Applications of the Naturalness of Software

Eddie Antonio Santos

Department of Computing Science

April 2018

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

Abstract

There is a wealth of software development artifacts such as source code, issue reports, and revision histories, contained within publicly-accessible and privately-accessible repositories. Mining this data presents myriad opportunities that may benefit future software development efforts; however it is unclear exactly how to leverage this data. This thesis explores the *naturalness of software*—the assertion that source code, much like natural languages, is regular and predictable. This enables the application of techniques borrowed from the fields of natural language processing (NLP) and information retrieval (IR) to gain insight from existing software repositories. This thesis demonstrate different methods of mapping software artifacts to natural language models and full-text databases. Then, we show how to use these models and databases in the tasks of predicting whether a commit may break the build; clustering crash reports in a scalable and time-efficient manner; and detecting and correcting syntax errors in code written by novices. This thesis empirically demonstrate the effectiveness of these tools on real world software repositories. I conclude by suggesting ways of further exploiting the data contained within software repositories.

Preface

Chapter 3 is adapted from “Judging a commit by its cover: correlating commit message entropy with build status on Travis-CI” published [133] at the 13th International Conference on Mining Software Repositories (MSR), 2016, and has contributions from Dr. Abram Hindle with manuscript composition.

Chapter 4 is adapted from “The unreasonable effectiveness of traditional information retrieval in crash report deduplication”, published [29] at the 13th International Conference on Mining Software Repositories (MSR), 2016, and has contributions from Joshua Charles Campbell, as an equal coauthor; and Dr. Abram Hindle with manuscript composition.

A version of Chapter 5 is adapted from “Syntax and *Sensibility*: Using language models to detect and correct syntax errors” published [132] to the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, and has contributions from Joshua Charles Campbell with concept formation, authoring code to perform the evaluation, and manuscript composition; Dhvani Patel with research assistance; Dr. Abram Hindle, and Dr. J. Nelson Amaral with concept formation and manuscript composition.

Acknowledgements

I would like to thank my supervisor, Dr. Abram Hindle for his continued mentoring, frank advice, and for his steady stream of corny jokes. I also thank Dr. Eleni Stroulia and Dr. Jia You for their efforts in admitting and accommodating me in my graduate program. I thank my many academic collaborators: Dr. Karim Ali, Dr. J. Nelson Amaral, Dr. Antti Arppe, Michael D. Feist, Isabell Hubert, Dr. Jordan Lachler, Carson McLean, Christopher Solinas, Dhvani Patel, and Ian Watts. In particular, I thank Hazel Campbell for our frequent collaboration and for their want to believe in the freedom of music. I also thank my lab mates Shaiful Alam Chowdhury, Candy Pang, Stephen Kalen Romansky, and Rameel Sethi for their frequent insight and kindness.

This work was assisted with funding from BioWare ULC, the University of Alberta Faculty of Science, NSERC, and the province of Alberta. I also thank Nvidia Corporation.

Finally, I thank my parents, Edy and Carmen, and my partner, Jessica, who have provided me endless support and encouragement. Te quiero mucho!

Table of Contents

List of Tables

List of Figures

List of Acronyms

API application programming interface

AST abstract syntax tree

CPU central processing unit

ECDF empirical cumulative distribution function

GPU graphical processing unit

GRU gated recurrent unit

HDF5 Hierarchical Data Format, version 5

IDE integrated development environment

IR information retrieval

JSON JavaScript Object Notation

LSTM long short-term memory

MLE maximum likelihood estimation

MLP multilayer perceptron

MRR mean reciprocal rank

MSR mining software repositories

NFC Unicode normalization form C

NLP natural language processing

RNN recurrent neural network

SBSE search-based software enginering

UTC coordinated universal time

VCS version control system

WER Windows Error Reporting

Chapter 1

Introduction

The exponential popularity of websites that cater to software development such as GitHub [52], and Travis CI [157], has brought along with it an avalanche of publicly available software artifacts; these are artifacts such as source code, commit logs, and issue reports, that are products of the software development process. Software engineering researchers and practitioners alike stand to benefit greatly from the analysis and dissection of this public data. This data can be leveraged to predict when a change to the codebase will introduce a fault; or we can categorize a new crash report with already known crashes; or we can use historical data to help find and fix mistakes as we're typing new code. If one poses these questions as *search problems*—that is, searching for a solution out of a large space of possible solutions—the quantity of possible solutions offered by these massive software repositories becomes dauntingly large. In order to make any such search problem feasible, we must find a way to **reduce the search space**.

Natural languages, such as English, American Sign Language, or Hausa, allow for great flexibility in expression; as a result, natural languages present an unfathomably large possibility space. Despite this, natural language processing (NLP) researchers have noted that utterances in natural language are often *regular* and *predictable*: the languages humans use to communicate with each other are, in general, rife with redundancy, and often posses very little novel information content [136]. As such, one can exploit these properties to heuristically prune the search space. For example, in speech recognition, one may need to determine whether a speaker said "I went to **their** house" or "I went to **there** house". These utterances are *homophones*—that is, most fluent English speakers will pronounce the two phrases identically, thus, a speech recognition system will not be able to distinguish between the two possibilities using the audio signal alone. With *statistical language models*, we can count how often “I went to” is followed by “there” and how often it is followed by “their”; likewise, we can count how often “house” follows “their” and how often it follows “there”. By counting the occurrence of these phrases in a *corpus*—a large body of text that is somehow representative of the language one is modelling—it becomes clear that “I went to **their** house” is quantifiably more likely, and thus, is likely what the speaker said (discussed in greater detail in Section 2.2.2). As a corollary, certain utterances are quantifiable *unlikely*: these anomalies may be indicative of mistakes or creative usage.

Hindle et al. [74] asserted that software source code “is a natural product of human effort”; thus, the authors posited that source code, like natural language, possesses the properties of being regular and predictable. Using *n*-gram language models, the authors measured the average information content (the *cross-entropy*) on Java source code and two English corpora. They found that Java source code had lower cross-entropies than the English corpora meaning that these software corpora are even *more* regular and predictable than English text. The authors dubbed this property “the *naturalness* of software”. The *naturalness* of software raises the possibility of using tools originally intended for natural language, such as *n*-gram language models, and full-text search engines, and applying them to software artifacts.

There is one problem, however: language models and full-text search engines work with *tokens*—the smallest unit of language that carries meaning by itself. Thus, to employ language modeling tools to reduce the search space in software engineering problems, one must create a mapping of software artifacts to a sequence of tokens. As I demonstrate throughout this thesis, *tokenization* is not always straightforward, and sometimes requires experimentation.

In this thesis, I discuss how we can exploit the *naturalness* of software—the regularity and repeatability of source code and other software artifacts—as a way to **reduce the search space** such as when finding possible duplicates of new crash report, or when finding the right way to fix a syntax error.

1.1 Contributions

The contributions I have made in this thesis are as follows:

- I established a methodology for the "unnaturalness" of commit log messages. Using this methodology, I empirically discovered that the "unnaturalness" of a commit log message is a weak, yet statistically significant feature in predicting build failure (Chapter 3).
- I co-implemented a crash report deduplication tool that uses the principals of information retrieval to provide online, non-stationary clustering of crash reports, without developer intervention (unsupervised clustering); it can cluster crash reports at the volumes seen by industry. This was in response to a number of existing crash report deduplication tools attested in both academia and industry that either used crude, hand-written, developer-maintained heuristics, or cluttering algorithms that run in at least $O(n^2)$ time—which are untenable for the volume of crash reports seen in industrial contexts. I collected a number of hand-written "signature generation" heuristics attested in literature, which we used as a baseline comparison. Our tool, PartyCrasher, outperformed existing heuristics in overall F-measure, as well as providing an adjustable trade-off of precision and recall *after clustering* (Chapter 4).
- I developed a heuristic based on naturalness that finds fixes for single-token syntax errors. The fix generation heuristic uses *naturalness* to bias the search for the correct syntax error location and the correct token to insert or substitute; thus, instead of performing an exhaustive search for a viable solution, my approach can generate a valid solution within a small constant number of trials. Based on my empirical analysis of BlueJ interaction logs, we found that single-token syntax errors account for the majority (57 %) of syntax errors made be novice programmers. Importantly, the training data for the heuristics does *not* come from novice code; rather, it comes from code sampled from the most popular Java repositories on GitHub. Thus, unlike prior work, the approach I proposed is general-purpose and redistributable. Given these two facts, I believe there is evidence that my approach is useful to novices (Chapter 5).

1.2 Thesis organization

This thesis is organized into the following chapters: Chapter 2 presents background knowledge on language modeling, information retrieval, and recurrent neural networks; Chapter 3 presents an application of naturalness on commit messages using the principles of tokenization and unnaturalness; Chapter 4 applies classical information retrieval algorithms on the problem of crash report clustering, discussing various methods of tokenization prior to the ingestion of documents; Chapter 5 shows how language modeling can be used to find and fix syntax errors in the Java programming language. Finally, Chapter 6 concludes this thesis and suggests future research directions.

Chapter 2

Background and related work

In this chapter, we explain what *naturalness* is by explaining its context in *mining software repositories* (Section 2.1). We describe the techniques borrowed from *natural language processing* (Section 2.2) and how they can be applied to software (Section 2.3). We describe *information retrieval* (Section 2.4) and *recurrent neural networks* (Section 2.5), which we used in this thesis to add value to software artifacts. Finally, we provide a listing of prior work in *the naturalness of software* and its existing applications (Section 2.7).

2.1 Mining software repositories

This thesis broadly concerns mining software repositories (MSR). Unpacking the phrase, a *software repository* is any cohesive collection of artifacts regarding a particular piece of software. Those familiar with *version control systems* might assume that a software repository is synonymous with a collection of a project's source code and its associated change history, or simply the source code itself in its present published form. However, in this thesis, we use the term "software repository" more broadly. Not only is a project's source code contained within a version control system (VCS) such as Git, Subversion (SVN), Concurrent Versions System (CVS) a software artifact; other software artifacts include:

commits (also, *revisions* or *changesets*) are discrete, incremental sets of changes to a source code repository, created by a primary **author** and accompanied by a **commit message** (also, a *log message*) describing, in natural language, what changes have been made. Commit logs are interesting, because the series of commits that make up the current version of the software show the incremental development and evolution of a piece of software. Commits are the primary artifact maintained by version control systems such as Git, CVS, or SVN; consequently, commit logs are available through public VCS hosting such as GitHub [52], SourceForge [140], BitBucket [15], and Google Code [56] (now defunct). Commits are also available through secondary services intended specifically for MSR tasks such as GHTorrent [60] and Boa [44].

bug trackers (also, *issue trackers*) are repositories of *bug reports* which are often hand-written reports regarding an issue, fault, or shortcoming of the software product. Since they are often free-text, bug reports consist of primarily natural language text, though they may also contain semi-structured metadata about how the issue was encountered. Publicly available bug trackers include Bugzilla [115], GitHub Issues [53], Jira [16], and Trac [45].

crash trackers are repositories of *crash reports*, which catalogue a particular *software crash* on a remote computer. As opposed to bug reports, crash reports are often automatically generated and tend to contain more structured data—diagnostics of the computer and software systems that encountered the software crash. For this reason, they may be considered a form of *telemetric error reporting*. Due to their similarity to bug reports, crash repositories may share the same infrastructure [30]. Some publicly accessible crash repositories are Canonical Launchpad [30] and Mozilla Crash Error Reports [113].

build history are repositories that enumerate the status of *builds* over the course of developing a software system. A build historically refers to the compilation of a codebase at a particular point in time, however, in modern parlance, a “build” may also include the results of running automated test suites. Periodic builds are often automated through the use of *continuous integration* services. These services not only perform periodic or on-demand builds of software systems, but also maintain a history of passed and failed builds. Publicly available continuous integration services include AppVeyor [14], CircleCI [34], Jenkins [143], and Travis CI [157].

These mounds of software artifacts are often *interrelated* and have explicit and implicit links. For example, a bug report may be related to a crash report; a crash may be caused due to a certain line of code; the code was written with a particular intent, documented by the developer’s commit message; the impetus for the change may have been a point of discussion in the issue tracker, or in the developers' mailing list; the discussion may reference a fix for an earlier failing build. Finding and studying the implicit and explict links between software repositories is called *traceability* [84].

To *mine* these repositories is to sift through the reams of data and add value in some way. "Value added" includes understanding how and when faults are introduced [141, 160, 64, 87], automatically triaging bug reports to the most appropriate developers [107, 4, 96, 90, 95, 130], understanding the evolution of software projects [49, 75, 71], increasing developer productivity [74, 134, 162, 5, 128], and preventing the introduction of faults [29].

There is little doubt that mining large repositories of software artifacts is a worthwhile endeavour, however, there remains the question of *how* to extract value from the semi-structured data sources that comprise a software project.

2.2 Natural language processing

Natural language processing (NLP) is the application of quantitative methods to examine, predict, and manipulate *natural language* [104]—that is, languages that are the result of unconscious, organic evolution such as English, Plains Cree, or Nicaraguan Sign Language. Contrast these to *artificial languages* that are consciously constructed and meticulously designed. Common tasks in natural language processing include **language modelling** and **prediction**, **machine translation**, **speech recognition**, **part-of-speech tagging**, and **optical character recognition**, among many more. Many tasks in the related discipline of **information retrieval** (discussed in Section 2.4) require preprocessing using techniques borrowed from NLP.

In this thesis, we focus primarily on the subfield of NLP called *language modelling*—capturing the essence of a language such that we may reason about its **regularity** and make **predictions** about incomplete utterances.

2.2.1 Tokenization

Prior to applying any quantitative methods to natural language within a computer, the input must be transformed into an appropriate *representation* of natural language. Natural language input may come in the form of a digital audio signal, a digital image, or as a linear sequence of *characters* or *code points*,[1] themselves representing a series of *graphemes* in a written language. In this thesis, we focus

[1] Henceforth, we will use *character* and *code point* interchangeably; however a *code point* is specifically an integer allocated in the Unicode [152] code space. Not all code points are defined, and not all defined code points are characters.

primarily on the latter type of input. Regardless of the medium, the original input representation is usually not amenable to quantitative analysis.

Language models require a representation where each element of the input must be an indivisible unit of meaning, called a *token* [104]. The process of transforming one representation of language into a linear sequence of meaningful units is called *tokenization*. Consider the sentence fragment `I’ll be right`. We want to determine the most likely English word that would proceed this fragment. This fragment is represented as a series of code points: latin capital letter i, apostrophe, latin small letter l, another latin small letter l, space, and so on. When modelling the English language, we may be tempted to use the code point as the basic unit of meaning. However, note that the space code point is simply used in written English to separate two words in the language. Thus, perhaps the better option is to coalesce each word in the language into a single token, and discard all space characters. Thus, instead of letting each character be an indivisible unit of meaning, our sequence of tokens is instead $\boxed{\text{I’ll}}$ $\boxed{\text{be}}$ $\boxed{\text{right}}$. Note that, under this scheme, we consider $\boxed{\text{I’ll}}$ to be one unit of meaning, whereas a fluent English speaker understands *I’ll* to be a contraction of the fragment *I will*. `I’ll` would itself be transformed into two tokens under this scheme: $\boxed{\text{I}}$ and $\boxed{\text{will}}$. We must chose whether it is more appropriate to model `I’ll` as one token ($\boxed{\text{I’ll}}$) or as two ($\boxed{\text{I}}$ $\boxed{\text{will}}$). Note, however, that the latter choice would transform the input `I will be right` into the same sequence of tokens, and thus both inputs would be consider as an identical utterance in the eyes of our language model. Perhaps, we may want to decompose `I’ll` into two tokens, however we’ll preserve the *clitic* `’ll` as its own token. Thus, the input `I’ll` is transformed into two tokens $\boxed{\text{I}}$ and $\boxed{\text{’ll}}$ that are distinct from the tokenization of the fragment `I will`: $\boxed{\text{I}}$, $\boxed{\text{will}}$.

