } b\end{smallmatrix} \right) \right]$$

In this case, the class is the bucket in the gold set, and the cluster is the bucket produced by the method being evaluated. The Inverse Purity metric, then, follows from the Purity metric by swapping the roles of the "class" and the "cluster" (Amigó et al., 2009), or in our case, the gold set and the method being evaluated:

$$\text{Inverse Purity} = \frac{1}{\left(\begin{smallmatrix}\text{total number}\\ \text{of documents}\end{smallmatrix}\right)} \sum_{g \in \text{classes}} \left[\max_{b \in \text{clusters}} \left(\begin{smallmatrix}\text{number of documents in class } g\\ \text{that were assigned cluster } b\end{smallmatrix} \right) \right]$$

So, if a method creates a bucket that is 51% composed of crashes from a single bug, the other 49% does not matter. That 49% could come from a different bug, or 200 different bugs, but the Purity would be the same value. The Purity value isn't affected by the other 49% because of the use of the max function in Purity and Inverse Purity. It is included in this evaluation for completeness, since it was used by Dang et al. (2012a).

In comparison, BCubed Precision and BCubed Recall based on the idea of the correct bucketing of a pair of documents (crashes). First, let $c_i \in b_i, c_i \in g_i$ be some crash that was assigned bucket b_i by a bucketing method, and bucket g_i in the gold set. Similarly, let $c_j \in b_j, c_j \in g_j$ be some crash that was assigned b_j and was in g_j in the gold set. Then c_i and c_j are bucketed correctly if they are placed in the same bucket and they should be in the same bucket, or they are in different buckets and should be in different buckets:

$$\text{Correct}\,(c_i, c_j) = \begin{cases} 1, & \text{if } b_i \equiv b_j \text{ and } g_i \equiv g_j \\ 1, & \text{if } b_i \not\equiv b_j \text{ and } g_i \not\equiv g_j \\ 0, & \text{if } b_i \equiv b_j \text{ and } g_i \not\equiv g_j \\ 0, & \text{if } b_i \not\equiv b_j \text{ and } g_i \equiv g_j \end{cases}$$

BCubed Precision is the average correctness over all pairs of documents (crashes) that were placed in the same bucket by the bucketing method, while BCubed Recall is the average correctness over all pairs of documents that were in the same bucket in the gold set.

$$\text{BCubed Precision} = \underset{c_j, c_j \text{ such that: } b_i \equiv b_j}{\text{average}} \begin{cases} 1, & \text{if } g_i \equiv g_j \\ 0, & \text{if } g_i \not\equiv g_j \end{cases}$$

$$\text{BCubed Recall} = \underset{c_i, c_j \text{ such that: } g_i \equiv g_j}{\text{average}} \begin{cases} 1, & \text{if } b_i \equiv b_j \\ 0, & \text{if } b_i \not\equiv b_j \end{cases}$$

In comparison, BCubed Precision and BCubed Recall based on the idea of the correct bucketing of a pair of documents (crashes). First, let $c_i \in b_i, c_i \in g_i$ be some crash that was assigned bucket b_i by a bucketing method, and bucket g_i in the gold set. Similarly, let $c_j \in b_j, c_j \in g_j$ be some crash that was assigned b_j and was in g_j in the gold set. Then c_i and c_j are bucketed correctly if they are placed in the same bucket and they should be in the same bucket, or they are in different buckets and should be in different buckets:

$$\text{Correct}\,(c_i, c_j) = \begin{cases} 1, & \text{if } b_i \equiv b_j \text{ and } g_i \equiv g_j \\ 1, & \text{if } b_i \not\equiv b_j \text{ and } g_i \not\equiv g_j \\ 0, & \text{if } b_i \equiv b_j \text{ and } g_i \not\equiv g_j \\ 0, & \text{if } b_i \not\equiv b_j \text{ and } g_i \equiv g_j \end{cases}$$

BCubed Precision is the average correctness over all pairs of documents (crashes) that were placed in the same bucket by the bucketing method, while BCubed Recall is the average correctness over all pairs of documents that were in the same bucket in the gold set.

$$\text{BCubed Precision} = \underset{c_j, c_j \text{ such that: } b_i \equiv b_j}{\text{average}} \begin{cases} 1, & \text{if } g_i \equiv g_j \\ 0, & \text{if } g_i \not\equiv g_j \end{cases}$$

$$\text{BCubed Recall} = \underset{c_i, c_j \text{ such that: } g_i \equiv g_j}{\text{average}} \begin{cases} 1, & \text{if } b_i \equiv b_j \\ 0, & \text{if } b_i \not\equiv b_j \end{cases}$$

Both metrics can be combined into F-scores. In this evaluation, F_1-scores were used, placing equal weight on BCubed Precision and Recall (or Purity and Inverse Purity.)

BCubed and Purity can be used with the gold set, hand-made buckets that are available from Ubuntu's Launchpad (Canonical Ltd., 2004) bug tracking system. Ubuntu developers and volunteers have manually marked many of the bugs in their bug tracker as duplicates. Furthermore, many of the bugs in the bug tracker are automatically filed by Ubuntu's automated crash reporting system, Apport. This evaluation uses only bugs that were both automatically filed by Apport and manually marked as duplicates of at least one other bug. The data set is biased to the distribution of crashes that are bucketed, which might be different than crashes that are not. Conversely, this prevents the evaluation data set from containing any crashes that have not yet evaluated by an Ubuntu developer or volunteer.

7.4 Results

After extracting crash reports from Launchpad, and implementing various crash report bucketing algorithms, the performance of these algorithms on the Launchpad gold set was evaluated. Evaluation is multifaceted as in most information retrieval studies since the importance of either precision or recall are tunable.

7.4.1 BCubed and Purity

Evaluation of the performance of bucketing algorithms is performed with BCubed and Purity metrics. Figure 7.4 shows the performance of a variety of clusteringmethods evaluated against the entire gold set of clusteredcrash reports. The `1File` and `1Addr` methods have the most precision, while `LerchC` has the most recall. F_1-score is dominated by `CamelC` and `Lerch`. As in the results of Lerch and Mezini (2013), using only the stacks outperforms using the stack plus its metadata and contextual information in terms of F_1-score. For the `CamelC`, `Lerch`, and `LerchC` simulations, a threshold of $T = 4.0$ was used.

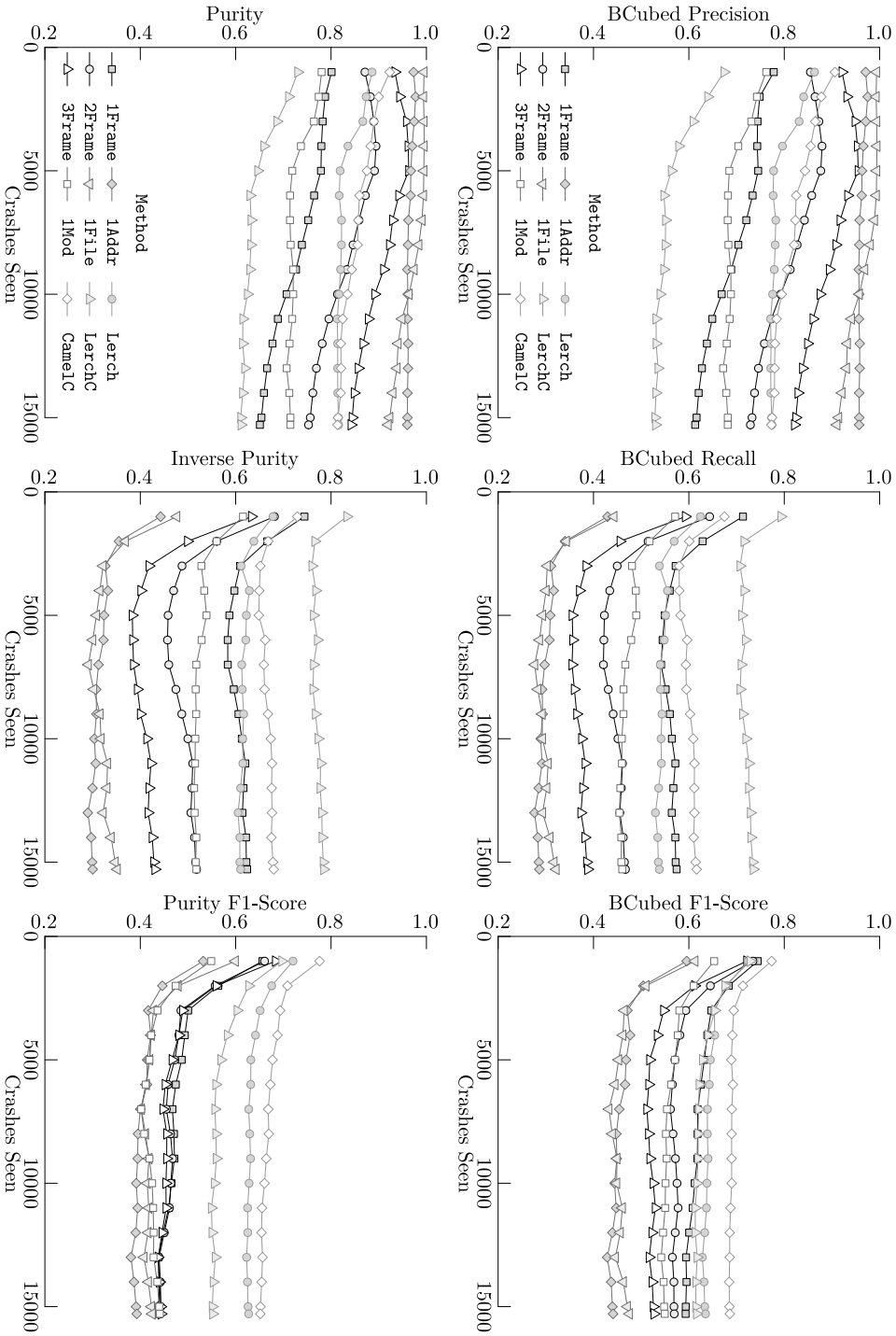

Figure 7.4: BCubed (top) and Purity-metric (bottom) scores for various methods of crash report clustering.

Amigó et al. (2009) observed differences in BCubed and Purity metrics. Their observation was tested empirically by the evaluation. In Figure 7.4, BCubed and Purity showed similar results. The best and worst methods in terms of BCubed precision are the same as the best and worst methods in terms of Purity; the same holds true for BCubed recall and Inverse Purity, and BCubed F_1-score and Purity F_1-score. However, some of the methods with intermediate performance are much closer together in Purity F_1-score than they are in BCubed F_1-score.

Figure 7.4 also shows that in general, if a method has a higher precision or Purity, it also has a lower recall and Inverse Purity. For example, `3Frame` has a higher precision than `2Frame`, having a higher precision than `1Frame`, but `1Frame` has a higher recall than `2Frame` and `3Frame`.

The `CamelC` crash bucketing method employs: TF-IDF; a tokenizer that attempts to break up identifiers such as variable names into their component words; and the entire context of the crash report including all fields reported in addition to the stack. It outperforms other bucketing methods evaluated.