Tokenization is rife with making decisions as illustrated in the previous example. The decisions of transforming a language utterance into a sequence of tokens factor into one’s goals in language modelling. In this example, we did not factor for differences in letter case or the consideration of punctuation—what’s more, we simply glossed over semantic distinctions that could be made given differences in spacing. A common theme in this thesis is the exploration of different tokenization schemes, guided by a specific modelling goal.

That said, when referring to code points in this thesis, we usually mean any code point that is a graphic or formatting character (letters, numbers, combining marks, and spacing characters).

2.2.2 Language modelling

Given tokens from an input language, a *language model* is a probability distribution defined over a sequence of tokens (an *utterance*) from an input language [104]. Language models are useful for determining how likely an utterance came from the input language, and for predicting the most likely tokens to complete an utterance, among other uses.

A language model for predicting tokens in an utterance can be considered as the conditional probability distribution of a token w_n as a function of all tokens preceding it, called its *history* [104].

$$\Pr(w_n|w_1, \ldots, w_{n-1})$$

However, it is impractical to maintain a record of every arbitrary prefix of tokens. A practical approximation employs the *Markov assumption*—only the *local* prior context affects the likelihood next token [104]. An n-gram language model uses a truncated history of $n-1$ tokens of local context of tokens immediately prior to the token we are interested in predicting, where n is the *order* of the n-gram model. To construct an n-gram model, we count the occurrence of every token sequence of length n witnessed in a *corpus.* A corpus is a large existing body of text that is considered to be representative of the language.

Thus, if one is interested in the probability of a token w_i using an n-gram model, we compute the maximum likelihood estimation (MLE) using,

$$\Pr(w_n|w_1, \ldots, w_{n-1}) \approx \frac{C(w_1 \cdots w_n)}{\sum_{w_* \in V} C(w_1 \cdots w_*)} \tag{2.1}$$

where $C(w_1 \cdots w_n)$ denotes the count of occurrences a particular n-gram has been witnessed in the corpus, $C(w_1 \cdots w_*)$ denotes how many times any n-gram with the same prefix of $n-1$ tokens has been witnessed in the corpus, and V denotes the *vocabulary*—the set of all possible individual tokens.

2.2.3 Cross-entropy and perplexity: measuring language model quality

To measure how well a language model reflects the actual language, we often use *cross-entropy* and *perplexity* as measures. Roughly speaking, cross-entropy, denoted as $H(p, q)$, is a measure of how much information[2] on average is required to explain an outcome x, $x \in X$. p and q are probability distributions, where p is the "true" distribution, and q is a probability distribution that models p. According to Manning and Schütze [104]:

> The secret [is] the idea that *entropy* is a measure of our uncertainty. The more we know about something, the lower the entropy will be because we are less surprised by the outcome of a trial.

Cross-entropy can be defined as the addition of the entropy of the true distribution p (denoted as $H(p)$), with the *relative entropy* (or Kullback-Leibler divergence) of p and q (denoted as $D_{\mathrm{KL}}(p||q)$):

$$\begin{aligned}
H(p,q) &= H(p) + D_{\mathrm{KL}}(p||q) \\
H(p,q) &= -\sum_{x\in X} p(x)\log p(x) + \sum_{x\in X} p(x)\frac{\log p(x)}{\log q(x)} \\
H(p,q) &= -\sum_{x\in X} p(x)\log q(x)
\end{aligned}$$

Muller et al. [117] provide an excellent intuitive explanation of cross-entropy:

> The cross entropy measures the amount of surprisal obtained when you believe an event is distributed as Y, but in reality is distributed as X. This amount is not typically symmetric in Y and X. The cross entropy is minimised when Y is selected to be equal to X, in which case the believed distribution equals reality. [3]

In this thesis, we use cross-entropy to quantify the surprisal obtained from learning that an estimated distribution q is actually distributed p. Greater surprisal means an event is more "unnatural", or contrary to expectations of the language model.

In the NLP community, a related measure, *perplexity*, is often used instead of cross-entropy to measure how well a method predicts samples from a probability distribution (such as a language). Perplexity is derived from cross-entropy in Equation 2.2:

$$\text{Perplexity} = 2^{H(p,q)} \tag{2.2}$$

[2] Information is routinely measured in *bits*, hence all equations in this section using an unadorned "log" refer to the base-two logarithm, or $\log_2$.

[3] What Muller et al. write as Y and X, we notate as p and q, respecively.

Perplexity may be preferred to cross-entropy, as it may be intuitively interpreted as the "number of choices", on average, that a model has to choose from to predict an outcome. The lower the perplexity, the more likely it is to predict an outcome correctly.

In this thesis, I prefer to use cross-entropy ("how much information") to perplexity ("how many choices"). We provide the definition of both, as one may be derived from the other.

2.2.4 Smoothing

When n-gram models are used in practice, is almost unavoidable to encounter the issue of *data sparsity*, especially as the order n becomes larger. It is infeasible that every n-gram possible in the language will exist in a corpus. There will always be an n-gram that a human observer would consider as being a probable, natural utterance in the language, yet is not present at all in the corpus. Since this token sequence in its entirety has never been witnessed, its count (the numerator in Equation 2.1) is zero, and thus its probability according to the n-gram language model will also be zero.[4] A key insight in resolving this is the fact that unknown n-length token sequences are often composed of smaller token sequences that *have* been witnessed in the language model.

To resolve the issue of data sparsity, *smoothing* is used to estimate the likelihood of unknown n-grams. "Smoothing" refers to the shape of the probability distribution after zeros are removed from it by "discounting" highly probable events [104]; token sequences that would normally have zero probability are lent the probability from more probable events. Many n-gram smoothing techniques have been devised; Chen and Goodman [32] performed an empirical study of the existing techniques at the time, and introduced a novel smoothing technique—an extension of an earlier smoothing technique by Kneser and Ney [91]—which they found to be the best performing of all smoothing techniques that they surveyed. Thus, this thesis exclusively uses **modified Kneser-Ney smoothing** for n-gram language models.

Returning to the topic of mining software repositories, are n-gram models—created to solve problems in natural language processing—appropriate for use on software source code?

[4] This also implies taking the logarithm of zero when calculating cross-entropy, which is an undefined result.

2.3 The *naturalness* of software

Hindle et al. [74] posited that software source code is “natural” in the sense that, due to it being created manually by human programmers, source code exhibits the predictability and regularity typical of natural languages, such as English [136]. That is, “despite being written in an artificial language (like C or Java), [source code] is a natural product of human effort” [74]. The authors validated this hypothesis by estimating (training) n-gram models on both English corpora (the Brown and Gutenberg corpora), as well as on software projects written in C and Java. Figure 2.1 plots the average cross-entropy as the order n of the n-gram models increases, for three cases: English text against an English corpus, two held-out code samples against its corresponding code corpus, and code in general. The average cross-entropies across all orders n tried were consistently lower in the Java source code language models than for the English language model. This suggests that code is *more* regular and predictable than English, and thus, source code lends itself to modeling with n-gram models. The authors implemented one possible application of the *naturalness*: a code suggestion engine that predicts the next token that a programmer types. By augmenting the Eclipse IDE’s existing suggestion engine, the authors demonstrated that statistical language models can provide from 33%–67% additional correct code suggestions, saving an estimated 61% of keystrokes that the programmer would otherwise have to input.

The authors demonstrate that statistical natural language modeling is adept at modeling software source code—are there any other models, traditionally used for natural language that can be used on source code, and other software artifacts?

2.4 Information retrieval

Information retrieval (IR) is the study of satisfying a user’s *information need*, typically under a tight time constraint. Typically, this means finding the most relevant *document* or subset of documents from a very large collection. According to Baeza-Yates et al. [17], IR encompasses “the representation, storage, organization of, and access to information items such as documents, web pages, online catalogs, structured and semi-structured records, and multimedia objects.” In this thesis, we will focus on IR’s applications to searching semi-structured text documents.

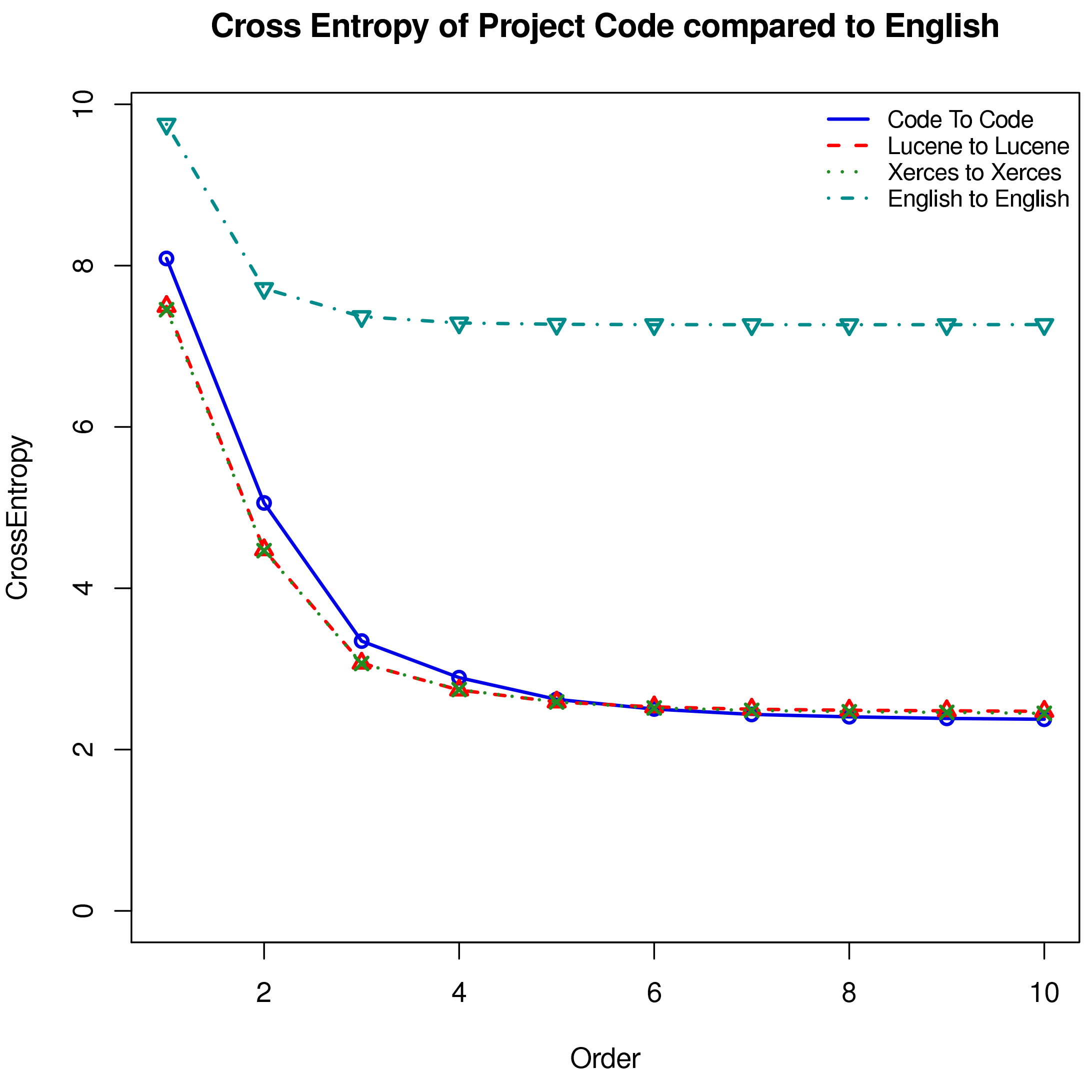


Figure 2.1: Cross-entropy of code-to-code, and English-to-English (adapted from Hindle et al. [74]).

To search a large collection of documents efficiently (in terms of time), one needs an effective data structure that facilitates searching. For this, an *inverted index* is used. An inverted index maps each *term* (roughly synonymous with token from Section 2.2) to documents that contain that term. Inverted indices are designed such that the time complexity for returning documents which contain a given term is either constant, with respect to number of documents in the collection, $|D|$, or, has a relatively small upper-bound, such as $O(\log |D|)$.

Quickly finding the set of all possibly relevant documents is one-half of the problem; the other problem is ranking the results from most relevant to least relevant. For this, tf-idf was created. tf-idf, or *term frequency* × *inverse document frequency*, ranks a document's relevance to a query using the importance of each term t of a query q, within the document (its *term frequency*—how many times the term t appears in the document d), and the rarity of the term t throughout the entire collection (its *inverse document frequency*—the inverse of the amount of documents in which term t is found). For example, the Lucene information retrieval system defines the score of a document d given a query q as follows [101]:

$$\mathit{score}(q, d) \propto \sum_{t \in q} \Big(\mathrm{tf}(t \in d) \cdot \mathrm{idf}(t)^2 \Big) \tag{2.3}$$

As with NLP, information retrieval often requires special care in tokenizing—both the documents ingested into the searchable collection, and the queries used to filter the results.

2.5 Recurrent neural networks

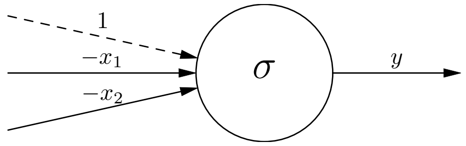
To discuss recurrent neural networks (RNNs), it is useful to discuss *artificial neurons*. An artificial neuron is a function which, given a linear combination of inputs $x_1 \cdots x_n$, produces an output, filtered through an *activation function* or "squashing function" [23]. Artificial neurons are parameterized by their input weights $w_1 \cdots w_n$, their bias b, and their activation function σ. Thus, the output y of an artificial neuron is given by Equation 2.4 [12, 23].

$$y = \sigma \left(b + \sum_{i=1}^{n} w_i x_i \right) \tag{2.4}$$

Consider a neuron with two inputs x_1, x_2, and the activation function σ such that its output is 1 when its input is greater than or equal to 0, and outputs 0 otherwise[5]. A neuron with a binary

[5] Also known as the *Heaviside step function*, denoted as H or θ.

output, such as that given by this activation function, is called a *perceptron.* Let the weights $w_1 = -1$, $w_2 = -1$ and let the bias $b = 1$. Notice that when x_1 and x_2 are given inputs in $\{0, 1\}$, the artificial neuron is equivalent to the binary NAND function ($\uparrow$) (Figure 2.2).



x_1	x_2	y	$x_1 \uparrow x_2$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Figure 2.2: The NAND perceptron.

Since NAND is functionally complete—all other boolean functions can be produced using a combination of NAND gates [137]—then artificial neurons may also be combined to produce any boolean function [109]. A function composed of multiple artificial neurons is called an *artificial neural network.* In practice, neurons are not composed arbitrarily, but rather by creating *layers* of mutually-disconnected neurons, and stacking the layers on top of each other, connecting the outputs of higher layers to every input in the layer directly below it, such that connections between neurons contains no cycles. Such a neural network architecture is called a multilayer perceptron (MLP) [23] or, more generally, a *feedforward neural network* [12].

An extension of the feedforward architecture is the aforementioned recurrent neural network (RNN). In an RNN architecture, the previous output of each neuron is fed back into itself (up to a fixed number of *time steps*) and becomes an additional input—hence, it is *recurrent.* RNNs use the recurrent input as a way to represent internal state, and are adept at learning time sequences and series. RNNs are Turing complete [139]. Thus, one may compose RNNs and assign precise parameters (weights and biases) to perform any computable function. This in and of itself is not remarkable; what is remarkable is that there are procedural means by which to *learn* the parameters given a wealth of *training examples*, so long as the activation function is continuously differentiable.

The algorithm to learn the parameters is called *backpropagation* [161]. As training examples are repeatedly presented to the training algorithm in *batches*, backpropagation updates the parameters iteratively, using the error or *loss* between the training outputs and the actual outputs of the network as a guide. Backpropagation calculates the *gradient* of each weight and bias in the network—that is, it determines how changing a parameter will affect the loss. This gradient dictates how the parameters should be tweaked in order to reduce the loss with respect to the example outputs. The process of using a batch of random training examples to update parameters (*stochastic gradient descent*) is iterated until loss is sufficiently minimized—that is, until the weights and biases in the network have learned the desired function to a satisfactory degree.

A problem presents itself when training "deep" neural networks such as RNNs. As the number of layers—or in the case of recurrent networks, time steps—increases, the gradients calculated during backpropagation become smaller and smaller, although the algorithm does not get proportionally closer to minimizing the training loss. This is known as the *vanishing gradient problem.* As a result of the compounding of smaller gradients which dictate increasingly minute changes in the weights and biases further back in the network, it becomes increasingly time-consuming to achieve high accuracy when training deep neural networks.

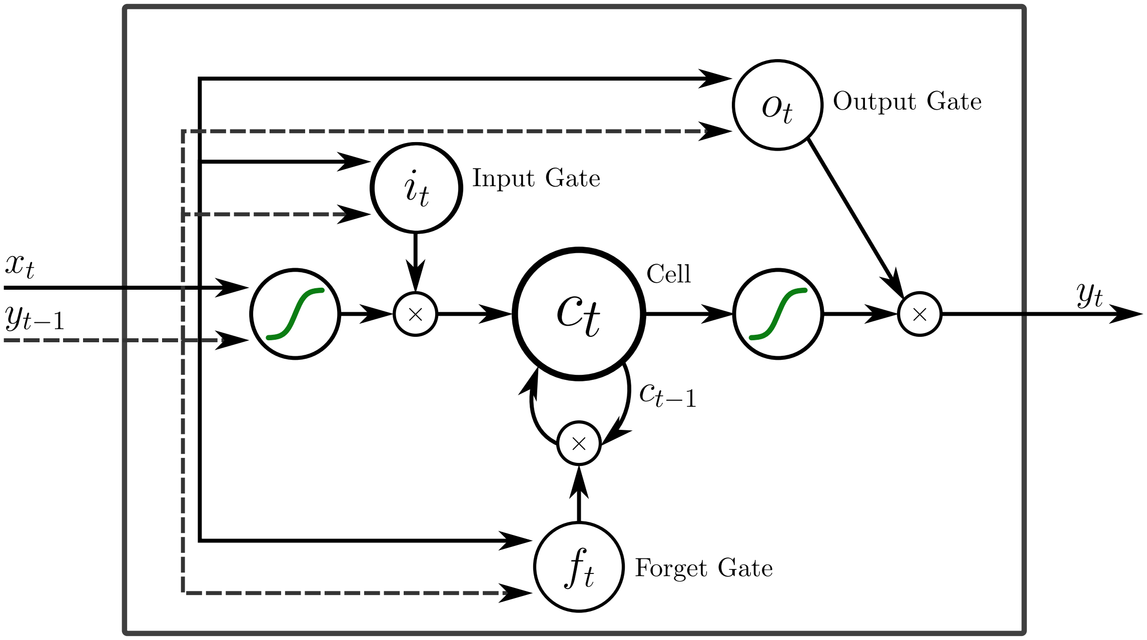


Figure 2.3: An LSTM block [76], featuring the memory cell c_t, the input gate i_t, the output gate o_t, and the forget gate f_t. The "S"-shaped circles represent some non-linear, contiguously differentiable "squashing" function, such as tanh or the logistic function. Each arrow represents a vector of inputs (adapted from Fig. 1. in Graves et al. [61]).

To alleviate the vanishing gradient problem, Hochreiter and Schmidhuber [76] created the long short-term memory (LSTM) architecture, as an extension of RNNs. LSTMs (Figure 2.3) protect the gradient of each neuron by abstracting it into a *memory cell* that "traps" the internal state of the neuron. Each memory cell is protected by three *gating units*—an input gate, an output gate, and a forget gate—which allows each "block" to determine how to consider its input, whether to overwrite the value in the memory cell, and whether to output the previous contents of its memory cell. Critically, the previous state of the memory cell c_{t-1} is never “squashed” by an activation function (such as tanh or the logistic function) prior to being fed back into the memory cell; instead, it is fed back verbatim. Thus, these processes can preserve a constant error rate as training progresses. As a result of the practical benefits of shorter training time, in Chapter 5, we use LSTMs exclusively as the recurrent neural network of choice.

As mentioned, RNNs and, by extension, LSTMs are adept at learning functions related to sequences. As such, they may be employed to create language models, similar to n-gram models (Section 2.2.2). LSTMs have enjoyed success in the space of speech recognition [61, 66, 10], creating “bidirectional” language models that learn acoustic and linguistic models of speech both “forwards and backwards in time”. Chapter 5 describes a similar approach, however we applied bidirectional LSTMs to create language models of software source code.

2.6 Search-based software engineering

Search-based software enginering (SBSE) is the application of *search-based optimization* to software engineering problems [67]. This means that a problem is defined as having a number of possible solutions with varying degrees of suitability or “fitness”. Given a *fitness function*, that allows one to compare the suitability of one solution to another, search-based optimization trawls through the search space to find one or more suitable solutions. Compare the usage of the word “search” in the context of SBSE to its usage in IR (Section 2.4): In IR, “search” means to apply a query to collection of documents, and find a set of suggestions that *may* be relevant to the query; in contrast,“search” in SBSE is the process of evaluating solutions and attempting to navigate the search space until at least one solution of suitable fitness is found.

Typically, the search space is so large and multidimensional that it is intractable to use an exhaustive search (evaluating the fitness of every possible candidate) to find a suitable solution. As such, a staple of SBSE is the use of heuristics to trim the search space when evaluating candidate solutions. Such heuristics include genetic algorithms and hill climbing [68] (such as stochastic gradient descent; c.f. Section 2.5).

An example application of SBSE is *automated program repair*, wherein the input is a buggy program, such as one containing a buffer overflow or an infinite loop, and the output is that same program lacking the bug. GenProg by Le Goues et al. [59] is one such example of automated program repair using SBSE principles. GenProg represents the space of possible solutions as pairs of an abstract syntax tree (representing a possible program) and a weighted path representing how often a statement in the program is run (this helps localize a fault). The fitness function takes the program (generated by the abstract syntax tree) and a test suite for the buggy program. The goal is to maximize the fitness (i.e., the program compiles and all test cases pass). To navigate the space of solutions, GenProg uses *genetic programming* to create many random edits to the program (a generation) and randomly mixes "traits" of the programs with the highest fitness. This process is iterated until the fault is removed (the entire test suite passes).

In this thesis, we pose *naturalness* as a way to bias search (in the SBSE sense). We posit that *naturalness* is useful for carving out the search space of possible solutions to software engineering problems in a time efficient manner. Cross-entropy (Section 2.2.3) can either be minimized ("naturalness") or maximized ("unnaturalness") as a fitness function to evaluate candidate solutions.

2.7 Applications of the *naturalness* of software

In this section, we discuss how *naturalness* has been applied to other software artifacts. Additional prior work is discussed in greater detail in Chapters 3, 4, and 5.

As an extension of Hindle et al. [74], Tu et al. [158] describe the *localness* of software: in addition to using n-gram language models, the authors employed the use of *cache language models*—dynamic language models which change the probability of utterances given *local* context. The authors show that software source code exhibits the properties of being *endemic* (tokens that occur only in one source file), *specific* (tokens that occur only in a few related source files), *proximally repetitive* (tokens that occur frequently in a small range of tokens). Applying *localness* to code suggestion, combing a "global" language model—trained on a large corpus of code—with a local, cache language model—trained on a sliding window of code examples local to the current modification point—improves suggestion accuracy considerably.

As opposed to code completion, Allamanis et al. [5] used *naturalness* to mine coding conventions in shared source code, to increase the *stylistic consistency* of large source code repositories. The authors' tool, *Naturalize* trains a language model on the stylistic conventions of a code base, including formatting and the naming of identifiers. The framework can not only determine if a file is "natural" with respect to the rest of the corpus, but can also generate stylistic fixes. The authors evaluated this by applying *Naturalize* to open source Java projects, and applying the automated changes. The authors opened *pull requests*, such that the maintainers of the respective Java projects may accept the changes and merge them in the codebase. Out of 15 changes suggested, all but one were accepted in some way.

Allamanis and Brockschmidt [6] applied deep learning on source code to approximate solutions to intractable program analysis problems. SmartPaste solves the problem of automatically adapting snippets of source code from a foreign domain—such as from a different project, or from an online example such as Stack Overflow—to a separate source code file. The problem can be reduced to mapping identifier names in the foreign snippet, to the semantically equivalent variable names in the source code file. As opposed to many of the methods described to create models from source code which only consider *lexical* and *syntactic* information, SmartPaste integrates *semantic* information, such as the static types and scope of variables, and the dataflow of the partially completed source code file. These data sources are used as features to the gated recurrent unit (GRU) (a type of RNN) deep neural networks, which, given a “placeholder” identifier in the foreign snippet, output the most likely identifier in the current context of the source code. Using this technique, the authors reported 58.6% accuracy in mapping all variables in a snippet to the correct names in the local context.

Chapter 3

Judging a commit by its cover

In this chapter, we present the usage of language models (Section 2.2.2) on natural language—with a twist: we demonstrate tokenization (Section 2.2.1) that takes into account many artificial languages that may be embedded in natural language utterances by software developers. We train n-gram models, and use *cross-entropy* as a measure of "unusualness" or *unnaturalness*—a theme we revisit in Chapter 5. The special-cased tokenization decisions made in this chapter avoid the high-degree of *hapax legomena*—tokens that only appear once in the data—that would have resulted from the many instances of structured artificial languages, such as file paths, method names, and version numbers. Replacing these embedded languages with synthetic tokens increases their probability in the corpus, thus allowing them to be judged as *unnatural.* In this chapter, we also explain how we derived correlations from two interrelated software repositories—commit logs from GitHub, and build histories on Travis CI.

The takeaways and lessons learned from this chapter include:

- a methodology for quantifying the "unusualness" of a commit log message
- an non-comprehensive list of embedded languages in commit messages
- commit messages on GitHub are not always written in English
- not all commit messages are written by hand
- a statistically significant correlation does not imply a positive practical result

Readers should pay particular attention to Figure 3.1: this graph plots commit message cross-entropy against the cumulative count of build statuses collected from Travis CI. This graph shows the slight bump in the otherwise normal curve (displayed cumulatively) that contributes to the statistically significant result, however, is so minor as to render the conclusion of this chapter a null result.

This chapter is relevant to researchers mining commit log messages, and the interaction of the commit metadata with continuous integration builds. The MSR paper [133] on which this chapter is based on is already cited as basis to determine whether committers are following "best practices" [125]; uses part of our approach to mine for commits [24]; learns from our negative result [118]; or references this research as a discussion piece [73, 62] Applications of this research include studying the introduction of faults into software projects; predicting when a change will introduce a fault [118]; mining developer intent (why did the developer write this commit?); analyzing patch acceptance [125]; studying the patterns between when and how developers commit [24]; and determining the interplay between non-functional requirements and build statuses [123].

My personal contributions were writing the Boa, Python, Ruby, and R scripts [1] to collect, clean, parse, and analyze data from GitHub and Travis CI; I came up with the original concept, devised the experiment, wrote most of the write-up, and formed an educated conclusion.

This chapter is adapted from "Judging a commit by its cover: correlating commit message entropy with build status on Travis CI", published [133] at the 13th International Conference on Mining Software Repositories (MSR), 2016, and has contributions from Dr. Abram Hindle with data analysis and manuscript composition.

[1] Available: https://github.com/eddieantonio/judging-commits/tree/msr2016/

Abstract

Developers summarize their changes to code in commit messages. When a message seems “unusual” or *unnatural*, however, this puts doubt into the quality of the code contained in the commit. We trained n-gram language models and used cross-entropy as an indicator of commit message “unnaturalness” of over 120,000 commits from open source projects. Build statuses collected from Travis CI were used as a proxy for code quality. We then compared the distributions of failed and successful commits with regards to the unnaturalness of their commit message. Our analysis yielded significant results when correlating cross-entropy with build status.

3.1 Introduction

Commit messages are summaries written by developers describing the changes they have made during the development process. Our experience working within the open source community has given the impression that developers tend to use a fairly limited vocabulary and restricted structure when writing commit messages. Alali et al. [3] provide empirical evidence for this observation reporting that over 36% of all commit messages contained the word “fix” and over 18% contained the word “add”. Thus, developers may regard short, terse, and to-the-point messages such as “Add test for visibility modifiers” [146] to be *usual* when browsing the commit log of a code repository.

When a developer reads an *unusual* commit message that defies their expectations, such as “Cargo-cult maven” [48], they may ask themselves a number of questions. “Why did they write ‘cargo cult maven’?” “What changes prompted this cryptic commit message?” *“Should I trust the code behind this commit message?”* Such unusual (henceforth, *unnatural*) messages may induce suspicion; a developer reading this message may question the commit’s quality.

Should developers trust their instinct? This chapter seeks to answer the question: Are unnatural commits hiding bad code? We break this down into the following research questions:

RQ1: How can the *unnaturalness* of a commit message be measured?

RQ2: Is the *unnaturalness* of a commit message related to the quality of the code committed?

3.2 Methodology

In natural language processing, *n*-gram language models are used to answer questions about frequency, surprise, and unnaturalness. We use the *cross-entropy* of a commit message with respect to a language model in order to quantify its unnaturalness. To determine the quality of a commit, we use its *build status* as provided by Travis CI [157], a continuous integration service popular among open source projects. The build status of any commit can then be evaluated with respect to the unusualness of its message as measured by *cross-entropy*.

To ensure that the commits indeed come from software development projects, we employed a number of strategies described in Section 3.2.1. We describe how commit quality is determined using data from Travis CI in Section 3.2.2. In Section 3.2.3, we describe how each commit message was *tokenized* so that they could be used as input for the *n*-gram language models. Finally, in

Section 3.2.4, we describe how the tokenized commit messages were used to train language models, and how these language models were used to calculate each commit's unnaturalness via cross-entropy.

3.2.1 How were commits chosen?

To obtain commit messages applicable for this study, we used Boa [44] to query the September 2015 GitHub dataset. The Boa query and its results are available online.[2] This query used the following criterion to establish if a given project was to be considered in the study.

1. Boa must have parsed abstract syntax trees of the projects. As of this writing, Boa only parses Java code, meaning that only projects that contained parsable Java code were obtained. Thus, repositories that are unlikely to be used for software development are pruned.
2. The project must have more than 200 abstract syntax tree nodes. This filters out stub projects.
3. The project must have more than 6 commits, to avoid personal and other stub projects, following the recommendations given in "The Promises and Perils of Mining GitHub" [85].
4. Finally, the project must have a file named `.travis.yml`, indicating that project uses Travis CI to track per-commit build status.