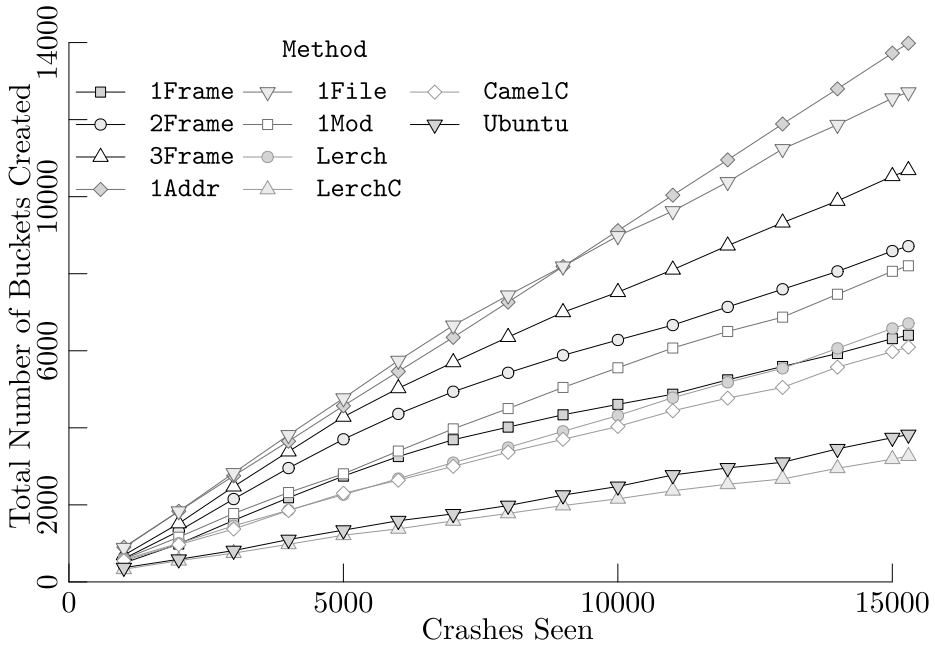

Figure 7.5: Number of buckets created as a function of number of crashes seen. The line labelled `Ubuntu` indicates the number of groups crashes that were marked as duplicates of each other by Ubuntu developers or volunteers.

7.4.2 Bucketing Effectiveness

Figure 7.5 shows the number of buckets created by a variety of clusteringmethods. The number of issue buckets extracted from the Ubuntu Launchpad gold set is plotted as the line labelled `Ubuntu`. The method that created a number of buckets most similar to the number mined from the Ubuntu Launchpad gold set was `LerchC`. For the `Lerch` and `LerchC` simulations, a threshold of $T = 4.0$ was used.

Figure 7.6 shows the performance of the `Lerch` method when used with a variety of different new-bucket thresholds, T. Figure 7.7 shows the number of buckets created by the same method with those same thresholds. Since Lerch and Mezini (2013) did not specify what threshold they used, this evaluation explored a range of thresholds. It can be seen from the plots that the relative performance of T thresholds, in terms of BCubed precision, BCubed recall, and BCubed F_1-score, becomes apparent after only 5000 crash reports. In practice, the authors of this chapter suggest that developers using this system start with a middle F_1-score of around $T = 4.0$ and adjust it as they use the system, rather than systematically examining thousands of crash reports.

It is possible for developers using this system to create multiple sets of buckets with different thresholds. This can be done efficiently as the crash reports are received, and would allow developers to choose a threshold at any time without re-bucketing. The implementation only requires a single query and can produce multiple buckets for each incoming crash report, since the threshold is applied after results from ElasticSearch are retrieved.

For all the results that do not specify a value for T, $T = 4.0$ was used. The highest F_1-score was observed at $T = 4.0$ after only processing 5000 bugs with a variety of different thresholds. For `Lerch`, a threshold of $3.5 < T < 4.5$ had the highest performance.

As shown in figure 7.8, $T = 4.0$ still has the highest F_1-score after every crash was processed. Furthermore, other values of T near 4.0 have the same F_1-score, including the range $3.5 \leq T \leq 4.5$. Figure 7.8 also shows how the threshold can be tuned to create a trade-off between precision and recall.

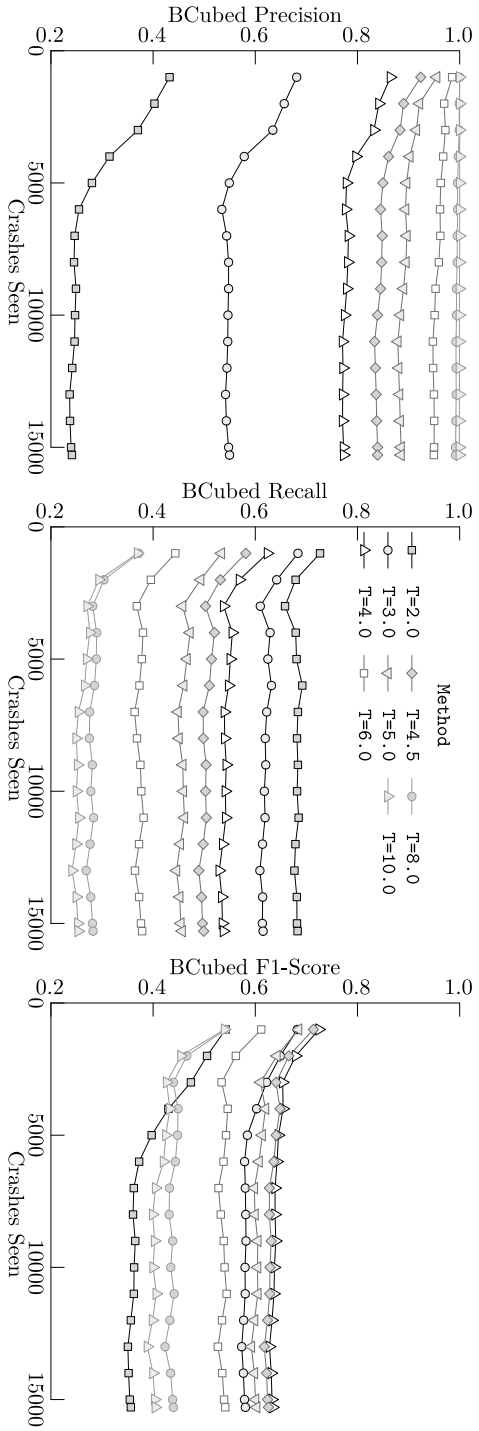

Figure 7.6: BCubed scores for the Lerch method of crash report clusteringat various new-bucket thresholds T.

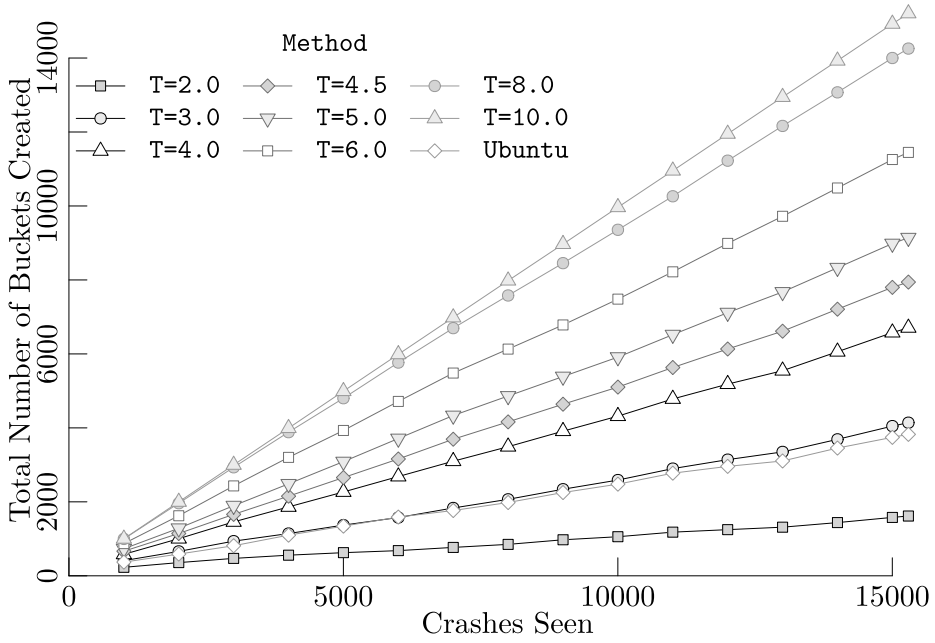

Figure 7.7: Number of buckets created as a function of number of crashes seen for the Lerch method of crash report clusteringat various new-bucket thresholds T. The line labelled `Ubuntu` indicates the number of groups crashes that were marked as duplicates of each other by Ubuntu developers or volunteers.

Setting a threshold of 0.0 is similar to instructing the system to put all of the crashes into a single bucket. This would be the correct choice if developers were satisfied with the explanation that all of those crashes were created by a single bug. In that case the bug would likely be filed as an issue titled, "Programs on Ubuntu Crash." The fact that setting the threshold to 0.0 does not result in recall quite at 1.0 is an artifact of optimizations employed in ElasticSearch, specifically ElasticSearch's inverted index.

Conversely, setting the threshold to 10.0 results in every crash being assigned to its own bucket, and therefore a perfect precision of 1.0. This would be the correct choice if developers considered every individual crash to be a distinct bug because the exact state of the computer was at least somewhat different during each crash. It might be more desirable to tune the value of T by using direct developer feedback rather than the technique employed here, comparing against an existing data set. Instead of using data, one could ask developers if they had seen too many crashes caused by unrelated bugs in a