From each project that passed the above criteria, commits were chosen only if the commit had an associated Travis CI status (described in Section 3.2.2), and if it was not a *merge*. Merge commits were excluded due to their messages being automatically generated by Git by default. Auto-generated merge commits are identifiable by their message starting with "Merge branch", "Merge pull request", or "Merge remote." The message text was used to detect merge commits since commit parent information is not available in Boa. Were this data available, merges would be detectable if the commit has more than one parent.

After filtering, 120,822 commits from 2,679 projects that fit the criteria described above and were used in our analysis.

3.2.2 Establishing commit quality: Travis CI

To establish the quality of a commit, we mined Travis CI. travis-ci.org is an online *continuous integration* service that is free for use by open source projects. When a commit is pushed to any branch on GitHub—be it the main branch, a derivative branch, or a pull request—Travis CI will

[2] http://boa.cs.iastate.edu/boa/?q=boa/job/public/30188

clone, build, and test that project's commit in a clean virtual machine or Linux container. The *install* phases sets up the machine by installing the project's dependencies, and populates test databases, if any. The *script* phase follows, in which the project is built (compiled) and its test suites are run. A Travis CI build may result in one of these statuses [156]:

errored An error occurred in the *install* phase. For example, a Java project using the Maven build system [108] may *error* due to a mis-configured `pom.xml` file.

failed An error occurred in the *script* phase. This usually means the project either failed to build or was successfully built, but failed its test suite.

passed The project built successfully and passed its tests.

A build may also be manually *cancelled* by a developer, but such commits were omitted from our analysis.

3.2.3 Tokenization

We used *n*-gram language models which use *tokens* as input. A token is an indivisible unit of meaning that makes up a message. The process of *tokenization* transforms raw text—a series of Unicode code points—into a series of tokens which can then be used with a language model. To avoid undue surprise, tokens fed to the language model must accurately represent the notion of "unnaturalness" defined in this thesis. Thus, we performed the following steps to tokenize each commit message.[3]

1. First, the message text is normalized using Unicode normalization form C (NFC) [151] to ensure that character sequences that "look the same" compare equal. That is, there exists many character sequence that look visually indistinguishable to human eyes when rendered on a screen, but are actually represented using different sequences Unicode code points are equal. We normalized the text using NFC form to remove such inconsistencies.
2. The message text is transformed into lower-case.
3. The text is then split on whitespace and separating punctuation.
4. Tokens with similar *meaning* are substituted with generic tokens. Upon manually observing over 2500 commit messages, a number of patterns were observed in their text. For example, one commit would read "Update README.md"; another would read "Update pom.xml". Such messages are *usual* or *natural*, yet the amount of individual variety may be immense, due to the

[3] https://github.com/eddieantonio/judging-commits/blob/msr2016/tokenize_commit.py#L425

NFC Normalized	Updating Manifest.txt. Related to f75a283 (#495)					
Lower-cased	updating manifest.txt. related to f75a283 (#495).					
Split	updating	manifest.txt	related	to	f75a283	#495
Substitutions	updating	FILE-PATTERN	related	to	GIT-SHA	ISSUE-NUMBER

Table 3.1: Each step of the tokenization process.

Substitution	Meaning	Examples
ISSUE-NUMBER	GitHub issue numbers [54]	#22, #34
FILE-PATTERN	Filenames and globs	README.md, **/*.jpg
METHOD-NAME	Method names	build_indexes()
VERSION-NUMBER	Version numbers[4]	2.2, 2.1.0-rc.1, 0.1-SNAPSHOT
GIT-SHA	SHA1 commit hashes	d670460

Table 3.2: Examples of semantic substitutions.

amount of different filenames possible. To capture this and other regularities, tokens that were similar were substituted with a generic token such as FILE-PATTERN. I defined and employed many such *semantic substitutions*, as demonstrated in Table 3.2.

Table 3.1 shows the tokenization applied to the message, "Updating Manifest.txt. Related to f75a283 (#495)." [38]

3.2.4 Training the n-gram language model

For each project in the dataset, I trained an n-gram language model on the commit messages of all *other* projects in the dataset. This is called the "leave-one-out" method. Excluding each project from its training set ensures that commit messages from the current project are not already "known" to the language model, thus biasing their "unnaturalness" score. The *order* or n of the n-gram model was set to three. n-gram models with an order of 3 are also known as *trigram* models. A trigram model was chosen because predictive performance on English text does not significantly improve for larger values of n [74].

The n-gram implementation used was MITLM [78], which implements modified Kneser-Ney smoothing to interpolate probabilities in the very likely case of data sparsity. With Kneser-Ney smoothing, the model is not "infinitely surprised" when it encounters a trigram it has never seen before—a trigram that is actually composed of one or more bigrams (2-gram) or unigrams (a single

[4] According to Semantic Versioning [126]

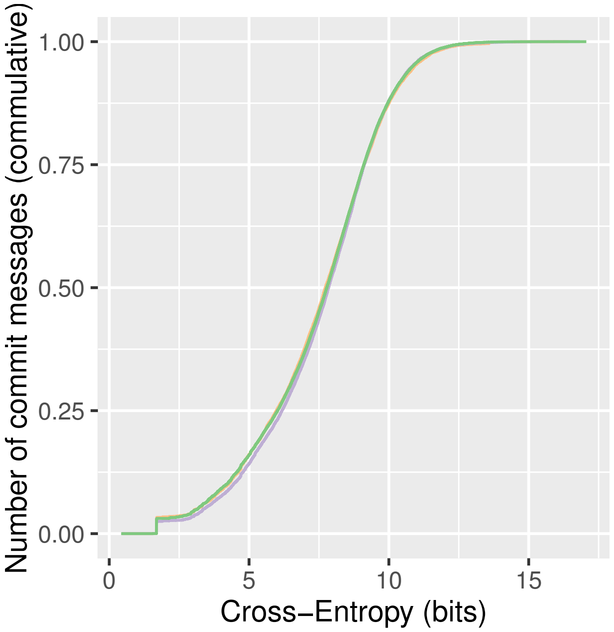


Figure 3.1: Empirical cumulative distribution function (ECDF) of the number of passed (in green), failed (in purple), and errored (in orange) commits as cross-entropy ("unnaturalness") increases. Note that failed, initially grows slower than passed and errored; by 10 bits, however, failed is indistinguishable from passed and errored.

token) that it *has* seen before. Instead, Kneser-Ney smoothing interpolates the probability with a penalty—that is, the model is surprised, but not too surprised.

To quantify the "unnaturalness" of each commit message, the text of each message is tokenized and evaluated using MITLM to calculate the *mean cross-entropy* of the commit message with respect to a language model trained on all other projects. That is, the cross-entropy of each trigram in a message is calculated with respect to the leave-one-out language model and averaged to produce the cross-entropy of the entire message. Intuitively, cross-entropy measures how much information a distribution—such as an *n*-gram language model—needs to explain an observation. The higher the cross-entropy, the more difficult it is for the model to explain a given observation. **The higher the cross-entropy a commit message has, the more unnatural it is**.

3.3 Results

Figure 3.2, left, is a histogram displaying the cross-entropies of all 120,822 commits. The width of the bins in the histogram were chosen using the Freedman-Diaconis rule [50]. What is instantly notable

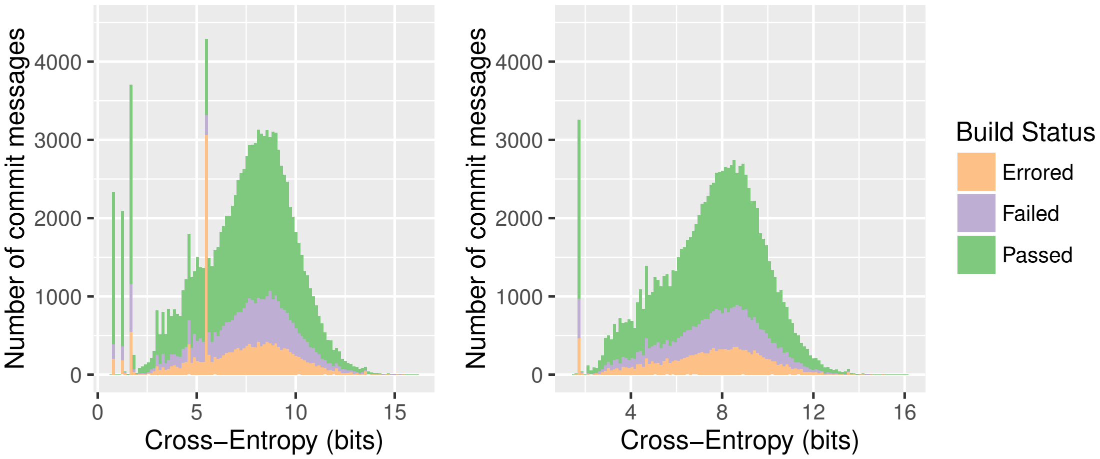


Figure 3.2: Histograms of commit message cross-entropies. Note the tall bins (left), which contain a large number of auto-generated commit messages that were not foreseen when training this model. We recalculated the histogram (right), removing auto-generated commits, as well as many non-English commit messages.

from this histogram are the four major "outlier" bins; in general, the distribution of cross-entropies tends to follow a somewhat normal distribution; however, these bins (quite literally) stick out.

Examining the messages in these bins revealed that, despite the previous discipline in removing merge commits due to their automatically generated commit messages, these automatic messages still crept in. With the exception of the second largest bin, which contains the messages of the form "Update FILE-PATTERN", the messages in these outliers are caused by tools which automatically make commits. Specifically, the Maven release plug-in, and its clone, the Gradle release plug-in both, of which generate messages of the form, "[maven-release-plugin] prepare for next development iteration" [138]. The largest outlier bin was filled with 2,880 messages similar to "GS-803 Build version advanced to 1150111 git-svn-id: svn://pc-lab14/SVN/xap/trunk/openspaces@170939 eb64e7373616-4df089415ee2ae88103d" [121]. These commits were automatically generated by a bot that modifies build configuration; most of these commits had a build status of errored.

Lesson: Not all commit messages are written by hand; even non-merge commits may be automatically generated.

We were curious about "very unnatural" messages—those belonging to the rightmost bins in Figure 3.2, left. Manual inspection of these bins revealed that the majority of the messages in the right-tail were not written in English. Since most of the corpus is in English, the language models would report commits in other languages as very unnatural indeed; additionally, this would affect our distribution and hypothesis, since a cursory glance of Spanish and Polish commits message show that they have a similar terseness and grammatical structure as “usual” English commits.

Lesson: commit messages on GitHub cannot be assumed to be written in English.

To determine if the cross-entropies of passed, failed, and errored commits had different distributions, we calculated pairwise comparisons using the Wilcoxon rank sum test. Letting $\alpha = 0.01$, we obtained a significant p-value near zero for all comparisons of passed vs. failed, passed vs. errored, and errored vs. failed, meaning that the cross-entropy distributions differed depending on build status.

However, we were suspicious of the effects of the outlier bins; note that the tallest bin visually seems to have a disproportionate amount of errored results. Thus, we repeated the pairwise Wilcoxon rank sum test with all five outlier bins *removed.* Only passed vs. failed and passed vs. errored distributions resulted as significantly different with a p-values near zero and 0.0021, respectively. Errored vs. failed, were not significantly different ($p = 0.2518$). This means that the cross-entropy distributions of passed vs. “broken” (either errored or failed) may be different. Still, since the bins were merely removed from the significance tests *after* the models had already been trained, the cross-entropies analyzed in these tests are still affected by commit messages in the outlier bins.

Given the significant effect of outliers and the fact that the tail had commits in languages other than English, it became evident that the leave-one-out methodology must be recalculated without such confounding factors.

We curated a list of auto-generated commits, and omitted such commits when constructing the new corpus. To filter for English-language commits, we used `langid.py` [103] to estimate the probability that a whole project’s commits are in English. We found that in isolated cases, `langid.py` would misclassified English commit messages (often as German or Dutch). Hence, we assisted it by manually verifying its results of over 100 projects, letting `langid.py` classify the rest using its default configuration. The resultant corpus contained 108,989 commits from 2,529 projects.

The results are in Figure 3.2, right. One outlier remains, which (as in the previous histogram) is filled with messages of the form “Update $\boxed{\text{FILE-PATTERN}}$”. A portion of these may be auto-generated, however we deemed these messages as plausibly hand-written; we did not discard them. Pairwise Wilcoxon rank sum test was retried and we found that passed vs. failed are still different distributions with a p-value near zero; similarly, failed vs. errored are different with a p-value of 0.0015. Given a p-value of 0.7527, we fail to reject passed vs. errored as different distributions. We plotted the ECDF of passed, errored, and failed commits (Figure 3.1). Strikingly, we found that failed *is* different, for lower values of cross-entropy. Additionally, it steadily closes the gap between itself and passed and errored. This means that failure rate is *lower* given a more usual commit, and gradually increases. Calculating Pearson's product-moment linear correlation coefficient yields a 99% confidence interval of $(0.007, 0.468)$. Since zero is not in the interval (zero would indicate no correlation) we conclude that build failure and "unnaturalness" may be positively correlated—but only marginally.

3.4 Discussion

Does this mean that developers can use commit messages to predict build failure? Can we cancel our continuous integration subscriptions? In short, no.

Though the results are statistically significant, we conclude that they are not *practically helpful* for the common developer. For example, which of the following commits failed its status check? "added init.d test to travis config" [36] (cross-entropy = 5.08), or "I'm sloppy" [70] (cross-entropy = 12.9)? The latter has a far more unnatural commit message than the former, yet it passed its status check; the "usual" commit failed. Thus, as a heuristic for estimating the probability of build failure, the unnaturalness of a commit message is not very useful.

For future work, we would like to investigate other features correlation with build status; are there any non-code features that *do* predict build failure?

3.5 Conclusions

RQ1: **How can the unnaturalness of a commit message be measured?** Using n-gram language models, one is able to use cross-entropy as an analogue for unnaturalness. Automatically-generated messages and non-English commits become easy to spot.

RQ2: **Is the unnaturalness of a commit message related to the quality of the code committed?** "Never judge a book by its cover." Similarly, never judge a commit by its log message. Despite some evidence to suggest that the "unnaturalness" of a commit message is positively correlated with build failure, the slope is so gradual that it is infeasible for an average developer to judge a commit by simply reading its log message.

Chapter 4

PartyCrasher: scalable crash report deduplication

In this chapter, I present *naturalness* as applied to crash report repositories, for the purpose of grouping together individual crashes into clusters of similar crashes ("buckets") that, ideally, are caused by the same underlying fault. To perform such bucketing, we employed techniques borrowed from information retrieval for unstructured, natural language text documents. In contrast to the previous chapter, which tokenized embedded snippets of structured artificial languages from natural language utterances, this chapter experiments with different tokenization strategies that best represent the semi-structured crash reports for the purposes of grouping similar crash reports together.

What to takeaway from this chapter

- systems built for natural language information retrieval (e.g., ElasticSearch/Lucene) are well suited for online deduplication of semi-structured data
- not all tokenization schemes compare equal, so experiment with tokenization
- the trade-off between precision and recall can be resolved at "runtime" rather than be an intrinsic property of the deduplication algorithm

Readers should refer to Figure 4.2 as to how our solution will integrate in a development context.