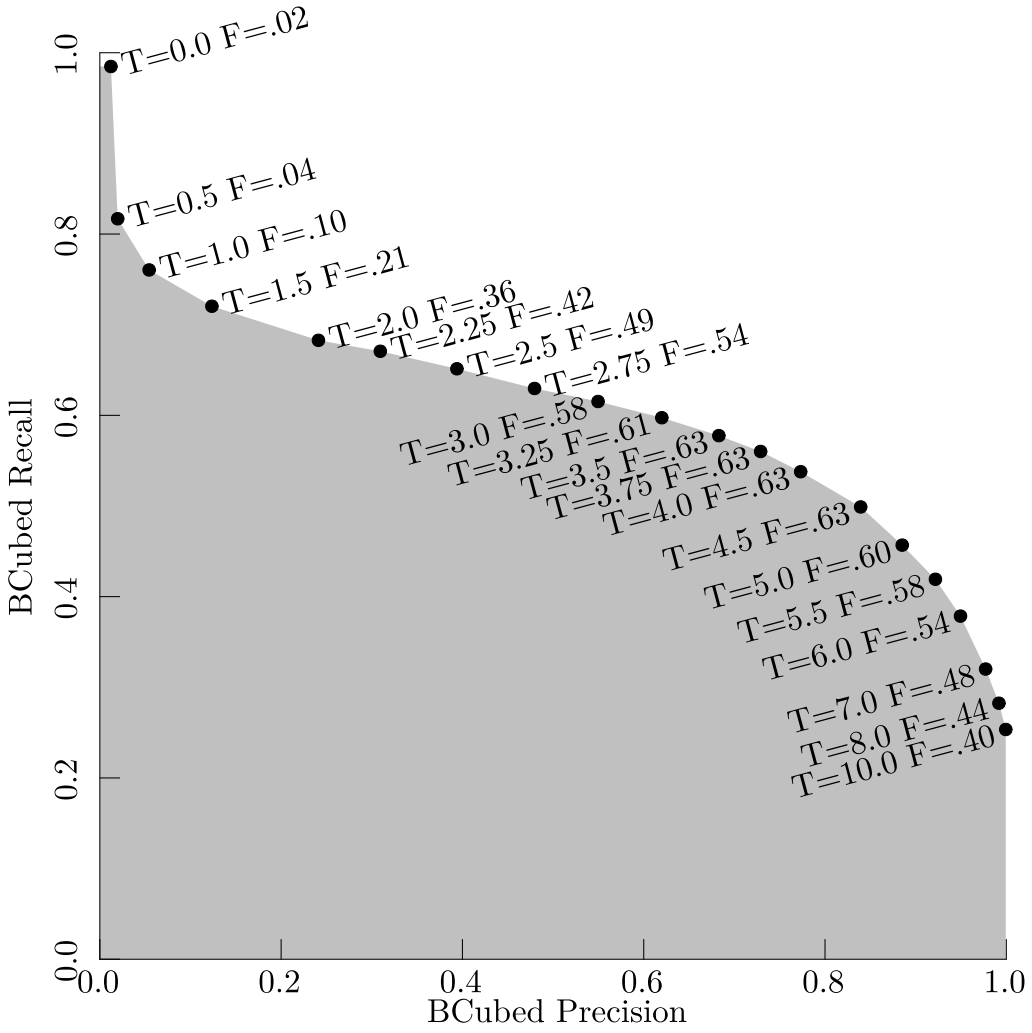

Figure 7.8: Precision/Recall plot showing the trade-off between BCubed precision and recall as the new-bucket threshold T is adjusted. BCubed F_1-score is also listed in the plot.

single bucket recently. If they had, then T should be increased. Or, T should be decreased if developers see multiple buckets that seemed to be focused on crashes caused by the same bug.

7.4.3 Tokenization

Threshold is not the only way that a trade-off between precision and recall can be made. A variety of methods were tested that use the ElasticSearch/Lucene TF-IDF-based search from Lerch and Mezini (2013), but do not follow their tokenization strategy. The performance of several tokenization strategies is shown in Figure 7.10. As in other cases, the methods with high precision had low recall, and the methods with high recall had low precision. All methods shown in Figure 7.10 used a threshold of $T = 4.0$.

The `Space` method is obtained by replacing the tokenization strategy in `Lerch` with one that splits words on whitespace only, such that it does not discard any tokens regardless of how short they are, and does not lowercase every letter in the input. The `Space` method performs worse than `Lerch`. However, when both stacktraces and context are used, the `SpaceC` method, performance improves slightly. This is the opposite behaviour of `Lerch`. Adding context (`LerchC`) causes performance to decrease slightly. A third tokenization strategy, `Camel` was evaluated. `Camel` attempts to break words that are written in CamelCase into their component words, using a method provided in the ElasticSearch documentation.[9] This strategy had the worst performance of the three, until it was used with context included, called `CamelC`. The addition of context allowed `CamelC` to outperform every other method evaluated in this chapter.

The worst-performing tokenization evaluated, `1Addr`, was also the method that produced the largest number of buckets. However, tuning methods to match the number of buckets in the gold set without concern for performance did not result in higher performance. `Lerch` with $T = 3.0$ and `SpaceC` with

[9] `https://github.com/elastic/elasticsearch/blob/1.6/docs/reference/analysis/analyzers/pattern-analyzer.asciidoc`

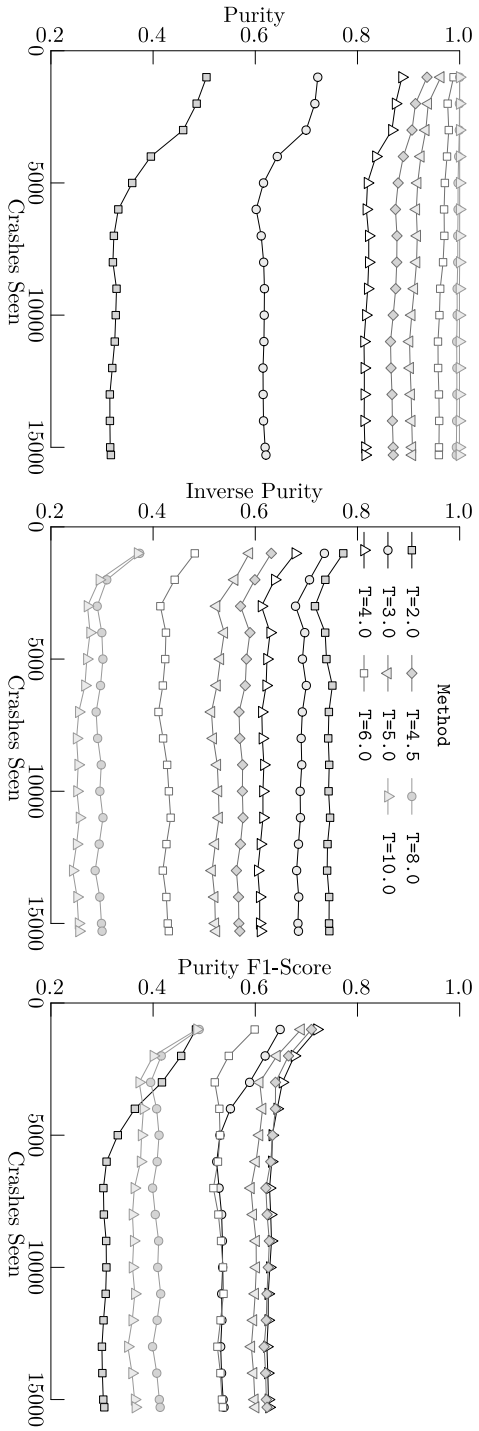

Figure 7.9: Purity-metric scores for the Lerch method of crash report clusteringat various new-bucket thresholds T.

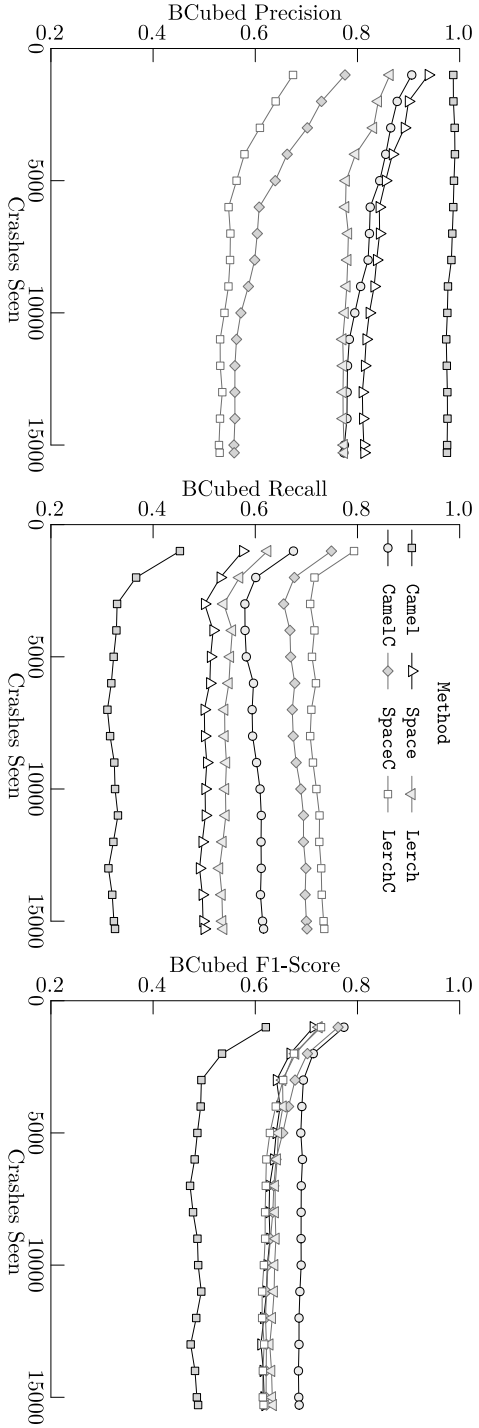

Figure 7.10: BCubed scores for the Lerch method of crash report clustering-with Lerch's tokenization technique replaced by a variety of other techniques.

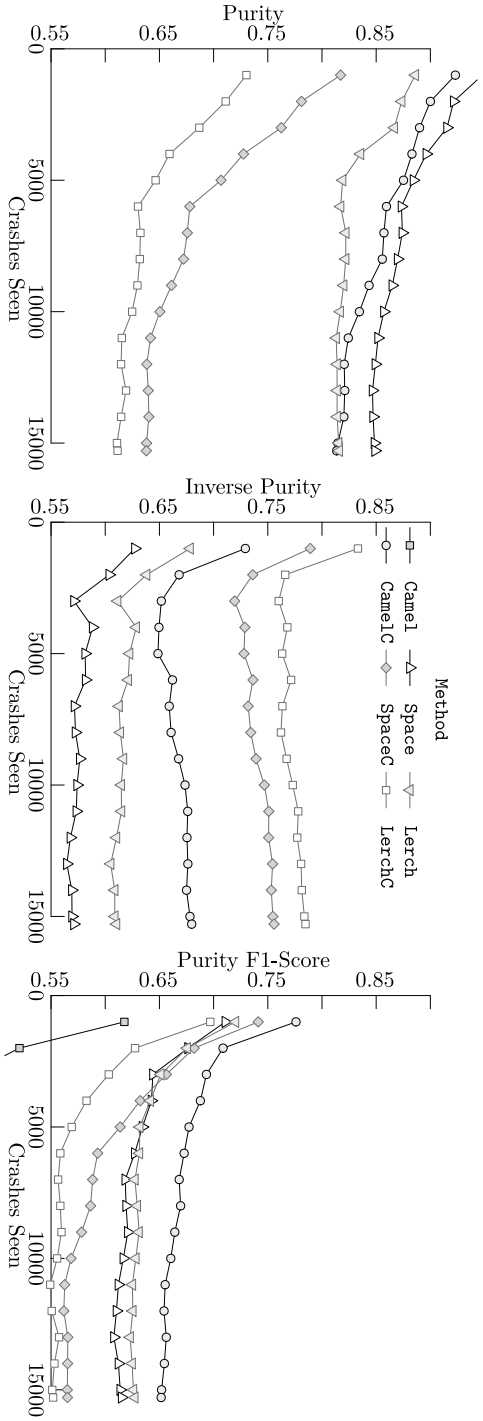

Figure 7.11: Purity-metric scores for the TF-IDF-based methods of crash report clusteringwith various tokenization strategies.

$T = 4.0$ were not the best-performing threshold or method, but both produced almost the same number of buckets as the gold set.

7.4.4 Runtime Performance

The current implementation of PartyCrasher requires only 45 minutes to bucket and ingest 15 293 crashes, using the slowest algorithm, `CamelC`, on a Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz machine with 32GiB of RAM and a Hitachi HDS723020BLE640 7200 RPM hard drive. Performance depends mainly on disk throughput, latency and RAM available for caching; ElasticSearch recommends using only solid-state drives. This works out to 335 crashes per minute, meeting the performance goal of 217 crashes per minute based on crash-stats from Mozilla. The performance of ElasticSearch is highly dependent on ElasticSearch's configuration settings. The settings used during these evaluations is available in the PartyCrasher repository.

7.5 Discussion

7.5.1 Threats to Validity

Results are dependent on the gold set—a manual classification of crash report by Ubuntu volunteers. The results may be biased due to the exclusive use of known duplicate crashes; the known and classified duplicates may not be representative of all crash reports. If any of these methods with with tunable parameters are deployed, the parameters should be tuned based on feedback from people working with the crash buckets, not just the gold set.

Since the evaluation only used data from open source software, it is unknown if our results are applicable to closed-source domains. Only stacks that originate from C and C++ projects have been evaluated; it is possible that other languages, compilers, and their runtimes have different characteristics in how they form stacktraces. However, these results are corroborated by studies that examined Java exclusively (Lerch and Mezini, 2013; Wang et al., 2013).

7.5.2 Related work

Although crash bucketing facilitates manual debugging of individual faults, crash buckets are much more beneficial as the input to other methods in soft-

ware engineering. Lerch and Mezini (2013) originally applied their technique to the field of deduplicating bug, not crash, reports; Khomh et al. (2011) used crash buckets to triage bugs: prioritizing developer effort on the most crucial bugs. Seo and Seo and Kim (2012a) leveraged crash buckets to predict "recurring crashes"—*i.e.,* bugs that were "fixed" but had to be fixed again in a later revision. Crash buckets may also serve as input to crash localization (Liu and Han, 2006; Wang et al., 2013; Wu et al., 2014) and crash visualization (Dang et al., 2012a; Kim et al., 2011).

7.5.3 Future Work

The results in this chapter indicate that there may be a large number of improvements that could be made to the relatively high-performance TF-IDF-based crash clusteringmethods.

Many stack comparison methods (Dhaliwal et al., 2011; Glerum et al., 2009; Modani et al., 2007; Wang et al., 2013), take into account the position that each frame is on the stack, giving more weight to the frames near the top of the stack and less weight to frames on the bottom of the stack, or consider stacks that have similar frames in a similar order. The best-performing method of crash clusteringpresented in this chapter completely disregards information about the order of the stack. It is likely that a technique based on TF-IDF that also incorporates information about the order of frames on the stack would outperform all of the methods evaluated in this chapter, since several previous works indicate that the top of the stack contains the most important information (Brodie et al., 2005a; Schröter et al., 2010; Wu et al., 2014). This could be achieved by giving words that appear in the top of the stack more weight when computing TF-IDF or by re-ranking the top results produced by TF-IDF according to stack similarity before choosing a bucket to place a crash in. Neither of these extensions would cause the method to be unable to scale.

The results presented indicate that improvements could be made to TF-IDF-based-crash clusteringmethods. For instance, a technique based on TF-

IDF that also incorporates information about the order of frames on the stack would likely outperform many of the presented methods.

The tokenization techniques evaluated in this chapter are extremely primitive. They are merely regular expressions that break up words based on certain types of characters such as spaces, symbols, uppercase letters, lowercase letters and numbers. Advanced tokenization techniques, such as the ones found in Guerrouj et al. (2013) and Hill et al. (2014), would likely outperform the basic techniques that have been evaluated in this chapter.

As shown in Figure 7.3, crash reports often contain a multitude of data apart from the stacktrace itself. This chapter only measured the performance of TF-IDF when using only the stacktrace or the entire crash report. Some fields in the crash report may be more important to obtaining a high performance than others. For example, `Architecture` (the computer architecture on which the crash occurred) might be more valuable for clusteringthan `CrashCounter` (the number of times that a crash has occurred on that computer) or vice-versa, but this has not been studied in the context of information retrieval.

We would like to extend information retrieval techniques with more sophisticated normalization. We want to investigate any effects that stack normalization, as first proposed by Brodie et al. (2005a), would have on our TF-IDF approach.

It would be valuable to measure the effectiveness of using the buckets produced by the `CamelC` technique as input to other methods, such as those that perform bug triaging (Khomh et al., 2011) and crash localization (Wu et al., 2014).

7.6 Conclusion

The results in this chapter indicate that off-the-shelf TF-IDF-based information retrieval tools can bucket crash reports in a completely unsupervised, large-scale setting when compared to a variety of other previously proposed

algorithms. Based on these results, a developer, such as Ada, should choose a TF-IDF-based crash clusteringmethod with tokenization that fits their data set, and intermediate new-bucket threshold. They should update this threshold based on feedback from developers, volunteers, or employees that work with the stacktraces directly. A TF-IDF approach that used the entire crash report and stacktrace, tokenized using camel-case had the best F_1-score on the Ubuntu Launchpad crash reports used in this work. In addition, there is a lot of room for improvements to these techniques. This conclusion is surprising in light of the fact that the TF-IDF-based techniques evaluated disregard information that is often considered to be essential to stacktraces, such as the order of the frames in the stack.

Finally the research questions can be answered:

RQ1: TF-IDF-based methods are effective, industrial-scale methods of crash report bucketing.

RQ2: New-bucket thresholds and tokenization strategies can be tuned to increase precision and recall.

Acknowledgements

The authors would like to thank the Mozilla foundation, especially Robert Helmer, Adrian Gaudebert, Peter Bengtsson and Chris Lonnen for their help, and for making Mozilla's massive collection of stack reports open and publicly available. Additionally, the authors would like to thank the Ubuntu project and all of its developers who manually clusteredbug reports that were submitted with crash reports. Funding for this research was provided by MITACS Accelerate with BioWare™, an Electronic Arts Inc. studio.

7.7 Errata

After publication of the work in this chapter in 2016, I found that the algorithm presented in this chapter's time complexity degrades for a specific type of input. Thus, PartyCrasher does not meet the time-complexity claims made in this

chapter in some cases. Input data that would cause performance degradation is not present in the evaluation data used in this chapter.

Degradation occurs after PartyCrasher buckets a large group of identical crash reports, causing the underlying search engine, ElasticSearch, to fail to meet performance expectations. Time taken to bucket an additional crash report increases linearly, as long as that crash report is also identical, resulting in an overall $O(n^2)$ time complexity. This behaviour is present in the implementation of ElasticSearch that was used "off-the-shelf," and not modified, and would similarly affect ElasticSearch's performance in other applications.

Fundamentally, the performance degradation occurs when ElasticSearch finds a large number of highly relevant documents, all with the same relevance to the query. These documents would all be tied to be the first result in the output of a typical search engine application. In addition, the query must have a large of terms (tokens or words), and ElasticSearch must fail to find any search term in the query that would break the tie based on that term's document frequency.

Chapter 8

Conclusion

This chapter concludes this thesis by summarizing the results, exploring avenues for future research, and finally, providing some concluding remarks.

8.1 Summary

The results presented throughout this thesis demonstrate that some software-engineering tasks, such as typo repair and crash bucketing, can be performed with on-line, off-the-shelf, statistical techniques, just as well, if not better than by using static rule-based approaches. I have demonstrated that fact with two "unnatural tools" (section 2.2) that were originally intended to work natural-language artifacts adapted to work with source code and crash reports. I have analyzed and presented a novel data set for use with one of these tools in Chapter 6.

In Chapter 3 I introduced UnnaturalCode, a technique and tool for locating the causes of compilation errors in Java. UnnaturalCode is focused on locating typos that cause compilation errors. The technique presented in Chapter 3 was a major departure from earlier techniques that used some form of static analysis. By using an n-gram language model from the field of natural language processing, UnnaturalCode had unique strengths and weaknesses when compared to earlier techniques.

Chapter 3 demonstrated the first part of my thesis statement (1.3.1): using an n-gram language model to find typos in source code can provide more

precise reports to the user of the location of those errors. Then, my results in Chapter 4 reinforce that conclusion by duplicating the same technique with a contrasting programming language.

UnnaturalCode in Chapter 3 had a major limitation that was addressed in Chapter 4. The first version required the use of an oracle to first check whether or not code was acceptable. A compiler acted as an oracle. However, accepting or rejecting source code may be undecidable in many languages, such as interpreted languages. I extended the technique to work with the interpreted language Python, in Chapter 4, and eliminate reliance on an oracle. Then, I re-evaluated UnnaturalCode with Python code instead of Java code. Extending and re-evaluating UnnaturalCode with an interpreted programming language presented several challenges that were not present with a compiled programming language like Java. I showed that performance (in terms of MRR) decreased when moving from Java to Python. Additionally, the evaluation in Chapter 4 was extended to include mutations that mimicked actual typographical errors.

Then, in chapter 5, I extended UnnaturalCode to not only locate typos, but also suggest changes to fix them. The n-gram model at the heart of UnnaturalCode was compared with a more sophisticated neural network language model. Additionally, an evaluation was performed with typos written by students rather than relying on mutations to simulate typos. The evaluation was also performed with UnnaturalCode in an off-line setting, with unrelated corpus and testing data sets. The off-line setting dramatically reduced performance (in terms of MRR), but UnnaturalCode was still able to suggest the correct fix for one third of the student generated typos, as long as those typos did not occur within an identifier or literal.

In Chapters 3 and 4, I was able to demonstrate the first part of my thesis statement for a language where an oracle for basic semantics, such as identifier spelling, is available and one where it is not. In Chapter 5 I was able show that n-gram language models can suggest corrections to the user in addition to precisely reporting the locations of typos. In the following chapters, I go

on to generalize the success of UnnaturalCode by demonstrating a second unnatural tool, PartyCrasher, that replaces n-gram representation with TF-IDF representation.

Then, in Chapter 6 and 7 I moved away from typos and demonstrated the second part of my thesis statement: automated crash report bucketing. Chapter 6 presents a novel data set of crash reports. A variety of statistical analyses are applied to the crash reports and the crash reports are characterized in several different ways. The analysis is geared toward using automated, statistics-driven techniques, such as those from the field of (natural-language) information retrieval. A few striking similarities between natural language and crash reports are described. A few additional statistics that may be relevant to software engineering research and practice are also detailed.

PartyCrasher was presented in Chapter 7. PartyCrasher is an application to bucket and explore crash reports. PartyCrasher buckets crash reports with the help of a distributed document search engine. I then evaluated PartyCrasher using the data set from Chapter 6. Despite the search engine being designed to search natural-language documents, PartyCrasher was able to bucket crash reports. Additionally, PartyCrasher's output was more similar to human-curated buckets than simple static, rule-based approaches to clustering crashes. A number of additional benefits were also gained when using an on-line, statistical approach.

Chapter 7 demonstrated the second part of my thesis statement (1.3.1): by using a searching engine, that used TF-IDF representation, PartyCrasher produced buckets that were more similar to buckets curated by developers than other common, simple algorithms for bucketing crashes.

Based on the success of UnnaturalCode and PartyCrasher, there are many directions for future work. Future work could proceed not only by answering questions related to the UnnaturalCode and PartyCrasher, but also in new unnatural tools.

8.2 Future Work

I will divide this section in to three parts, consisting of first, motivating future work on unnatural tools, and then enumerating possibilities for future research in unnatural tools in general, future research specifically inspired by UnnaturalCode, and future research specifically inspired by PartyCrasher.