This chapter is relevant to anyone interested in large-scale clustering or deduplication of semi-structured data. Due to fact that our implementation (PartyCrasher) is open source, future work can freely use PartyCrasher as a baseline for future crash report deduplication software, as was done by Moroo et al. [112]. However, I recommend our approach more broadly to any researchers clustering or deduplicating artifacts that combine structured data from the world of software with unstructured natural language text. This includes finding duplicates of bug reports, as well as duplicates of StackOverflow [145] questions. PartyCrasher has been cited [31] as a pre-requisite step to crash cluster analysis—finding the root cause of a cluster of crashes.

My personal contributions in this chapter include a hand in the implementation of the PartyCrasher; a literature review which aggregated existing work from academia and industry alike; creation of diagrams, and manuscript composition.

This chapter is adapted from "The unreasonable effectiveness of traditional information retrieval in crash report deduplication", published [29] at the 13th International Conference on Mining Software Repositories (MSR), 2016, and has contributions from Hazel Campbell as an equal coauthor, and Dr. Abram Hindle with manuscript composition.

Abstract

Organizations like Mozilla, Microsoft, and Apple are flooded with thousands of automated crash reports per day. Although crash reports contain valuable information for debugging, there are often too many for developers to examine individually. Therefore, in industry, crash reports are often automatically grouped together in buckets. Canonical's repository contains crashes from hundreds of software systems available in Ubuntu. A variety of crash report bucketing methods are evaluated using data collected by Ubuntu's Apport automated crash reporting system. We explore the trade-off between precision and recall of numerous scalable crash deduplication techniques. We present a set of criteria that a crash deduplication method must meet and we evaluate several methods that meet these criteria on a new dataset. The evaluations presented in this chapter show that using off-the-shelf information retrieval techniques, that were not designed to be used with crash reports, outperform other techniques which are specifically designed for the task of crash bucketing at realistic industrial scales. This research indicates that automated crash bucketing still has a lot of room for improvement, especially in terms of identifier tokenization.

4.1 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 saying that the software she just shipped has a crashing bug affecting two-thirds of all users. Worse yet, Ada and her team can't 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* [135], with some *context* information about what machine encountered the crash, and a *stack trace* [135] from each thread. Developers consider stack traces to be an indispensable tool for debugging crashed programs—a crash report with even one stack trace will help fix the bug significantly faster than if there were no stack traces available at all [134].

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 [113], Microsoft's Windows Error Reporting (WER) [55], or Apple's Crash Reporter [13]. 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 [1].

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 crash reports are actually relevant to the bug Ada is trying to fix?

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 stack traces from this *crash bucket* [55], and get on with debugging her application. Since this hypothetical bucket has all crash stack traces caused by the same bug, Ada could analyze any number of stack traces and pinpoint exactly where the fault is and how to fix it.

[1] https://crash-stats.mozilla.com/api/SuperSearch/?date=%3E%3D2016-01-01&date=%3C%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 questions that this chapter seeks to answer are:

RQ1: Do tf-idf-based methods make for 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, 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.

4.1.1 Contributions

This chapter presents PartyCrasher, a technique that buckets crash reports. It extends the work done by Lerch and Mezini [96] to the field of crash report deduplication and show that despite its simplicity, it is quite effective. This chapter contributes:

1. a criterion for industrial-scale crash report deduplication techniques;
2. replication of some existing methods of deduplication (such as Wang et al. [160] and Lerch and Mezini [96]) 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 deduplication method that outperforms other methods when contextual information is included along with the stack trace.

4.1.2 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* [106]. Algorithms that run in $O\left(n^2\right)$ are infeasible for the increasingly large amount of crash reports that need to be bucketed. Therefore, an absolute upper-bound of $O\left(n \log n\right)$ 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 deduplication requirements. Therefore, it must run in $O\left(n \log n\right)$ total time. Equivalently, each new, incoming crash must be able to be assigned a bucket in $O\left(\log n\right)$ 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 deduplication methods 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.

```
#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
```

Figure 4.1: An example stack trace.

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`

Table 4.1: Types of signatures.

Method	Tokenization
No tokenization	`#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`

Table 4.2: Different ways of tokenizing the signature.

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 4.1 shows an example stack trace; Table 4.1 demonstrates how the various signatures are extracted from Figure 4.1 using the signature generation methods described; and Table 4.2 demonstrates the different tokenization strategies applied to the first stack frame in Figure 4.1. All of the signature-based methods, as implemented, run in $O(n \log n)$ total time or $O(\log n)$ amortized time per crash.

The second category of methods are those based on tf-idf [131] and inverted indices, as implemented by the off-the-shelf information-retrieval software ElasticSearch 1.6 [46]. tf-idf is a way to normalize a *token* based on both on 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.

4.2 Background

Of course, the idea of crash bucketing is not new; Mozilla's system performs bucketing [42, 1], as does WER [55]. Many approaches make the assumption that two crash reports are similar if their stack traces are similar. Consequently, researchers [25, 99, 111, 19, 55, 42, 40, 160, 96, 163] have proposed various methods of finding similar stack traces, crash report similarity, crash report deduplication, 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 stack trace [25, 134, 163]. As such, many bucketing heuristics employ higher weighting for grouping functions near the top of the stack [111, 55, 160]. 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 stack trace [96]. The most influential discriminative factors seem to be function name [96] and module name [19, 55].

Lerch and Mezini [96] did not directly address crash report bucketing; they addressed *bug report* deduplication through stack trace similarity. They deduplicated bug reports that included stack traces 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-deduplication, the authors did not compare this technique against the other crash report deduplication techniques. 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 Lerch and Mezini [96]'s method of bug report deduplication to crash report deduplication, both excluding *contextual* data from the crash report as suggested by Lerch and Mezini [96] 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 stack trace. 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 [46] 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 stack trace along with the stack trace itself. Table 4.2 shows how each method tokenizes a sample stack frame.

Modani et al. [111] provide 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. did not use tf-idf in their evaluation). The implementation is provided by ElasticSearch 1.6 [46]'s 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.

4.2.1 Methods not appearing in this chapter

Mozilla's deduplication technique, at the time of writing, as it is implemented in Socorro [114] 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 dataset contains crashes from 616 other systems.

In 2005, Brodie et al. [25] 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 $O\left(n^2\right)$–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 [99] grouped crashes together if they suggest the same fault location. The fault locations were found using a statistical debugging tool called SOBER [100], that, trained on failing and passing *execution traces* (based on instrumenting Boolean predicates in code [98]), returns a ranked list of possible fault locations. Methods involving full instrumentation [99] or static call graph analysis [163] 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. [88] and Wu et al. [163] are disregarded as well.

Modani et al. [111] propose several algorithms. The first algorithm employs edit distance, requiring $O\left(n^2\right)$ 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 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. [111] comparison algorithms were used.

In addition to comparison algorithms that might be used for deduplication directly, Modani et al. [111] 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. [19] also used edit distance on the stack trace, 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 stack trace. 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 stack trace's importance.

The methods based on edit distance—*viz.*, Brodie et al. [25], Modani et al. [111], Bartz et al. [19][2]—are disqualified due to their requirement of pairwise comparisons between stack traces, with an upper-bound of $O\left(n^2\right)$.

Schröter et al. [134] empirically studied developers' use of stack traces in debugging and found that bugs are more likely to be fixed in the top 10 frames of their respective crash stack trace, 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. [163].

Glerum et al. [55] describe the methods used by Microsoft's 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. [42] proposed a weighted edit distance technique that creates *representative stack traces*—a probability distribution based on all stack traces seen within a bucket. Thus, instead of computing similarity against all stack traces in a bucket, one would only use the weights derived from all stack traces in the bucket simultaneously.

The method described in Dhaliwal et al. [42] is not included in the evaluation because it first subdivides buckets produced by the `1Frame` deduplication method, 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 too many buckets. Despite the optimization in Dhaliwal et al. [42] 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

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

at a slower rate, this method will eventually become computationally unfeasible if old data is not discarded.

Kim et al. [88] 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. [40] 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. [160] 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. [160] 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 [96].

4.3 Methodology

First, the requirements for an industrial-scale automated crash deduplication system 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 deduplication.

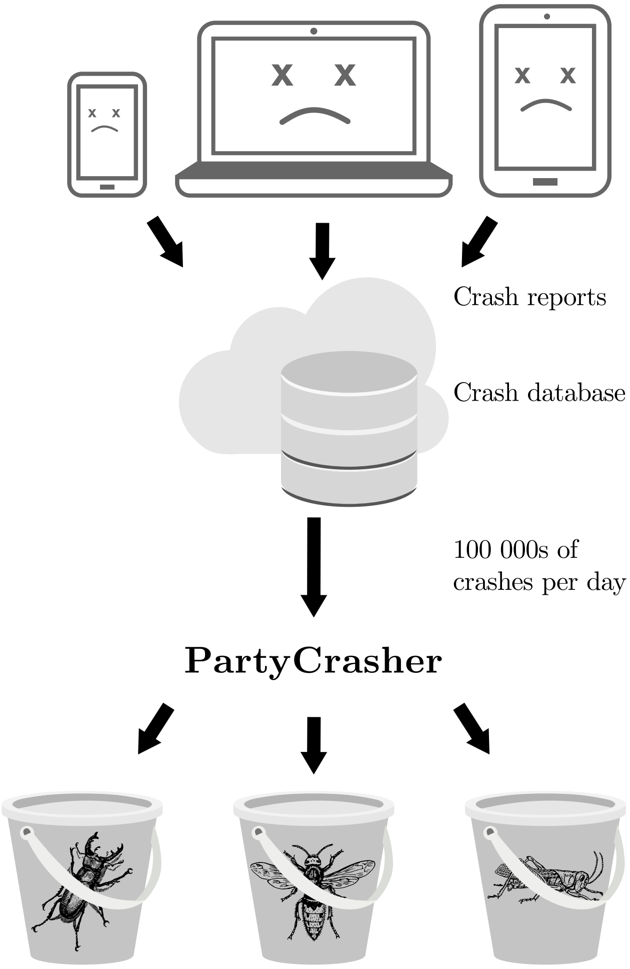


Figure 4.2: PartyCrasher within a development context.