8.2.1 General Motivations Future Work

Software artifacts, even if they are not written in a natural language, are still symbolic communication. At the very least, they are symbolic communication between two pieces of software, if not between a human and a piece of software like a compiler. Thus in a lot of situations where currently static tools are employed, we can simply pretend the language being spoken is a natural one and immediately gain a path toward addressing that situation with more abstract, and thus, flexible NL/IR techniques. NL/IR techniques are more abstract because they typically will not "understand" the structure of the language being spoken beyond what is conveyed through tokenization and what they are capable of "learning" by processing many examples.

The process of developing software generates huge quantities of information that often goes unused, despite the burgeoning field (and practice) of data mining software repositories. In fact, much of this data does not make it into software repositories at all. Developers often make a small code change and then check how that change impacts the behaviour of their software. This is especially true for students. UnnaturalCode is *not* designed to act on and possibly incorporate information from commits to a software repository. Instead, UnnaturalCode is designed to act on and possibly incorporate information from every single compilation attempt, which can and often do occur multiple times in between commits. For a single project or a single developer this information is not much larger than a repository, yet it is discarded.

Furthermore, each time code is compiled, run, or static analysis of any sort is performed, a wealth of information is inferred by the compiler, interpreter,

or analysis tool. However, usually this information is not recorded anyplace where it can be accessed outside of the compiler, interpreter, or analysis tool. The main exceptions to this are debugging information (debugging symbols) and profiling records. Plenty of other information, such as whether two control statements (such as loops) could trade places, or whether a recursive function can be easily transformed into a loop is never recorded. Yet it may be the case that such information would be valuable to the developer or to a software engineering tool like a refactoring tool.

Indeed, compilers, interpreters, and static analysis tools often completely ignore statistical information that might be employed to make those tools more effective. With the notable exception of optimization, which began to integrate statistical information in 1991 (Chang et al., 1991), software engineering tools are often created with only the information that can be deduced through static analysis of the source code in mind.

While static analysis is the obvious choice for reasoning about the computational or spatial complexity of code, software engineering brings us the additional idea of measuring how *complicated* a piece of software is, or how difficult it is for a human to understand a piece of software and reason about it. This is a seperate concept from software *complexity* that describes how many interactions different pieces of code have with each other. While complexity is used to get an idea of how complicated a piece of code is, it is not necessarily measuring the same thing.

It would be valuable to get an idea of how complicated a piece of code is by using language modelling or information retrieval to estimate the effect of familiarity and fluency (the *mere exposure effect*) with the code. Motivation for this has been explored by Casalnuovo et al. (2019). For example, even though a particular piece of code may be highly complex, such that it has many interactions within itself and with other pieces of code, it may also adhere to a common pattern, such that the developer is has had experience with similar pieces of code, making it easier for them to understand.

8.2.2 Questions for Future Work

8.2.2.1 General Questions

The results described in this thesis have encouraging implications for research that does not focus on n-gram models, typos, search engines, or crash reports. Future work on unnatural tools could, and should, cover a wide variety of questions, such as:

- Given a parallel corpus of program code and corresponding unit test code, could a natural-language translation algorithm be trained "translate" new program code into corresponding unit tests? Such a parallel corpus could be obtained automatically by test suite coverage measurement tools.

- Could a natural language translation technique be applied to add additional information to documentation produced by documentation generation tools, which currently rely on static analysis?

- Linters typically rely on static analysis. Could they be improved by including a language model or search engine to produce warnings and suggestions that developers would be more likely to agree with during code review?

- Linters are also often tasked with checking whether code conforms to an organizations stylistic conventions. Could a linter model how an organization uses whitespace? Could a linter then use that model to flag code that does not conform to organizational conventions?

- Does the size of the vocabulary used in a library API (after suitable tokenization) correlate with the difficulty using that particular library?

- Can a natural language model be used to help identify features or syntax of a programming language that are not used often enough to justify their maintenance?

- By generating an n-gram model (or any relevant Markov model) of the procedure calls between two different layers of software, such as an application and a framework, can problematic call sequences be detected?
- Can stochastic language models, especially those that were developed to deal with natural language, be used to infer the correct contents of missing sections of abstract syntax trees?
- The field of Search-Based Software Engineering has had great success repairing bugs using metaheuristic search, as described in the yearly proceedings of the International Symposium on Search-Based Software Engineering (Nejati and Gay, 2019). Could repairs be improved if metaheuristics were combined with heuristics derived from likelihood estimation based on stochastic models built on human-written code?

8.2.2.2 Typos

Even considering the narrower scope of typo correction, there is research left to be done.

- Are mutations a good approximation of human-made typos?
- How do typos made by novice programmers differ from typos made by expert programmers?
- Is there an effective way to integrate a tool like UnnaturalCode into an integrated development environment (IDE)?
- Can UnnaturalCode, or a similar tool, effectively suggest repairs to code that compiles but fails to pass tests?
- Would UnnaturalCode work as an immediate "autocorrect" tool that fixes typos in real time as soon as they are typed?
- Can UnnaturalCode be extended to fix multiple typos at the same time?

- Can UnnaturalCode be improved by modelling both known-good and known-bad code?
- Can UnnaturalCode be improved by using more advanced graphical models in place of n-grams?

8.2.2.3 Crashes

Just like with typos, there are plenty of research opportunities to explore what useful information and tools can be built around crash reports.

- Can crashes be analyzed statistically to automatically identify and diagnose hardware malfunctions?
- Can crashes caused by cosmic rays or other external influences be isolated automatically?
- Can crashes be automatically separated into those caused by bugs in the software itself, and those caused by library and operating system bugs?
- Can PartyCrasher be combined with UnnaturalCode to automatically locate the cause, in the code, of the crashes?
- Can the human component be further removed from crash analysis to the point where useful bug reports can be filed automatically for new trends in the crash reports as the appear?
- Is PartyCrasher, or a similar tool, effective in bucketing non-fatal errors?

8.3 Remarks

Software-engineering tools can be developed with the option of employing on-line, off-the-shelf, statistical techniques for natural language in mind.

Apart from the above statement, which I demonstrated in this thesis, the following comments are my thoughts on this work and do not have hard evidence for them. However, based on my experience, I believe them to be true.

8.3.1 Thoughts

I do not claim that alternative approaches similar to the ones presented here will always perform comparably or better than static analysis and rule-based techniques. However, I believe the results presented in this thesis, as well as in the prior work show that similar techniques deserve consideration.

However, by applying techniques already proven in the fields of natural-language processing and (natural language) information retrieval, I have created viable software engineering tools in spaces where similar tools have traditionally not employed statistics of any kind. UnnaturalCode and PartyCrasher both solve real-world problems that face stakeholders at different stages of the software engineering process.

While both PartyCrasher and UnnaturalCode are preceded by tools with the same goals, this does not necessarily have to be the case. There is no reason that unnatural tools cannot solve software engineering challenges for which no tool currently exists. Unnatural tools can also be combined with tools that currently rely on static analysis.

Chapters 3 through 7 show that unnatural tools can be provided to assist software engineers at time-consuming points in the software development process. While how much time they might save engineers is not empirically known, they can provide engineers with useful information at wildly different stages in the software development process. This naturally leads to the possibility that unnatural tools may be able to assist developers with other tasks.

My experience with UnnaturalCode and PartyCrasher has also taught me that the exact tokenization method used in an unnatural tool is critical to the tool's development and eventual success. During the development of UnnaturalCode, several different lexical analyzers were used to tokenize Java source code because they were readily available. However, all of the lexical analyzers plagued development with problems. Eventually, I resorted to isolating the lexical analyzer inside Java 8 compiler, that is written in Java, extracting the analyzer from the compiler, and wrapping it so that it could be called from UnnaturalCode, which is written in Python. When tokenizing source code, a

lexical analyzer that was "close enough" was never actually close enough. This was especially true when attempting scientific evaluations.

In contrast, the tokenization scheme that was found to be the most effective for PartyCrasher, as shown in Figure 7.4, was the "CamelCase" tokenizer. The CamelCase tokenizer is not specific to any specific programming language. Additionally, the CamelCase tokenizer attempts to split identifiers that consist of multiple natural-language words into a seperate token for each word (Figure 7.1).

My experience suggests that tasks that are not usually a focus of tooling or a focus of software engineering research in general, the tasks that are generally glossed over, will benefit the most from unnatural tools. This was exactly the case for fixing typos in code before I introduced UnnaturalCode in 2014.

References

I. Ahmed, N. Mohan, and C. Jensen (2014). "The Impact of Automatic Crash Reports on Bug Triaging and Development in Mozilla". In: *Proceedings of The International Symposium on Open Collaboration*. OpenSym '14. New York, NY, USA: ACM, 1:1–1:8. ISBN: 978-1-4503-3016-9. DOI: https://doi.org/10.1145/2641580.264158510.1145/2641580.2641585. URL: `http://doi.acm.org/10.1145/2641580.2641585` (visited on 01/08/2016) (cit. on pp. 168, 173).

U. Z. Ahmed, R. Sindhgatta, N. Srivastava, and A. Karkare (Nov. 2019). "Targeted Example Generation for Compilation Errors". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 327–338. DOI: https://doi.org/10.1109/ASE.2019.0003910.1109/ASE.2019.00039 (cit. on p. 94).

Airbrake (2020). *Crash Reporting Software*. URL: `https://airbrake.io/features/crash-reporting-software` (visited on 09/30/2010) (cit. on p. 5).

M. Allamanis and C. Sutton (2013). "Mining source code repositories at massive scale using language modeling". In: *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, pp. 207–216 (cit. on p. 50).

A. Altadmri and N. C. C. Brown (2015). "37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, pp. 522–527 (cit. on p. 97).

E. Amigó, J. Gonzalo, J. Artiles, and F. Verdejo (2009). "A comparison of extrinsic clustering evaluation metrics based on formal constraints". In: *Information retrieval* 12.4, pp. 461–486. URL: `http://link.springer.com.login.ezproxy.library.ualberta.ca/article/10.1007/s10791-008-9066-8` (visited on 01/26/2016) (cit. on pp. 185, 186, 190).

D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis (2007). "RPython: a step towards reconciling dynamically and statically typed OO languages".

In: *Proceedings of the 2007 symposium on Dynamic languages*. ACM, pp. 53–64 (cit. on p. 63).

K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle (2008). "Finding Similar Failures Using Callstack Similarity." In: *SysML*. URL: https://www.usenix.org/event/sysml08/tech/full_papers/bartz/bartz_html/ (visited on 01/19/2016) (cit. on pp. 120, 127, 173, 174, 177).

K. Beck (2003). *Test-driven development : by example*. Boston: Addison-Wesley. ISBN: 978-0-321-14653-3 (cit. on p. 65).

B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera, J. L. Pearce, and J. Prather (2019). "Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research". In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '19. Aberdeen, Scotland Uk: Association for Computing Machinery, pp. 177–210. ISBN: 9781450375672. DOI: https://doi.org/10.1145/3344429.337250810.1145/3344429.3372508. URL: https://doi.org/10.1145/3344429.3372508 (cit. on p. 8).

T. Bell, P. Andreae, and A. Robins (2012). "Computer Science in NZ High Schools: The First Year of the New Standards". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. Raleigh, North Carolina, USA: ACM, pp. 343–348. ISBN: 978-1-4503-1098-7. DOI: https://doi.org/10.1145/2157136.215724010.1145/2157136.2157240. URL: http://doi.acm.org/10.1145/2157136.2157240 (cit. on p. 58).

M. Brodie, S. Ma, G. Lohman, L. Mignet, M. Wilding, J. Champlin, and P. Sohn (June 2005a). "Quickly Finding Known Software Problems via Automated Symptom Matching". In: *Second International Conference on Autonomic Computing, 2005. ICAC 2005. Proceedings*. Second International Conference on Autonomic Computing, 2005. ICAC 2005. Proceedings, pp. 101–110. DOI: https://doi.org/10.1109/ICAC.2005.4910.1109/ICAC.2005.49 (cit. on pp. 127, 128, 173–175, 177, 200, 201).

M. Brodie, S. Ma, L. Rachevsky, and J. Champlin (2005b). "Automated problem determination using call-stack matching". In: *Journal of Network and Systems Management* 13.2, pp. 219–237. URL: http://link.springer.com/article/10.1007/s10922-005-4443-8 (visited on 01/19/2016) (cit. on p. 127).

N. C. C. Brown, A. Altadmri, S. Sentance, and M. Kölling (2018). "Blackbox, Five Years On: An Evaluation of a Large-Scale Programming Data Collection Project". In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER '18. Espoo, Finland: Association for Computing Machinery, pp. 196–204.

ISBN: 9781450356282.
DOI: https://doi.org/10.1145/3230977.323099110.1145/3230977.3230991.
URL: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3230977.3230991 (cit. on p. 97).

N. C. C. Brown and A. Altadmri (2014). "Investigating novice programming mistakes: Educator beliefs vs. student data". In: *Proceedings of the tenth annual conference on International computing education research*. ACM, pp. 43–50 (cit. on pp. 96, 97).

N. C. C. Brown, M. Kölling, D. McCall, and I. Utting (2014). "Blackbox: a large scale repository of novice programmers' activity". In: *Proceedings of the 45th ACM technical symposium on Computer Science Education*. ACM, pp. 223–228 (cit. on pp. 97, 104).

BugSplat (2020). *Learn About BugSplat's Features*.
URL: https://www.bugsplat.com/features/ (visited on 09/30/2010) (cit. on p. 5).

M. G. Burke and G. A. Fisher (Mar. 1987). "A practical method for LR and LL syntactic error diagnosis and recovery". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.2, pp. 164–197 (cit. on pp. 30, 32, 62).

J. C. Campbell, E. A. Santos, and A. Hindle (May 2016a). "The Unreasonable Effectiveness of Traditional Information Retrieval in Crash Report Deduplication". In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. Los Alamitos, CA, USA: IEEE Computer Society, pp. 269–280.
DOI: https://doi.org/10.1109/MSR.2016.03510.1109/MSR.2016.035.
URL: https://doi.ieeecomputersociety.org/10.1109/MSR.2016.035 (cit. on pp. v, 120, 127, 128, 130, 138, 146, 154).

J. C. Campbell, A. Hindle, and J. N. Amaral (2015). *Error location in Python: where the mutants hide*. Tech. rep. PeerJ PrePrints (cit. on pp. iv, 56).

J. C. Campbell, E. A. Santos, and A. Hindle (Nov. 2016b). "Anatomy of a crash repository". In: *PeerJ Preprints* 4, e2601v1. ISSN: 2167-9843. DOI: https://doi.org/10.7287/peerj.preprints.2601v110.7287/peerj.preprints.2601v1.
URL: https://doi.org/10.7287/peerj.preprints.2601v1 (cit. on pp. v, 119).

J. C. Campbell, A. Hindle, and J. N. Amaral (2014). "Syntax errors just aren't natural: improving error reporting with language models". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, pp. 252–261 (cit. on pp. iv, 23, 58–60, 70, 73, 76, 78, 80, 87, 88, 92).

B. Cannon (2005). "LOCALIZED TYPE INFERENCE OF ATOMIC TYPES IN PYTHON". MA thesis. California Polytechnic State University (cit. on pp. 63, 87).

Canonical Ltd. (2004). *Launchpad*.
URL: `https://launchpad.net/` (visited on 01/20/2016)
(cit. on pp. 120, 180, 188).

C. Casalnuovo, K. Lee, H. Wang, P. Devanbu, and E. Morgan (2019).
"Do People Prefer" Natural" code?" In: *arXiv preprint arXiv:1910.03704*
(cit. on p. 208).

P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu (1991).
"Using profile information to assist classic code optimizations".
In: *Software: Practice and Experience* 21.12, pp. 1301–1321
(cit. on p. 208).

Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus (2019). "SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair".
In: *IEEE Transactions on Software Engineering*. ISSN: 1939-3520. DOI: https://doi.org/10.1109/TSE.2019.294017910.1109/TSE.2019.2940179
(cit. on p. 94).

R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro (Nov. 2002).
"Repairing syntax errors in LR parsers".
In: *ACM Trans. Program. Lang. Syst.* 24.6, pp. 698–710. ISSN: 0164-0925.
DOI: https://doi.org/10.1145/586088.58609210.1145/586088.586092
(cit. on pp. 26, 30, 33, 54, 63, 86).

Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel (2012a).
"ReBucket: a method for clustering duplicate crash reports based on call stack similarity". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, pp. 1084–1093. URL:
`http://dl.acm.org/citation.cfm?id=2337364` (visited on 01/08/2016)
(cit. on pp. 120, 128, 173, 178, 186, 200).

Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel (2012b).
"Rebucket: A method for clustering duplicate crash reports based on call stack similarity".
In: *2012 34th International Conference on Software Engineering (ICSE)*.
IEEE, pp. 1084–1093 (cit. on p. 167).

M. L. Delignette-Muller, C. Dutang, et al. (2015).
"fitdistrplus: An R package for fitting distributions".
In: *Journal of Statistical Software* 64.4, pp. 1–34 (cit. on p. 141).

A. Derezinska and K. Hałas (2015).
"Improving mutation testing process of python programs".
In: *Software Engineering in Intelligent Systems*. Springer, pp. 233–242
(cit. on p. 57).

A. Derezińska and K. Hałas (2014).
"Operators for Mutation Testing of Python Programs".
In: *ICS Research Report* (cit. on pp. 66, 82).

T. Dhaliwal, F. Khomh, and Y. Zou (Sept. 2011). "Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011

27th IEEE International Conference on Software Maintenance (ICSM), pp. 333–342. DOI: https://doi.org/10.1109/ICSM.2011.608080010.1109/ICSM.2011.6080800 (cit. on pp. 120, 127, 173, 177, 178, 200).

Elasticsearch BV (2016). *Elasticsearch*. URL: https://www.elastic.co/products/elasticsearch (visited on 01/20/2016) (cit. on pp. 173–175, 184).

N. Eldredge (Oct. 2008). *Re: When is SIGTRAP raised?* URL: http://linux.derkeiler.com/newsgroups/comp.os.linux.development.apps/2008-10/msg00107.html (cit. on p. 145).

S. Evert and M. Baroni (2007). "zipfR: Word frequency distributions in R". In: *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics, pp. 29–32 (cit. on pp. 135, 138, 159).

C. Ford (Apr. 13, 2008). *mono crashed*. Accessed: 2016-10-29. URL: https://bugs.launchpad.net/ubuntu/+source/mono/+bug/217012 (visited on 04/13/2008) (cit. on p. 150).

A. Ganapathi, V. Ganapathi, and D. A. Patterson (2006). "Windows XP Kernel Crash Analysis." In: *LISA*. Vol. 6, pp. 49–159. (Visited on 08/10/2016) (cit. on pp. 126, 127).

A. Ganapathi and D. A. Patterson (2005). "Crash Data Collection: A Windows Case Study." In: *DSN*. Vol. 5. Citeseer, pp. 280–285. (Visited on 08/10/2016) (cit. on pp. 126, 127).

S. Garner, P. Haden, and A. Robins (2005). "My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems". In: *Proceedings of the 7th Australasian conference on Computing education-Volume 42*. Australian Computer Society, Inc., pp. 173–180 (cit. on pp. 25, 26, 32, 58, 61).

K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt (2009). "Debugging in the (Very) Large: Ten Years of Implementation and Experience". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. New York, NY, USA: ACM, pp. 103–116. ISBN: 978-1-60558-752-3. DOI: https://doi.org/10.1145/1629575.162958610.1145/1629575.1629586. URL: http://doi.acm.org/10.1145/1629575.1629586 (visited on 01/15/2016) (cit. on pp. 120, 158, 168, 169, 173, 174, 177, 200).

M. Gómez, R. Rouvoy, and L. Seinturier (2015). *Reproducing Context-Sensitive Crashes in Mobile Apps Using Crowdsourced Debugging*. Report. Inria Lille ; INRIA. URL: https://hal.inria.fr/hal-01155597/document (visited on 08/22/2016) (cit. on p. 126).

Google Inc. (2016). *Firebase | App Success Made Simple*.
URL: https://firebase.google.com/ (visited on 10/26/2016)
(cit. on p. 120).

C. Gormley (May 2015). *Elasticsearch Reference: Pattern Analyzer*. 1.6.
Available: https://github.com/elastic/elasticsearch/blob/1.6/docs/reference/analysis/analyzers/pattern-analyzer.asciidoc
(cit. on p. 154).

J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley (Feb. 2015).
"Lexical Structure". In: *The Java Language Specification*. 8th ed.
Available:
https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html.
Oracle Corporation. Chap. 3 (cit. on p. 98).

S. L. Graham, C. B. Haley, and W. N. Joy (Aug. 1979).
"Practical LR error recovery". In: *SIGPLAN Not.* 14.8, pp. 168–175.
ISSN: 0362-1340.
DOI: https://doi.org/10.1145/872732.80696710.1145/872732.806967
(cit. on pp. 30, 32, 62).

V. Grishchenko (2006).
Plot of word frequency in Wikipedia-dump 2006-11-27.
[Online; accessed 17-August-2016].
URL: https://commons.wikimedia.org/w/index.php?title=File:Wikipedia-n-zipf.png&oldid=184312342 (cit. on p. 135).

L. Guerrouj, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc (2013).
"Tidier: an identifier splitting approach using speech recognition techniques".
In: *Journal of Software: Evolution and Process* 25.6, pp. 575–599
(cit. on p. 201).

J. N. Hall, J. A. McAdams, et al. (2010).
Effective Perl Programming: Ways to Write Better, More Idiomatic Perl.
Pearson Education (cit. on p. 83).

S. Hanenberg (2010).
"An Experiment about Static and Dynamic Type Systems: Doubts about the Positive Impact of Static Type Systems on Development Time".
In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10.
Reno/Tahoe, Nevada, USA: Association for Computing Machinery,
pp. 22–35. ISBN: 9781450302036.
DOI: https://doi.org/10.1145/1869459.186946210.1145/1869459.1869462.
URL: https://doi.org/10.1145/1869459.1869462 (cit. on p. 84).