Several methods that met the requirements were selected. A general purpose Python framework in which any of the selected deduplication methods 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 dataset that could be used as a gold set to judge the performance of such methods was obtained. The dataset was then filtered to include only crash reports that had been deduplicated by 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 4.3

```
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 4.3: An example crash report, including stack [11].

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 stack trace on the bottom.

4.3.1 Mining crash reports

The first step in the evaluation procedure is mining of crash reports from Ubuntu's bug repository, Launchpad [30]. This was done using a modified version of Bicho [130], a software repository mining tool.[3] Over the course of one month, Bicho was able to retrieve 126,609 issues from Launchpad,

[3] Available: https://github.com/orezpraw/Bicho/. The changes were to augment Bicho's the Launchpad backend: to download crashes only; to include fields relating to whether the crash was a duplicate of another one; and to store the original Launchpad ID.

including 80,478 stack traces in 44,465 issues. Some issues contain more than one stack trace. For issues that contained more than one stack trace, the first stack trace posted to that issue was selected, yielding 44,465 issues with crash reports and stack traces. The first stack trace 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 deduplicated by Ubuntu developers and volunteers. Other data sources, such as Mozilla's Crash Reports have already been deduplicated by Mozilla's own automated system, not by humans.

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 this chapter, to prevent confounding with groups of crash reports, that will be referred to as "crash buckets." This dataset is available.[4]

4.3.1.1 Stack trace extraction

Each issue and stack trace obtained from Ubuntu is formatted as plain text, as shown in Figure 4.3. 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 stack traces contain unintelligible binary data in place of the function name. This could be caused by memory corruption when the stack trace was captured. 2,216 crash reports and stack traces were thrown out because their formatting could not be parsed, leaving 41,708 crash reports with stack traces.

4.3.1.2 Crash report and stack trace data

Issues were then filtered to only those that had been deduplicated by Ubuntu developers and other volunteers, yielding 15,293 issues and stack traces in 3,824 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

[4] https://archive.org/details/bugkets-2016-01-30

[5] https://launchpad.net/apport

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 system is Gnome[6], which has 2,154 crash reports with stack traces. This dataset is large, comprehensive and covers a wide variety of projects.

4.3.2 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.

4.3.3 Deciding when a crash is not like the others

For methods based on Lerch and Mezini, there is a threshold value, T, that 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, 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 [46]. Details of this tf-idf-based scoring method are described within the ElasticSearch documentation [102]. The scoring algorithm is based on tf-idf, but contains a few minor adjustments intended to make scores returned from different queries more comparable.

4.3.4 Implementation

The complete implementation of the evaluation presented in this chapter is available in the open-source software PartyCrasher.[7] The implementation includes every deduplication method we claimed to evaluate above, a general-purpose deduplication framework, 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.

[6] https://www.gnome.org/

[7] https://github.com/naturalness/partycrasher

4.3.5 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. [9]. The mathematical formulae for both metrics can be found in Amigó *et al.* [9]. However, purity also has an advantage over BCubed: specifically that it does not require $O\left(n^2\right)$ total time to compute whereas BCubed does during the evaluation.

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 recall 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 two buckets that overlap the most. Thus, if a method creates a bucket that is 51% composed of crashes from a single bug, the other 49% doesn't matter. That 49% could come from a different bug, or 200 different bugs, but the purity would be the same value. It is included in this evaluation for completeness, since it was used by Dang et al. [40].

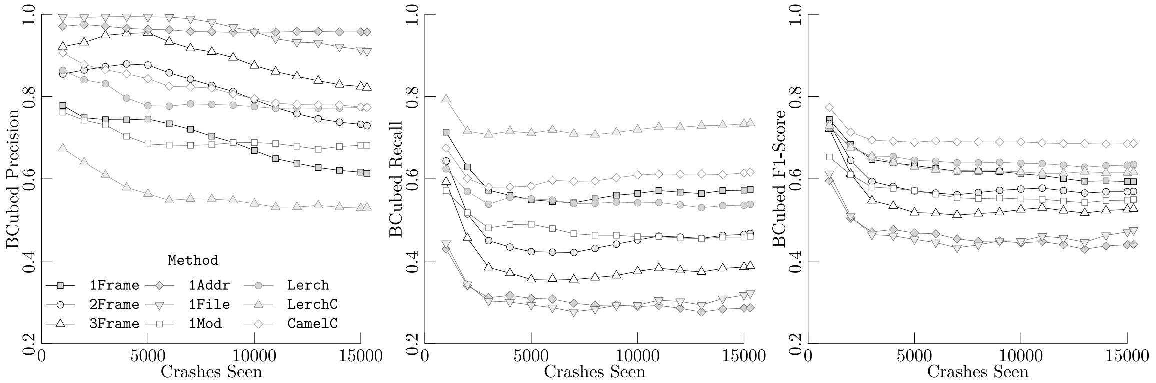


Figure 4.4: BCubed (top) and Purity-metric (bottom) scores for various methods of crash report deduplication.

Both metrics can be combined into F-scores. In this evaluation, F_1-scores were used, placing equal weight on 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 [30] 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 dataset is biased to the distribution of crashes that are bucketed, which might be different than crashes that are not. Conversely, this prevents the evaluation dataset from containing any crashes that have not yet evaluated by an Ubuntu developer or volunteer.

4.4 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.

4.4.1 BCubed and purity

Evaluation of the performance of bucketing algorithms is performed with BCubed and purity metrics. Figure 4.4 shows the performance of a variety of deduplication methods evaluated against the entire

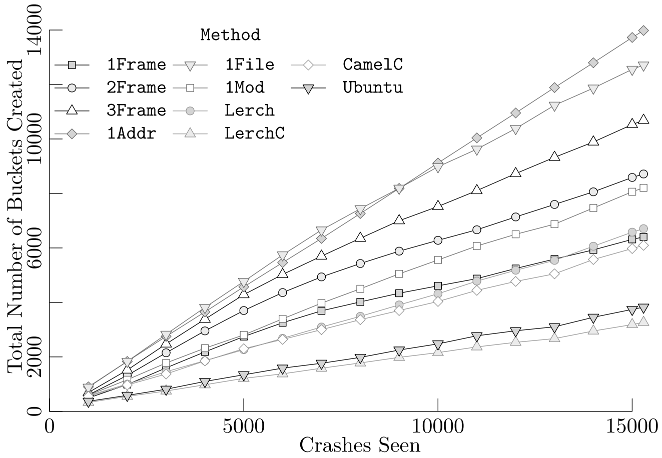


Figure 4.5: Number of buckets created as a function of number of crashes seen. The line labelled `Ubuntu` indicates the number of groups of crashes that were marked as duplicates of each other by Ubuntu developers or volunteers.

gold set of deduplicated crash 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 [96], 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.

Amigó et al. [9] observed differences in BCubed and purity metrics. Their observation was tested empirically by the evaluation. In Figure 4.4, BCubed and purity produce 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 4.4 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.

4.4.2 Bucketing effectiveness

Figure 4.5 shows the number of buckets created by a variety of deduplication methods. The number of issue buckets extracted from the Ubuntu Launchpad gold set is plotted as the line labelled `Ubuntu`. The method that creates a number of buckets most similar to the number mined from the Ubuntu Launchpad gold set is `LerchC`. For the `Lerch` and `LerchC` simulations, a threshold of $T = 4.0$ was used.

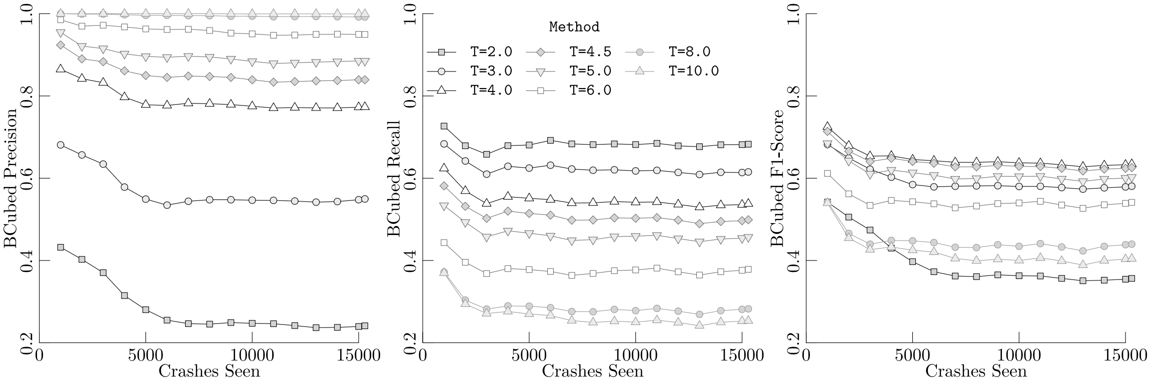


Figure 4.6: BCubed scores for the Lerch method of crash report deduplication at various new-bucket thresholds T.

Figure 4.6 shows the performance of the `Lerch` method when used with a variety of different new-bucket thresholds, T. Figure 4.7 shows the number of buckets created by the same method with those same thresholds. Since Lerch and Mezini [96] 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 5,000 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.

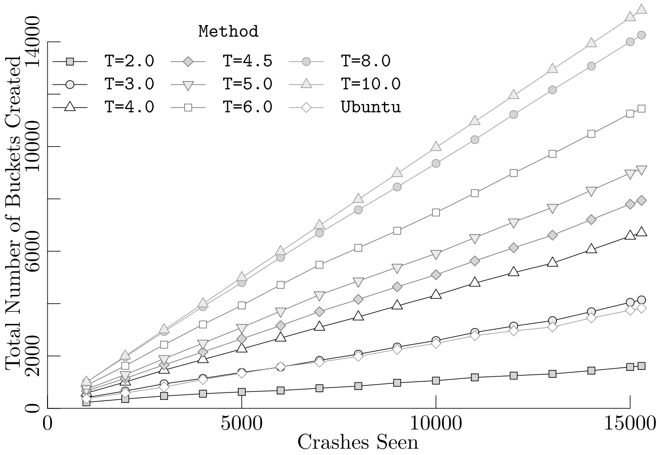


Figure 4.7: Number of buckets created as a function of number of crashes seen for the Lerch method of crash report deduplication at various new-bucket thresholds T. The line labelled `Ubuntu` indicates the number of groups crashes that were marked as duplicates of each other by Ubuntu developers or volunteers.

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 5,000 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 4.8, $T = 4.0$ 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 4.8 also shows how the threshold can be tuned to create a trade-off between precision and recall. 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. 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 dataset. Instead of using data, one could ask developers if they had seen too many crashes caused by unrelated bugs in a single bucket recently. If they had, then T should be increased. Or, T should

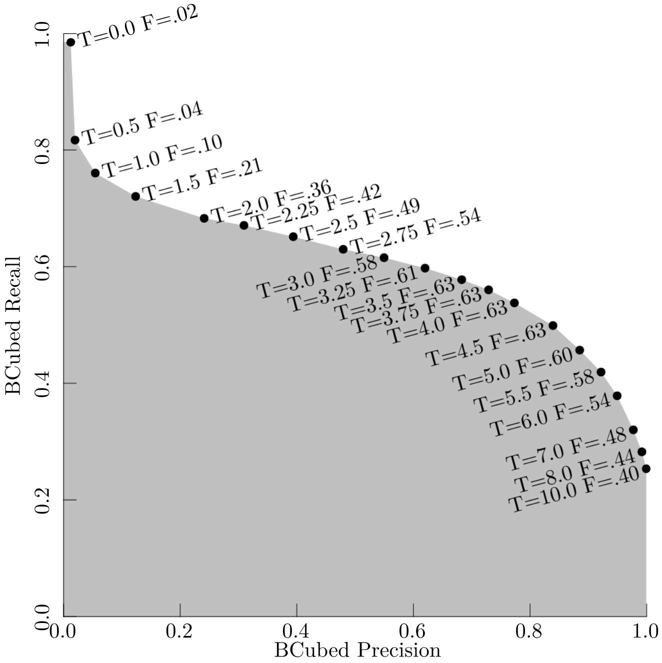


Figure 4.8: Precision/Recall plot showing the trade-off between BCubed precision and recall as the new-bucket threshold T is adjusted. BCubed F$_1$-score is also listed in the plot.

be decreased if developers see multiple buckets that seemed to be focused on crashes caused by the same bug.

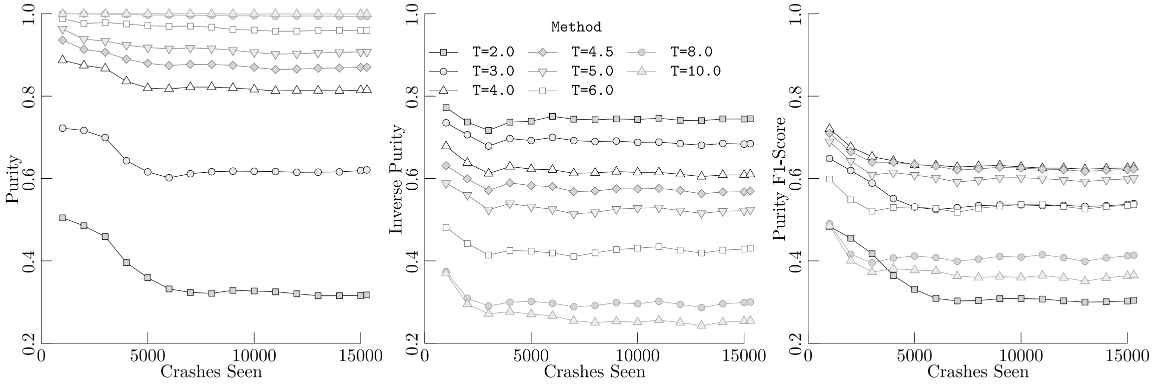


Figure 4.9: Purity-metric scores for the Lerch method of crash report deduplication at various new-bucket thresholds T.

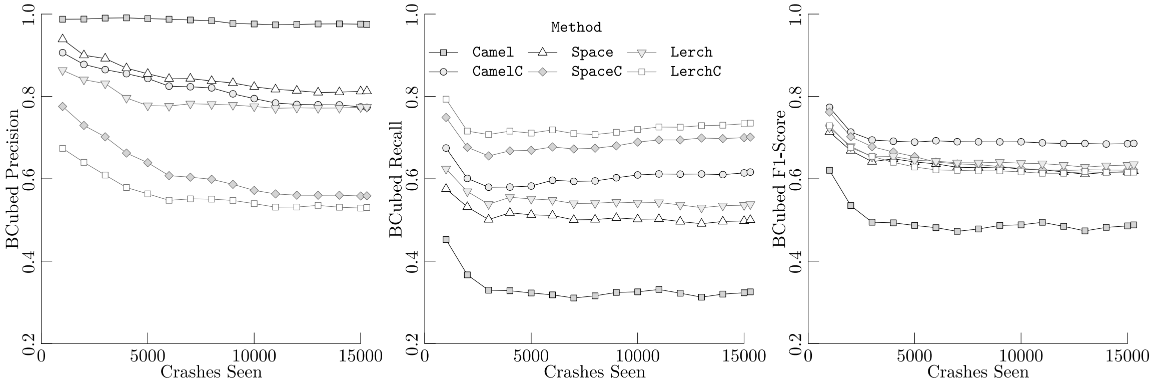


Figure 4.10: BCubed scores for the Lerch method of crash report deduplication with Lerch's tokenization technique replaced by a variety of other techniques.

4.4.3 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 [96], but do not follow their tokenization strategy. The performance of several tokenization strategies is shown in Figure 4.10. 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 4.10 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 stack traces 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 [47]. 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 $T = 4.0$ were not the best-performing threshold or method, but both produced almost the same number of buckets as the gold set.

4.4.4 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® Core™ 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.

4.5 Discussion

4.5.1 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 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 stack traces. However, these results are corroborated by studies that examined Java exclusively [160, 96].

4.5.2 Related work

Although crash bucketing facilitates manual debugging of individual faults, crash buckets are much more beneficial as the input to other methods in software engineering. Lerch and Mezini [96] originally applied their technique to the field of deduplicating bug, not crash, reports; Khomh et al. [86] used crash buckets to triage bugs: prioritizing developer effort on the most crucial bugs. Seo and Kim [135] 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 [99, 160, 163] and crash visualization [88, 40].

4.5.3 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 deduplication methods.

Many stack comparison methods [111, 55, 42, 160], 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 deduplication presented 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 [25, 134, 163]. 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 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. [63] and Hill et al. [72], would likely outperform the basic techniques that have been evaluated in this chapter.

As shown in Figure 4.3, crash reports often contain a multitude of data apart from the stack trace itself. This chapter only measured the performance of tf-idf when using only the stack trace 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 deduplication than `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. [25], 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 [86] and crash localization [163]. We also wish to investigate whether post-processing can be applied to the buckets produced by our method to minimize the "second bucket problem" as discussed by Glerum et al. [55] and Dang et al. [40].

4.6 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 deduplication method with tokenization that fits their dataset, and intermediate new-bucket threshold. They should update this threshold based on feedback from developers, volunteers, or employees that work with the stack traces directly. A tf-idf approach that used the entire crash report and stack trace, 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 stack traces, 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: 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 deduplicated bug reports that were submitted with crash reports. Funding for this research was provided by MITACS Accelerate with BioWare™, an Electronic Arts Inc. studio.

Chapter 5

Syntax and *Sensibility*

In this chapter, we apply the concept of *naturalness* to the problem of source code syntax error correction in two ways: we flag quantifiably unlikely utterances in source code ("unnatural" code) as potential syntax errors; and we use the *naturalness* of software to search for the appropriate edit that will fix the error. As in Chapter 3 ("Judging a commit by its cover"), we investigate *unnatural* utterances and apply application-specific tokenization to drastically reduce the amount of tokens that only appear once in the corpus. Like Chapter 4 ("The unreasonable effectiveness of traditional information retrieval in crash report deduplication"), we assume that new source code is similar to previously seen source code, thus, we can apply traditional natural language processing (NLP) techniques to find a suitable fix for the syntax error.

The takeaways from this chapter are:

- Syntax can be modeled by abstracting "open" tokens.
- Machine learning on "professional" code can be applied to novice code; there is no need to constrain a technique to train and evaluate only on novice code.
- Forwards/backwards long short-term memory (LSTM) language models can outperform n-gram models on the task of automatic syntax error correction.
- Given a search space of fixes, *naturalness* can be used to suggest a fixed number of possible fixes, thus greatly reducing the search space when compared to more brute-force approach.

Readers should pay particular attention to Figure 5.3. Looking at the results of "LSTM Impr." on creating a valid fix, we see that its number 1 suggestion will fix a syntax error over half of the time. This is a testament to *naturalness*'s ability to bias search.

This chapter is relevant to people looking to support novice programmers by creating tools that lower the barriers to learning how to code. This includes academics, but also developers of text editors and integrated development environments (IDEs). For example, statistics derived from corpora of correct code (in a manner described in this chapter) can be incorporated in the `PSIElement` interface of the IntelliJ IDE platform [83]. This information can guide the parser when it has encountered a syntax error to determine if there are more likely parses of the incorrect file, given a different series of tokens in the parse tree. This chapter is also interesting to those comparing the effectiveness of deep learning techniques (such as LSTM neural networks) compared to simpler machine learning algorithms such as *n*-grams. This chapter helps provide evidence to answer the question: is it viable to create a syntax error correction model that is general-purpose, and can be useful even when deployed on code from a domain that it was not trained on?

My personal contributions to this chapter were creating the syntax fixing algorithm (Section 5.5.2); devising the vocabulary abstraction technique (Section 5.3.2); training and experimenting with the LSTM models (discussed in greater detail in Appendix A); writing code and tests to ensure a correct implementation of all the algorithms mentioned in this chapter; gaining access to and mining the Blackbox database (Section 5.6); and manuscript composition.

This chapter is adapted from "Syntax and *Sensibility*: Using language models to detect and correct syntax errors", published [132] at the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, and has contributions from Hazel Campbell with concept formation, authoring the evaluation code, writing the *n*-gram implementation, and manuscript composition; Dhvani Patel with research assistance; Dr. Abram Hindle, and Dr. J. Nelson Amaral with concept formation and manuscript composition.

Abstract

Syntax errors are made by novice and experienced programmers alike; however, novice programmers lack the years of experience that help them quickly resolve these frustrating errors. Standard LR parsers are of little help, typically resolving syntax errors and their precise location poorly. We propose a methodology that locates where syntax errors occur, and suggests possible changes to the token stream that can fix the error identified. This methodology finds syntax errors by using language models trained on correct source code to find tokens that seem out of place. Fixes are synthesized by consulting the language models to determine what tokens are more likely at the estimated error location. We compare n-gram and LSTM language models for this task, each trained on a large corpus of Java code collected from GitHub. Unlike prior work, our methodology does not require that the problem source code comes from the same domain as the training data. We evaluated against a repository of real student mistakes. Our tools are able to find a syntactically-valid fix within its top-2 suggestions, often producing the exact fix that the student used to resolve the error. The results show that this tool and methodology can locate and suggest corrections for syntax errors. Our methodology is of practical use to all programmers, but will be especially useful to novices frustrated with incomprehensible syntax errors.