A. E. Hassan, R. C. Holt, and A. Mockus, eds. (2004).
MSR 2004: Proceedings of the 11th Working Conference on Mining Software Repositories. Edinburgh, Scotland, UK. URL: http://2004.msrconf.org/MSR2004ProceedingsFINAL_IEE_Acrobat4.pdf
(cit. on p. 19).

B. J. Heeren (2005). “Top Quality Type Error Messages”.
PhD thesis. Nederlands: Universiteit Utrecht (cit. on pp. 26, 30, 33, 63).

I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles (2007).
“Towards a theoretical model for software growth”. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*.
IEEE Computer Society, p. 21 (cit. on p. 126).

E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker (2014).
“An empirical study of identifier splitting techniques”.
In: *Empirical Software Engineering* 19.6, pp. 1754–1780 (cit. on p. 201).

A. Hindle (Mar. 2019). “Complexity: Let’s Not Make This Complicated”.
In: *IEEE Software* 36.2, pp. 130–132. ISSN: 1937-4194.
DOI: https://doi.org/10.1109/MS.2018.288387510.1109/MS.2018.2883875
(cit. on p. 119).

A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu (June 2012).
“On the naturalness of software”.
In: *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 837–847. DOI:
https://doi.org/10.1109/ICSE.2012.622713510.1109/ICSE.2012.6227135
(cit. on pp. 11, 22, 26, 30, 31, 35, 54, 60).

A. Holkner and J. Harland (2009).
“Evaluating the Dynamic Behaviour of Python Applications”.
In: *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*. ACSC ’09.
Wellington, New Zealand: Australian Computer Society, Inc., pp. 19–28.
ISBN: 978-1-920682-72-9.
URL: `http://dl.acm.org/citation.cfm?id=1862659.1862665`
(cit. on p. 82).

M. Hristova, A. Misra, M. Rutter, and R. Mercuri (2003).
“Identifying and correcting Java programming errors for introductory computer science students”. In: *ACM SIGCSE Bulletin* 35.1, pp. 153–156
(cit. on pp. 26, 30, 33, 64).

B. Hsu and J. Glass (2008). “Iterative language model estimation: efficient data structure & algorithms”. In: (cit. on pp. 29, 61).

A. A. Hwang, I. A. Stefanovici, and B. Schroeder (2012).
“Cosmic rays don’t strike twice: understanding the nature of DRAM errors and the implications for system design”.
In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM, pp. 111–122
(cit. on p. 130).

J. Jackson, M. Cobb, and C. Carver (2005).
“Identifying top Java errors for novice programmers”. In: *Frontiers in Education, 2005. FIE’05. Proceedings 35th Annual Conference*. IEEE, T4C–T4C (cit. on pp. 25, 32, 58, 61).

M. C. Jadud (2005).
“A first look at novice compilation behaviour using BlueJ”.

In: *Computer Science Education* 15.1, pp. 25–40
(cit. on pp. 25, 32, 58, 61, 62).

M. C. Jadud (2006).
"Methods and tools for exploring novice compilation behaviour".
In: *Proceedings of the second international workshop on Computing education research*. ACM, pp. 73–84 (cit. on pp. 25, 32, 58, 61, 62).

F. James (2000). *Modified Kneser-Ney Smoothing of n-Gram Models*.
Tech. rep. (cit. on p. 99).

T. Jim (2012). *Python is not context free*.
`http://trevorjim.com/python-is-not-context-free/`. blog
(cit. on p. 85).

M. Kechagia and D. Spinellis (2014).
"Undocumented and Unchecked: Exceptions That Spell Trouble".
In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. New York, NY, USA: ACM, pp. 312–315.
ISBN: 978-1-4503-2863-0. (Visited on 08/10/2016) (cit. on pp. 126, 145).

J. Kehne (Jan. 30, 2013). *umbrello crashed with SIGSEGV in*.
Accessed: 2016-10-29. URL:
`https://bugs.launchpad.net/ubuntu/+source/kdesdk/+bug/1110251`
(visited on 01/30/2013) (cit. on p. 156).

F. Khomh, B. Chan, Y. Zou, and A. Hassan (Oct. 2011).
"An Entropy Evaluation Approach for Triaging Field Crashes: A Case Study of Mozilla Firefox".
In: *2011 18th Working Conference on Reverse Engineering (WCRE)*.
2011 18th Working Conference on Reverse Engineering (WCRE), pp. 261–270.
DOI: https://doi.org/10.1109/WCRE.2011.3910.1109/WCRE.2011.39
(cit. on pp. 120, 200, 201).

I.-S. Kim and K.-M. Choe (July 2001).
"Error repair with validation in LR-based parsing".
In: *ACM Trans. Program. Lang. Syst.* 23.4, pp. 451–471. ISSN: 0164-0925.
DOI: https://doi.org/10.1145/504083.50408410.1145/504083.504084
(cit. on pp. 30, 33, 54, 63, 86).

S. Kim, T. Zimmermann, and N. Nagappan (June 2011). "Crash graphs: An aggregated view of multiple crashes to improve crash triage".
In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), pp. 486–493. DOI:
https://doi.org/10.1109/DSN.2011.595826110.1109/DSN.2011.5958261
(cit. on pp. 120, 127, 176, 178, 200).

S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik (2012).
"Do static type systems improve the maintainability of software systems? An empirical study". In: *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, pp. 153–162 (cit. on p. 84).

M. Kölling and contributors (June 2016). *BlueJ*. (Accessed on 08/10/2017) (cit. on p. 97).

M. Kölling, B. Quig, A. Patterson, and J. Rosenberg (2003). "The BlueJ System and its Pedagogy". In: *Computer Science Education* 13.4, pp. 249–268. DOI: https://doi.org/10.1076/csed.13.4.249.1749610.1076/csed.13.4.249.17496. eprint: `http://www.tandfonline.com/doi/pdf/10.1076/csed.13.4.249.17496`. URL: `http://www.tandfonline.com/doi/abs/10.1076/csed.13.4.249.17496` (cit. on pp. 97, 104).

D. Krpan and I. Bilobrk (2011). "Introductory programming languages in higher education". In: *MIPRO, 2011 Proceedings of the 34th International Convention*. IEEE, pp. 1331–1336 (cit. on pp. 58, 59).

S. K. Kummerfeld and J. Kay (2003). "The neglected battle fields of syntax errors". In: *Proceedings of the fifth Australasian conference on Computing education-Volume 20*. Australian Computer Society, Inc., pp. 105–111 (cit. on pp. 25, 32, 62).

Launchpad contributors (2016). *bug—Launchpad web service API documentation*. Accessed: 2016-10-30. URL: `https://launchpad.net/+apidoc/1.0.html#bug` (visited on 10/20/2016) (cit. on p. 138).

D. Lawrie, H. Feild, and D. Binkley (2007a). "Quantifying identifier quality: an analysis of trends". In: *Empirical Software Engineering* 12.4, pp. 359–388 (cit. on p. 154).

D. Lawrie, C. Morrell, H. Feild, and D. Binkley (2007b). "Effective identifier names for comprehension and memory". In: *Innovations in Systems and Software Engineering* 3.4, pp. 303–318 (cit. on p. 154).

J. Lerch and M. Mezini (Mar. 2013). "Finding Duplicates of Your Yet Unwritten Bug Report". In: *2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*. 2013 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 69–78. DOI: https://doi.org/10.1109/CSMR.2013.1710.1109/CSMR.2013.17 (cit. on pp. 128, 169, 170, 173, 174, 179, 183, 184, 188, 191, 195, 199, 200).

B. S. Lerner, M. Flower, D. Grossman, and C. Chambers (2007). "Searching for type-error messages". In: *Conference on Programming Language Design and Implementation (PLDI)*. San Diego, CA, USA, pp. 425–434 (cit. on pp. 30, 33, 64).

B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan (2005). "Scalable statistical bug isolation". In: *ACM SIGPLAN Notices*. Vol. 40. ACM, pp. 15–26. URL: `http://dl.acm.org/citation.cfm?id=1065014` (visited on 01/08/2016) (cit. on pp. 121, 176).

Linux man-pages project (Aug. 2014). *mmap(2) Linux User's Manual*. 3.74. The Linux Foundation (cit. on p. 146).

C. Liu and J. Han (2006). "Failure Proximity: A Fault Localization-based Approach". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, pp. 46–56. ISBN: 1-59593-468-5. DOI: https://doi.org/10.1145/1181775.118178210.1145/1181775.1181782. URL: http://doi.acm.org/10.1145/1181775.1181782 (visited on 01/19/2016) (cit. on pp. 173, 176, 200).

C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff (2005). "SOBER: Statistical Model-based Bug Localization". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. New York, NY, USA: ACM, pp. 286–295. ISBN: 1-59593-014-0. DOI: https://doi.org/10.1145/1081706.108175310.1145/1081706.1081753. URL: http://doi.acm.org/10.1145/1081706.1081753 (visited on 01/20/2016) (cit. on p. 176).

B. Marr (Mar. 2015). *Why only one of the 5 Vs of big data really matters*. http://www.ibmbigdatahub.com/blog/why-only-one-5-vs-big-data-really-matters (cit. on p. 170).

J. Matthew, P. Fragoso, J. Livingston, R. P. Skadberg, B. Nocera, C. Fergeau, and W. J. McCann (Apr. 14, 2009). *rb-mtp-source.c*. Accessed: 2016-09-02. URL: https://github.com/ivankelly/Rhythmbox-Spotify-Plugin/blob/2947d775d494848bbf6fbfd5a91004cebf159926/plugins/mtpdevice/rb-mtp-source.c#L217 (visited on 04/14/2009) (cit. on p. 153).

C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik (2012). "An empirical study of the influence of static type systems on the usability of undocumented software". In: *ACM SIGPLAN Notices*. Vol. 47. 10. ACM, pp. 683–702 (cit. on p. 84).

W. J. McCann, J. Livingston, R. P. Skadberg, B. Nocera, C. Fergeau, and J. Matthew (Nov. 13, 2008). *rb-removable-media-source.c*. Accessed: 2016-10-29. URL: https://github.com/ivankelly/Rhythmbox-Spotify-Plugin/blob/a89630dba55eab0baa5a58c9b2cfc01e1bef4ef2/sources/rb-removable-media-source.c#L657 (visited on 11/13/2008) (cit. on p. 153).

L. McIver (2000). "The effect of programming language on error rates of novice programmers". In: *12th Annual Workshop of the Psychology of Programming Interest Group*. Citeseer, pp. 181–192 (cit. on pp. 25, 26, 32, 61).

K. J. Millman and M. Aivazis (2011). “Python for Scientists and Engineers”. In: *Computing in Science & Engineering* 13.2, pp. 9–12 (cit. on p. 58).

N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet (Apr. 2007). “Automatically Identifying Known Software Problems”. In: *2007 IEEE 23rd International Conference on Data Engineering Workshop*. 2007 IEEE 23rd International Conference on Data Engineering Workshop, pp. 433–441. DOI: https://doi.org/10.1109/ICDEW.2007.440102610.1109/ICDEW.2007.4401026 (cit. on pp. 120, 127, 128, 158, 173–177, 200).

M. Monperrus (July 2020). “The living review on automated program repair”. In: URL: `https://www.monperrus.net/martin/repair-living-review.pdf` (visited on 07/13/2020) (cit. on p. 94).