```
1 public class A {
2     public static void main(String args[]) {
3        if (args.length < 2)
4           System.out.println("Not enough args!");
5           System.exit(1);
6        }
7        System.out.println("Hello, world!");
8     }
9 }
```

Listing 5.1: Syntactically invalid Java code. An open brace (`{`) is missing at the end of line 3.

5.1 Introduction

Computer program source code is often expressed in plain text files. Plain text is a simple, flexible medium that has been preferred by programmers and their tools for decades. Yet, plain text can be a major hurdle for novices learning how to code [147, 80, 148]. Not only do novices have to learn the semantics of a programming language, but they also have to learn how to place arcane symbols in the right order for the computer to understand their intent. The positioning of symbols in just the right way is called *syntax*, and sometimes humans—especially novices—get it wrong [147].

The tools meant for interpreting the human-written source code, *parsers*, are often made such that they excel in understanding well-structured input; however, if they are given an input with so much as one mistake, they can fail catastrophically. What's worse, the parser may come up with a misleading conclusion as to where the actual error is. Consider the Java source code in Listing 5.1. A single *token*—an open brace (`{`) at the end of line 3—in the input is different from that of the correct file that the programmer intended to write. Give this input to a Java 8 compiler such as OpenJDK's `javac` [122], and it reports that there is an error with the source file, but it overwhelms the user with misleading error messages to identify the location of the mistake made by the programmer.

```
A.java:7: error: <identifier> expected
       System.out.println("Hello, world!");
                         ^
A.java:7: error: illegal start of type
       System.out.println("Hello, world!");
                          ^
A.java:9: error: class, interface, or enum expected
}
^
3 errors
```

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 [26], yet the compiler offers little help in resolving them.

In this chapter, we present Sensibility, which finds and fixes **single token syntax errors**. The general problem we seek to solve is:

Given a source code file with a syntax error, how can one accurately pinpoint its location and produce a single token suggestion that will fix it?

We compare the use of n-gram models and long short-term memory (LSTM) models for modelling source code for the purposes of correcting syntax errors. All models were trained on a large corpus of hand-written Java source code collected from GitHub. Whereas prior works offer solutions that are limited to fixing syntax errors within the same domain as the training data—such as only fixing errors within the same source repository [28], or within the same introductory programming assignment [65, 22]—As such, we evaluated against a corpus of real syntax errors collected from the Blackbox repository of novice programmers’ activity [27].

In this chapter, we focus on correcting syntax errors at the token level. We do not consider *semantic errors* that can occur given a valid parse tree such as type mismatches, and—in our abstract models (Section 5.3.2)—misspelled variable names. These errors are already handled by other tools given a source file with valid syntax. For example, the Clang C++ compiler [35] can detect misspelled variable names and—since it has a valid parse tree—Clang can suggest which variable in scope may be intended. As such, we focus our attention in this chapter to errors that occur *before* a compiler can reach this stage—namely, errors that prevent a valid parse of a source code file. In particular, we consider syntax errors that are non-trivial to detect using simple handwritten rules, as used in some parsers [82, 150].

Our contributions include:

- Showing that language models can successfully locate and fix syntax errors in human-written code in seconds without parsing.
- Comparing three different language models including two *n*-gram models and one deep neural network.
- Evaluating how all three models perform on a corpus of real-world syntax errors with known fixes provided by students.

5.2 Prior work

Prior work relevant to this research addresses repositories of source code and repositories of mistakes, past attempts at locating and fixing syntax errors in and out of the parser, methods for preventing syntax errors, deep learning, and applying deep learning to syntax error locating and fixing.

Data-sources and repositories are needed to train models and evaluate the effectiveness of techniques on syntax errors. Brown et al. [27, 7] created the Blackbox repository of programmers' activity collected from the BlueJ Java IDE [92], which is used internationally in introductory computer science classes. Upon installation, BlueJ asks the user whether they consent to anonymized data collection of editor events. The dataset—which is continually updated—enables a wealth of analyses concerning how novices program Java. Our empirical evaluation, described in Section 5.6, is one such analysis using the Blackbox data.

Syntax errors are errors whereby the developer wrote code that cannot be recognized by the language rules of the parser. It can be as simple as missing a semi-colon at the end of a statement. The long history of work on syntax error messages is often motivated by the need to better serve novice programmers [147, 80, 148, 81, 79, 51, 110, 93, 77]. Denny et al. [41] categorized common syntax errors that novices make by the error messages the compiler generates and how long it takes for the programmer to fix them. This research shows that novices and advanced programmers alike struggle with syntax errors and their accompanying error messages. Dy and Rodrigo [43] developed a system that detects compiler error messages that do not indicate the actual fault, which they name “non-literal error messages”, and assists students in solving them. Nienaltowski et al. [119] found that more detailed compiler error messages do not necessarily help students avoid being confused by error messages. They also found that students presented with error messages only including the file, line and a short message did better at identifying the actual error in the code more often for some types of errors. Marceau et al. [105] developed a rubric for grading the error messages produced by compilers. Becker’s dissertation [20] attempted to enhance compiler error messages in order to improve student performance. Barik et al. [18] studied how developers visualize compilation errors. They motivate their research with an example of a compiler misreporting the location of a fault. Pritchard [127] shows that the most common type of errors novices make in Python are syntax errors. Brown et al. [26] investigated the opinions of educators regarding what they believe are the most common Java programming mistakes made by novices, and contrasted it with mistakes mined from the Blackbox dataset. They found that educators share a weak consensus of what errors are most frequent, and furthermore show that across 14,235,239 compilation events, educators’ opinion demonstrates a low level of agreement against the actual data mined. The most common error witnessed was unbalanced parenthesis and brackets, which was ranked the eleventh most common on average by educators. The other most common errors were semantic and type errors.

Earlier research attempted to tackle errors at the parsing stage. In 1972, Aho [2] introduced an algorithm to attempt to repair parse failures by minimizing the number of errors that were generated. In 1976, Thompson [154] provided a theoretical basis for error-correcting probabilistic compiler parsers. He also criticized the lack of probabilistic error correction in then-modern compilers. However, this trend continues to this day.

Parr et al. [124] discusses the strategy used in ANTLR, a popular LL(*) parser-generator. This strategy involves an increased look-ahead strategy to produce error messages that take into account more of the surrounding code as context, and uses single-token insertion, deletion, substitution, and predictive parsing to produce better error messages The parser attempts to repair the code when it encounters an error using the context around the error so it can continue parsing. This strategy allows ANTLR parsers to detect multiple problems instead of stopping on the first error. Jeffery [82] created Merr, an extension of the Bison parser generator, which allows the grammar writer to provide examples of expected syntax errors that may occur in practice, accompanied with a custom error message. Merr automatically generates a `yyerror()` function that recovers the parser state in an error condition to determine if any of the given errors occurred, and displays the provided error message appropriately. While the grammar writer could compose an error message that suggests the fix to a syntax-error in Merr, our work is explicitly focused on providing the possible fix. In contrast to Merr, our work does not require any hand-written rules in order to provide a suggestion for a syntax error, and is not reliant on the parser state, which may be oblivious to the actual location of the syntax error.

Recent research has applied language models to syntax error detection and correction. Campbell et al. [28] created UnnaturalCode which leverages n-gram language models to locate syntax errors in Java source code. UnnaturalCode wraps the invocation of the Java compiler. Every time a program is syntactically-valid, it augments the existing n-gram model. When a syntax error is detected, UnnaturalCode calculates the *entropy* of each token. The token sequence with the highest entropy with respect to the language model would be the likely location of the true syntax error, in contrast to the location where the parser may report the syntax error. Using code-mutation evaluation, the authors were able to find that a combination of UnnaturalCode's reported error location and the Java compiler's reported error locations would yield the best mean-reciprocal rank for the true error location. Using a conceptually similar technique to UnnaturalCode, our work detects the location of syntax errors; unlike UnnaturalCode, our work can also suggest the token that will fix the syntax error.

Preventing syntax errors is the goal of research that wishes to reduce syntax errors or make syntax errors impossible to make, even in text-based languages. This is often accomplished by blurring the line between the program editor and the language itself and engaging in tree edits, such as the tree edit states of Omar et al. [120]'s proposed Hazel, or Project Lambdu [94] where an abstract syntax tree (AST) is modified within the editor instead of text.

Numerous compilers have placed a focus on more user-friendly error messages that explain the error and provide solutions. Among these are Clang [35], Rust [159], Scala [116], and Elm [37].

Deep learning is a technique to model token distributions on software text [129, 162]. Recurrent neural networks (RNNs), unlike feedforward neural networks, have links between layers that form a directed cycle, effectively creating a temporal memory. RNNs have been successful in speech recognition [66]. LSTM neural networks, extend RNNs by protecting its internal memory cells from being affected by the "squashing" effect of the logistic activation functions across recurrent links. Typically neural networks' weights and parameters are trained by some form of stochastic gradient descent, a gradient descent that shuffles inputs each round. RNNs and LSTMs have shown value in processing natural language texts and programming language texts. Hellendoorn and Devanbu [69] discuss the effectiveness of deep learning—specifically RNNs and LSTMs—on the task of suggesting the next token in a file. The authors compare deep learning against a n-gram/cache language model that dynamically changes based on what file and packages are in scope in an editing session within a IDE. They find that combining deep learning and the aforementioned cache language model achieves very low entropy. The disadvantage is that deep learning often requires keeping a closed vocabulary, making deep learning unsuitable for predicting novel identifiers and literals in ever-changing scopes. The LSTM model we present in this chapter maintains a closed vocabulary; for the purpose of detecting syntax errors, the exact value of identifiers and literals is irrelevant. Others have applied RNNs to source code. White et al. [162] trained RNNs on source code and showed their practicality in code completion. Similarly, Raychev et al. [129] used RNNs in code completion to synthesize method call chains in Java code. Dam et al. [39] provides an overview of LSTMs instead of ordinary RNNs as language models for code. Our work is similar to code completion, in that given a file with one token missing, Sensibility may suggest how to complete it; however, our focus is on syntax errors, rather than helping complete code as it is being written.

Deep learning and syntax error fixing has already been attempted by a few researchers with different degrees of success and treatment. Bhatia and Singh [22] present SynFix, which uses RNN and LSTM networks (collectively, RNNs) to automatically correct syntax errors in student assignments. The RNNs are trained on syntactically-correct student submissions, one model per programming assignment. RNNs takes a 9-token window from the input file and is trained to return a 9-token window shifted one token to the right. In other words, it outputs the next overlapping sliding window. Given an invalid file, SynFix naïvely generates a sequence of fixes at the error location as returned by the parser. The LSTM approach described in this chapter is quite similar to SynFix, however it varies in a few important ways: we use two LSTM models, whereas SynFix uses only one; SynFix is trained on a corpus comprised entirely of syntactically-valid submissions of one particular assignment, whereas our model is trained on a large corpus of syntactically-valid source code collected from GitHub. SynFix uses a thresholding method to rename identifiers, whereas we abstract identifiers (n-grams and LSTMs) and maintain all unique identifiers in the corpus (n-grams).

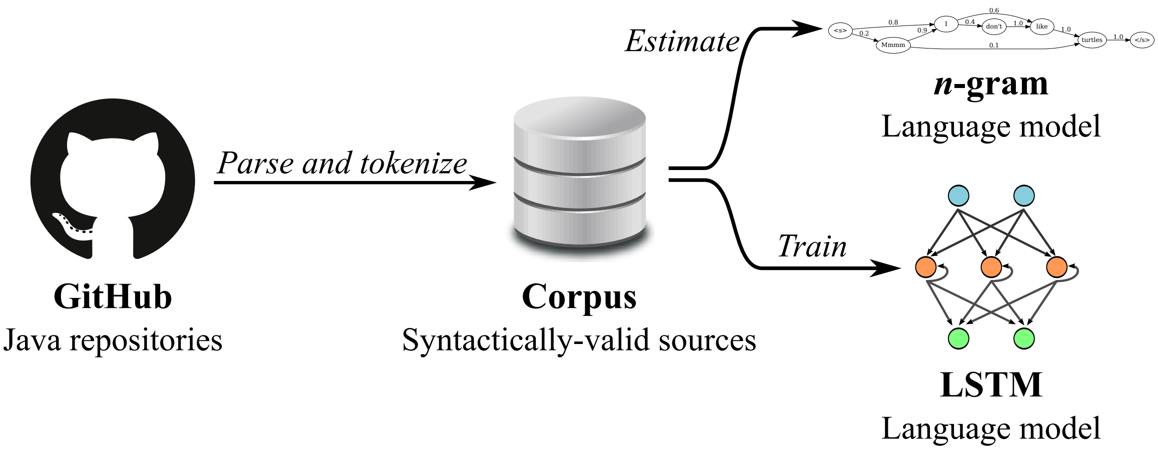


Figure 5.1: Methodology for training language models of code.

Finally, our method of generating fixes does not rely on the parser's conception of where the syntax error is, as the parser can be quite unreliable in this regard [28]. SynFix was able to completely fix 31.69% of student submissions, and partially fix 6.39% more files.

Gupta et al. [65] describe DeepFix, which models C source code with "common programming errors" (including syntax errors) as a noisy channel. The authors employed with a sequence-to-sequence gated recurrent units (GRUs)—a type of recurrent neural network—to decode erroneous C code to compilable C code. They trained their multilayer GRU models on 100–400 token long student-submitted C source code for which they have mutated to introduce synthesised errors. DeepFix is able to completely fix 27% of all erroneous student submissions, and partially fixed an additional 19% of student submissions.

5.3 Methodology

In order to suggest a fix for a syntax error, first we must find the error. For both finding errors and fixing syntax errors, it is useful to have a function that determines the likelihood of the adjacent token in the token stream given some *context* from the incorrect source code file (Equation 5.1).

$$P\left(\textit{adjacent-token} \middle| \textit{context}\right) \tag{5.1}$$

We estimated smoothed n-gram models to approximate the function in Equation 5.1 (Section 5.4). We also trained long short-term memory (LSTM) recurrent neural networks in a similar manner

(Section 5.5). In order to train the models, we needed a vast corpus of positive examples. For this, we mined over nine thousand of the most popular open source Java repositories from GitHub (Section 5.3.1). This source code was tokenized (Section 5.3.2) such that it could be used to train the Java language models. Finally, we used the approximated functions expressed in Equation 5.1 to detect a syntax error in a file and suggest a plausible fix. Figure 5.1 graphically describes this process.

5.3.1 Mining GitHub for syntactically-valid training examples

To obtain the training data,[1] we downloaded Java source code from GitHub. Since we required Java tokens from each file, other GitHub mining resources such as Boa [44] and GHTorrent [60] were insufficient. Thus, at the end of June 2017, we downloaded the top 10,000 Java repositories by stars (as an analog of popularity). Since GitHub's search application programming interface (API) outputs a total of 1,000 search results per query, we had to perform 10 separate queries, each time using the previous least popular repository as the upper bound of stars per repository for the next query. In total, we successfully downloaded 9,993 Java repositories.

For each repository, we downloaded an archive containing the latest snapshot of the Git repository's default branch (usually named `master`). We extracted every file whose filename ended with `.java`, storing a SHA-256 hash to avoid storing byte-for-byte duplicate files. We used `javac`'s scanner (tokenizer) and parser as implemented in OpenJDK 8 [122] to tokenize and parse every Java file downloaded. Parsing each source file allowed us to filter only Java source files that were syntactically valid according to the Java Platform Standard Edition 8 [57], more commonly known as Java 8. In total, we tokenized 2,322,481 syntactically-valid Java files out of 2,346,323 total `.java` files downloaded. All relevant data—repository metadata, source code, and repository licenses—were stored in an SQLite3 database.[2]

5.3.2 Tokenization

A *token* is the smallest meaningful unit of source code, and usually consists of one or more characters. For example, a semicolon is a token that indicates the end of a statement.

[1] Note: we used an unrelated corpus of syntactically-*invalid* Java code to evaluate our methods, described in Section 5.6.

[2] Available: https://archive.org/details/sensibility-saner2018

The set of all possible **unique** tokens tracked by a language model is called the *vocabulary*. Each new source file will likely contain novel variable names or string literals that have never been seen before. This complicates the creation of language models [69], since every novel file presented to the model will likely contain out-of-vocabulary tokens. For the purposes of learning the syntax of the language, we deem these problematic tokens to be irrelevant to the task. Thus, to keep a generally small, bounded vocabulary that has enough unique tokens to faithfully represent the syntax and regularity of handwritten Java, we *abstracted* certain tokens.

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`, `0xC0FFEE`, `0755` `0b101010`, `.3e-02d`, `0xFFp+12f`, ’’
String literal	Abstracted	`"hello, world"`

Table 5.1: Token kinds according to the Java SE 8 Specification [58], and whether we abstracted them or used them verbatim.

A key insight is that, in order to model the *syntax* of a programming language, it is unnecessary to model precise variable names and literal values. Thus, when creating a fixed vocabulary, we abstracted certain tokens that vary between files, and kept all other tokens verbatim (Table 5.1). The result was 110 unique tokens for the Java 8 standard. In addition to these tokens, we added the synthetic `<unk>` token to encode out-of-vocabulary tokens, and `<s>` and `</s>` tokens such that we could encode beyond the start and end of a source file, respectively [104]. Thus, the total size of our vocabulary was 113 unique tokens for the abstract models.

Although prior work [69] has found that limiting the size of the vocabulary is unhelpful for the task of code completion, we hypothesize that having a closed vocabulary would improve performance for the task of syntax error detection.

Original source code	`greeting = "hello";`
Tokenization	Identifier(`"greeting"`), Operator(`"="`), String(`"hello"`), Separator(`";"`)
Vocabulary abstraction	Identifier `=` String `;`
Vectorization	$[3\ 0\ 2\ 1]$

Table 5.2: The series of token transformations from source code to vectors suitable for training the n-gram and LSTM models. The simplified vocabulary indices are "`=`" = 0, "`;`" = 1, "String" = 2, and "Identifier" = 3.

5.3.3 Tokenization Pipeline

To convert a source file to a form that is suitable for training the models, we performed a series of transformations (Table 5.2). The raw source code was tokenized in a form that is suitable as input for the Java parser. As mentioned in Section 5.3.1, this was done for each Java file using `javac`.

Then, we normalized the token stream such that each token is an entry of the abstracted vocabulary. The exact text of tokens belonging to open classes was discarded for training, except in the case of the 10-gram Concrete model (Section 5.4). Each token in the vocabulary is assigned a non-negative integer index for the LSTM model or a text name for the 10-gram Abstract model.

5.4 Training n-gram models for syntax error correction

Inspired by the prior work by Campbell et al. [28], we implemented two separate 10-gram models for syntax error correction. The models work by first estimating a Modified Kneser-Ney smoothed 10-gram model on the valid code in the training corpus. Since the 10-gram model expects a corpus of space-separated words, each token in the training source code file was converted into a single "word". Unlike LSTMs, the n-gram model can easily handle arbitrary extensions to its vocabulary, so we created both an *abstract* model—like the LSTM models—and a *concrete* model, where token text was ingested verbatim for all tokens.

5.4.1 Detecting syntax errors with n-gram models

When presented with a syntactically invalid file which needs correction, the tool breaks the source code into 21-token-long windows, which are compared against the model for their cross-entropy. Equivalently, this is the negative logarithm of the probability of encountering that specific 21-token window according to the Kneser-Ney smoothed 10-gram model. The model in this case is either the abstract model, containing token types, or the concrete model, containing the actual text of each token.

Then, the cross-entropy is converted to a specific value for each token. Each token is assigned a score equal to the average entropy of every window which it was a member of. Since the window length is 21, each token is a member of 21 windows, and so its score is the mean entropy of those 21 windows.

The window length, 21 was chosen to match the LSTM window length in the next section and so that there is at least one 10-gram before each token and one 10-gram after each token being examined. This takes advantage of the fact that a single token can affect the entropy of a window ending with that token, the entropy of a window beginning with that token, and windows at every position in between.

The tool considers the token with the highest score to be the most likely location of a code mistake, since it contributed the most entropy to the 21 windows it was in.

5.4.2 Fixing syntax errors with n-gram models

The top-10 scoring tokens are then considered in turn for correction. This is limited to 10 to limit runtime. For each of the top-10 scoring tokens, the model attempts to delete, insert a token before, or substitute that token.

For deletion, there is only one option for each of the top-10 scoring tokens: try deleting that token. For insertion and substitution, the model has many options. It can insert any token it has seen before at that location, or it can substitute the token at that location with any token it has seen before. The tool tries any token it has seen in the training corpus with frequency ≥ 1000. This lower limit on token frequency is also imposed to limit the runtime of the tool.

The above process produces many possible fixes. For each possible fix, the tool uses the model to compute the entropy of the 21-token window with the deleted, inserted, or substituted token at the center. This entropy is subtracted from the original entropy, before the suggestion was applied. The resulting value is how much the cross-entropy of the erroneous file was decreased (or increased) by applying a possible fix.

If the entropy has decreased, this means the probability of this code being observed before has increased, which means, according to the model, the file (with the possible fix applied) is more likely to be syntactically correct. The fixes are ranked based on how much the entropy decreased after the fix suggestion was applied. The suggestion which causes the greatest decrease in entropy when applied is reported first.

The search process described above takes less than two seconds on a modern central processing unit (CPU) (Intel® Core™ i7-3700K) to produce a list of fixes and check them for syntactic-validity.

5.5 Training LSTMs for syntax error correction

To train the LSTMs, each source file was converted into a vector, representing the tokens of an entire file. The vector is constructed by substituting each token with its corresponding numeric index in the abstracted vocabulary (Section 5.3.2). Finally, each vector was converted into a *one-hot encoded* matrix (also known as *one-of-k encoding*). In a one-hot encoding, exactly one item in each column is given the value one; the rest of the values in the column are zero. The one-hot bit in this encoding represents the index in the vocabulary. Thus, the matrix has as many columns as tokens in the file and has as many rows as entries in the abstracted vocabulary. Each column corresponds to a token at that position in the file, and has a single one bit assigned to the row corresponding to its (zero-indexed) entry in the vocabulary.

The modelling goal is to approximate a function that, given a context from source code, determines the likelihood of the adjacent token. If this function judges the token as unlikely, it indicates a syntax error. This function solves the first problem: finding the location of a syntax error, as demonstrated by Campbell et al. [28]. However, to fix errors, it is also necessary to know what tokens actually *are* likely for each given context. We rephrase Equation 5.1 such that, instead of predicting the likelihood solely of the adjacent token, it returns the likelihood of *every entry in the vocabulary.* In other words, we want a function that returns a *categorical distribution* (Equation 5.2).

$$
adjacent[context] = \begin{cases} P(\texttt{if}|context) \\ P(\texttt{else}|context) \\ P(\text{Identifer}|context) \\ P(\text{String}|context) \\ \ldots \\ P(\texttt{\}}|context) \end{cases} \tag{5.2}
$$