I. R. Moore (2000–2013). *Jester: the JUnit test tester*. `http://jester.sourceforge.net/` (cit. on pp. 66, 67, 82, 88).

A. Moroo, A. Aizawa, and T. Hamamoto (2017). “Reranking-based Crash Report Deduplication.” In: *SEKE*, pp. 507–510 (cit. on p. 167).

Mozilla Corporation (2012). *Mozilla Crash Reports*. URL: `http://crash-stats.mozilla.com` (cit. on pp. 120, 163, 168).

mozilla/socorro: Socorro is a server to accept and process Breakpad crash reports. (2016). URL: `https://github.com/mozilla/socorro` (visited on 01/24/2016) (cit. on p. 175).

F. Murtagh and P. Contreras (2012). “Algorithms for hierarchical clustering: an overview”. In: *WIREs Data Mining and Knowledge Discovery* 2.1, pp. 86–97. DOI: https://doi.org/https://doi.org/10.1002/widm.53https://doi.org/10.1002/widm.53. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.53`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.53` (cit. on p. 183).

Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31–September 1, 2019, Proceedings (2019) 11664 (cit. on p. 210).

Options That Control Optimization—GCC online documentation (Oct. 2016). 6.2. Available: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-foptimize-sibling-calls-754`. Free Software Foundation, Inc. (cit. on p. 125).

T. J. Parr and R. W. Quong (1995). “ANTLR: A predicated-LL(k) parser generator”. In: *Software: Practice and Experience* 25.7, pp. 789–810. ISSN: 1097-024X. DOI: https://doi.org/10.1002/spe.438025070510.1002/spe.4380250705. URL: `http://dx.doi.org/10.1002/spe.4380250705` (cit. on p. 85).

T. Parr and K. Fisher (2011). “LL(*): The Foundation of the ANTLR Parser Generator”.

In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11.
San Jose, California, USA: ACM, pp. 425–436. ISBN: 978-1-4503-0663-8.
DOI: https://doi.org/10.1145/1993498.199354810.1145/1993498.1993548.
URL: http://doi.acm.org/10.1145/1993498.1993548 (cit. on p. 63).

F. Pérez, B. E. Granger, and J. D. Hunter (2011).
"Python: an ecosystem for scientific computing".
In: *Computing in Science & Engineering* 13.2, pp. 13–21 (cit. on p. 58).

Piping core dumps to a program—core(5) Linux User's Manual (Dec. 2015).
4.4. The Linux Foundation (cit. on p. 129).

J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi (Oct. 2013).
"Python: The Full Monty". In: *SIGPLAN Not.* 48.10, pp. 217–232.
ISSN: 0362-1340.
DOI: https://doi.org/10.1145/2544173.250953610.1145/2544173.2509536.
URL: http://doi.acm.org/10.1145/2544173.2509536 (cit. on p. 83).

Y. Qi, S. Marchi, M. Kościelnicki, and J. Brobecker (2014). *gdb/tracefile.c*.
Accessed: 2016-10-28.
URL: https://chromium.googlesource.com/native_client/nacl-binutils/+/upstream/master/gdb/tracefile.c#65 (visited on 02/10/2014) (cit. on p. 150).

Raygun (2020). *Error Tracking & Crash Reporting*.
URL: https://raygun.com/platform/crash-reporting (visited on 09/30/2010) (cit. on p. 5).

A. Rigo (2004). "Representation-based just-in-time specialization and the psyco prototype for python".
In: *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. ACM, pp. 15–26 (cit. on p. 63).

A. Rigo and S. Pedroni (2006).
"PyPy's Approach to Virtual Machine Construction".
In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*.
OOPSLA '06. Portland, Oregon, USA: ACM, pp. 944–953.
ISBN: 1-59593-491-X.
DOI: https://doi.org/10.1145/1176617.117675310.1145/1176617.1176753.
URL: http://doi.acm.org/10.1145/1176617.1176753 (cit. on p. 63).

A. Rigo and C. Tismer (2001). *Psyco, the Python specializing compiler*,
http://psyco.sourceforge.net/ (cit. on p. 63).

G. Robles, J. M. González-Barahona, D. Izquierdo-Cortazar, and I. Herraiz (2011). "Tools and Datasets for Mining Libre Software Repositories".
In: *Multi-Disciplinary Advancement in Open Source Software and Processes*, p. 24 (cit. on pp. 130, 180).

Rollbar (2020). *Rollbar - Error Tracking Software for JavaScript, PHP, Ruby, Python and more*.

URL: https://rollbar.com/product/triage/ (visited on 09/30/2010) (cit. on p. 5).

M. Salib (2004). "Faster than C: Static type inference with Starkiller". In: *in PyCon Proceedings, Washington DC*. SpringerVerlag, pp. 2–26 (cit. on p. 63).

G. Salton and M. J. McGill (1983). *Introduction to modern information retrieval*. McGraw-Hill computer science series. New York: McGraw-Hill, p. 63. 448 pp. ISBN: 978-0-07-054484-0 (cit. on p. 173).

E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral (Mar. 2018). "Syntax and sensibility: Using language models to detect and correct syntax errors". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 311–322. DOI: https://doi.org/10.1109/SANER.2018.833021910.1109/SANER.2018.8330219 (cit. on pp. 20, 93–96, 98, 99, 105–107, 109, 110, 112, 115).

E. A. Santos (2018). "Applications of the Naturalness of Software". MA thesis. University of Alberta. DOI: https://doi.org/10.7939/R3W37MB7X10.7939/R3W37MB7X. URL: https://era.library.ualberta.ca/items/2d3cf508-cc17-4534-bb9c-1a0fd7e920c4 (cit. on p. 96).

M. Schaaf (2012). *rhythmbox crashed with SIGSEGV*. URL: https://bugs.launchpad.net/ubuntu/+source/rhythmbox/+bug/806306 (cit. on p. 123).

A. Schröter, N. Bettenburg, and R. Premraj (May 2010). "Do stack traces help developers fix bugs?" In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*. 2010 7th IEEE Working Conference on Mining Software Repositories (MSR), pp. 118–121. DOI: https://doi.org/10.1109/MSR.2010.546328010.1109/MSR.2010.5463280 (cit. on pp. 168, 174, 177, 200).

R. Sedgewick and K. Wayne (2011). *Algorithms, Fourth Edition.* Addison-Wesley Professional. ISBN: 0132762560. URL: https://login.ezproxy.library.ualberta.ca/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat03710a&AN=alb.9389116&site=eds-live&scope=site (cit. on p. 183).

H. Seo and S. Kim (2012a). "Predicting Recurring Crash Stacks". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. New York, NY, USA: ACM, pp. 180–189. ISBN: 978-1-4503-1204-2. DOI: https://doi.org/10.1145/2351676.235170210.1145/2351676.2351702. URL: http://doi.acm.org/10.1145/2351676.2351702 (visited on 01/08/2016) (cit. on pp. 120, 128, 168, 200).

H. Seo and S. Kim (2012b). “Predicting recurring crash stacks”. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, pp. 180–189 (cit. on p. 8).

B. Sigurd, M. Eeg-Olofsson, and J. Van Weijer (2004). “Word length, sentence length and frequency–Zipf revisited”. In: *Studia Linguistica* 58.1, pp. 37–52 (cit. on p. 154).

S. Sippu and E. Soisalon-Soininen (1990). *Parsing Theory: LR(k) and LL(k) Parsing*. EATCS monographs on theoretical computer sciences: European Association for Theoretical Computer Science. Springer. ISBN: 9783540517320. URL: http://books.google.ca/books?id=en4Qk6T9PrAC (cit. on p. 87).

F. Stajano (2000). “Python in education: Raising a generation of native speakers”. In: *Proceedings of 8 th International Python Conference*, pp. 2000–01 (cit. on p. 59).

E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud (2011). “Predicting At-risk Novice Java Programmers Through the Analysis of Online Protocols”. In: *Proceedings of the Seventh International Workshop on Computing Education Research*. ICER ’11. Providence, Rhode Island, USA: ACM, pp. 85–92. ISBN: 978-1-4503-0829-8. DOI: https://doi.org/10.1145/2016911.201693010.1145/2016911.2016930. URL: http://doi.acm.org/10.1145/2016911.2016930 (cit. on pp. 32, 62).

E. S. Tabanao, M. M. T. Rodrigo, and M. C. Jadud (2008). “Identifying at-risk novice java programmers through the analysis of online protocols”. In: *Philippine Computing Science Congress* (cit. on pp. 25, 26, 32, 58, 61, 62).

Technical Note TN2123: CrashReporter (2016). URL: https://developer.apple.com/library/mac/technotes/tn2004/tn2123.html (visited on 01/24/2016) (cit. on pp. 120, 168).

Ubuntu Wiki contributors (Nov. 22, 2012). *Apport*. Ed. by C. Watson. Wiki article. Accessed: 2016-08-30. URL: https://wiki.ubuntu.com/Apport (visited on 08/25/2016) (cit. on p. 129).

E. M. Voorhees et al. (1999). “The TREC-8 question answering track report”. In: *Proceedings of TREC*. Vol. 8, pp. 77–82 (cit. on p. 39).

C. Walters (Apr. 2011). *Use SIGTRAP (via G_BREAKPOINT()) if G_DEBUG=fatal-warnings*. URL: https://mail.gnome.org/archives/commits-list/2011-april/msg11156.html (cit. on p. 145).

M. Wand (1986). “Finding the Source of Type Errors”. In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’86.

St. Petersburg Beach, Florida: Association for Computing Machinery, pp. 38–43. ISBN: 9781450373470. DOI: https://doi.org/10.1145/512644.51264810.1145/512644.512648. URL: https://doi.org/10.1145/512644.512648 (cit. on p. 1).

S. Wang, F. Khomh, and Y. Zou (May 2013). "Improving bug localization using correlations in crash reports". In: *2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*. 2013 10th IEEE Working Conference on Mining Software Repositories (MSR), pp. 247–256. DOI: https://doi.org/10.1109/MSR.2013.662403610.1109/MSR.2013.6624036 (cit. on pp. 120, 128, 158, 170, 173, 174, 178, 179, 199, 200).

W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest (2009). "Automatically finding patches using genetic programming". In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, pp. 364–374. ISBN: 978-1-4244-3453-4. DOI: https://doi.org/10.1109/ICSE.2009.507053610.1109/ICSE.2009.5070536 (cit. on pp. 30, 33, 54, 64).

R. Wu, H. Zhang, S.-C. Cheung, and S. Kim (2014). "Crashlocator: Locating crashing faults based on crash stacks". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, pp. 204–214. URL: http://dl.acm.org/citation.cfm?id=2610386 (visited on 01/08/2016) (cit. on pp. 173, 174, 176, 177, 200, 201).

Zerone (Apr. 14, 2009). *Rhythmbox crashes while copying songs to psp*. Accessed: 2016-09-02. URL: https://bugs.launchpad.net/ubuntu/+source/rhythmbox/+bug/361259 (visited on 04/14/2009) (cit. on pp. 143, 153).

Y. Zhao and G. Karypis (2001). *Criterion functions for document clustering: Experiments and analysis*. Tech. rep. Retrieved from the University of Minnesota Digital Conservancy. URL: https://hdl.handle.net/11299/215490 (cit. on p. 186).