```
if (activity == null) {
    this.active = false;
}
```

Listing 5.2: Syntactically-valid Java

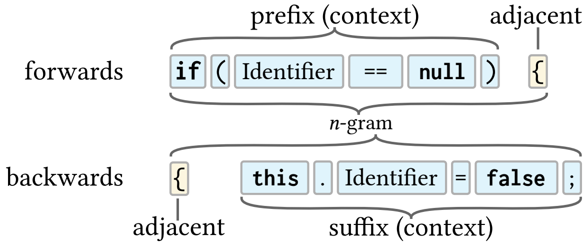


Figure 5.2: The relationship between an n-gram, the contexts, and the adjacent token. In this diagram, $n = 7$, and the adjacent token is the `{` in both cases. Thus, the contexts (both prefix and suffix) are $n - 1$ or 6 tokens long.

The categorical distribution can also be seen as a vector where each index corresponds to the likelihood of an entry in the vocabulary being the adjacent token. Being a probability distribution, the sum of the elements in this vector add up to 1.0. The probability distribution works double duty—because it outputs probabilities, it can determine what the most probable adjacent token *should be*; hence, it can be used to determine possible fixes (discussed in Section 5.5.2).

To approximate such a function, we used deep learning to map contexts to categorical distributions of the adjacent token. For this task, we employed LSTM recurrent neural networks, as they were successfully used by prior work in predicting tokens from source code [129, 162]. Unlike the prior work, we have trained **two** models—the *forwards* model, given a *prefix* context and returning the distribution of the next token; and the *backwards* model, given a *suffix* context and returning the distribution of the previous token. Using recurrent neural networks in two different directions was used successfully in speech recognition [66, 10], but has yet to be applied to source code.

Our insight is that models with opposite viewpoints (that is, different contexts for the same adjacent token) may return *different* categorical distributions. That is, whilst the forwards model may declare that the keyword `this` is likely for the next token, the backwards model may declare that an open brace (`{`) is far more likely than the keyword `this`. With this formulation, we are able to both detect the location of syntax errors and produce possible fixes.

As an example, consider the syntactically-valid Java snippet in Listing 5.2. Figure 5.2 illustrates the contexts—both prefix and suffix—when estimating the likelihood of the open brace (`{`) on the first line of Listing 5.2.

Input	One-hot matrix, dimensions = $\tau \cdot \|V\|$		
	Type	Parameters	Activation
	LSTM	300 hidden units	tanh; recur.: hard sigmoid
	Dropout	rate = 0.75	
	Dense		softmax
Output	Categorical distribution, size = $\|V\|$		
Loss	Categorical cross-entropy		
Optimizer	Adam, initial learning rate = 0.001		

Table 5.3: Summary of the neural network hyperparameters we used. $|V| = 113$ is the size of the vocabulary (Section 5.3.2) and $\tau = 20$ is the length of each context in number of tokens.

As training input, we iterated over each token in each file, moving a *sliding window* over the tokens of the source code file. Based on our experiments (Appendix A), we chose a context length τ of 20 tokens. This corresponds to an n-gram length of 21 tokens, as an n-gram includes both the context and the adjacent token. The prefix was provided as the example input to the forwards model, and the suffix was provided to the backwards model. Both contexts were provided as a one-hot matrix (alternatively, a sequence of one-hot vectors). As example output to both models, we provided the adjacent token, as a one-hot vector. To handle tokens whose contexts extend beyond the start and end of the file, we inserted synthetic `<s>` and `</s>` tokens, collectively called *padding* tokens [104]. This means that the first prefix context in the file is comprised entirely of 20 `<s>` tokens; likewise, the last suffix context in the file is comprised of 20 `</s>` tokens.

We used Keras 2.0.8 [33], a Python deep neural network framework, to define our model architecture, using the Theano 0.8.2 [153] backend to train the models proper. A summary of the precise architecture and hyperparameters that we used is given in Table 5.3. We arrived at these hyperparameters by performing a grid search over the course of one-and-a-half months; Appendix A describes this process in detail. Our best performing models include an LSTM layer with 300 hidden units, and a τ of 20 tokens. We used the Adam [89] gradient descent optimizer with an initial learning rate of 0.001, optimizing to minimize categorical cross-entropy. We ran a variable number of *epochs*—full iterations of the training examples—to train the models, using *early stopping* to determine when to stop training. Early stopping was configured to stop upon detecting 10 consecutive epochs (its *patience* parameter) that yield no improvement to the categorical cross-entropy with respect to the validation examples.

In order to train an model with a form of stochastic gradient descent, the training examples must be presented in random order. Thus, for each epoch, we shuffled the training files. We concatenated the windows from the list of shuffled files. We trained by presenting the optimizer with mini-batches containing 32 examples each. We constructed the mini-batches by further shuffling the windows *within* files. In our early experiments, we found that identifier abstraction caused many mini-batches to contain over half of its windows' targets be IDENTIFIER; subsequently, many models "learned" the syntax of the Java language by always outputting that the adjacent token is IDENTIFIER. Shuffling windows within files makes the creation of such imbalanced mini-batches less likely; thus the gradient descent optimizer is less able to "cheat" by simply outputting the majority class (i.e., IDENTIFIER) to get a lower loss value.

Once each model was trained, it was serialized in Hierarchical Data Format, version 5 (HDF5). In total, the weights and biases of each individual model resulted in 6.1 MiBs of data.[3] Each model individually took between 2½ and 11½ days to train; up to six models were trained simultaneously on two Nvidia® GeForce® GTX 1070 graphical processing units (GPUs). Section 5.7.1 discusses how files were chosen for the training, validation, and testing sets.

5.5.1 Detecting syntax errors with dual LSTM models

We used the output of the models trained in Sections 5.4 and 5.5 to find the likely location of the syntax error. Given a file with one syntax error, we tokenized it using `javac`. `javac`'s tokenizer is able to tolerate erroneous input in the tokenization stage, which produces a token stream (as opposed to the parsing stage, which produces an abstract syntax tree).

Recall that each model outputs the probability of the adjacent token given a *context* from the token stream, but each model differs in where the context is located relative to the adjacent token. We used the two independent probability distributions to determine which tokens are quantifiably unlikely ("unnatural" [28]).

To calculate "naturalness", we summed the cross-entropy of each probability model with respect to the erroneous source file. Once the naturalness of every single token in the file is calculated, we return a sorted list of the least likely tokens, or, in other words, the locations that are most likely to be a syntax error.

[3] Available: https://archive.org/details/sensibility-saner2018

For a discrete distribution $p(x)$ and a distribution $\hat{q}(x)$ that models $p(x)$, the cross-entropy is computed as follows

$$H(p, \hat{q}) = -\sum_{x \in X} p(x) \log_2 \hat{q}(x)$$

This returns a value in $[0, \infty)$, where 0 indicates that $\hat{q}$ models p perfectly; as the cross-entropy increases, this increases the amount of information required to model $p(x)$. Hence, a value closer to 0 is more "natural", whereas larger values indicates a more "unnatural" event.

For each token position in the erroneous source file, we calculate the cross-entropy with respect to the current token in the file, for both the forwards and backwards models. That is, we let $p(x)$ equal 1 *iff* x is equal to the token at the current position in the file.

$$p(x) = \begin{cases} 1, x = \text{actual token} \\ 0, \text{otherwise} \end{cases}$$

This is equivalent to a one-hot encoding of the token at the current position.

We then use $\hat{q}_f(x|\text{prefix})$ and $\hat{q}_b(x|\text{suffix})$ obtained from consulting the forwards and backwards models with their corresponding context, respectively. We then combine the two cross-entropies by summing. Thus, the "unnaturalness" of a token t from the source file is:

$$\textit{unnaturalness} = H(p, \hat{q}_f) + H(p, \hat{q}_b)$$

To obtain a ranked list of possible syntax error locations, we sort the list of tokens in the file in descending order of unnaturalness. That is, the positions with the highest summed cross-entropy are the most likely location of the syntax error.

5.5.2 Fixing syntax errors with dual LSTM models

Given the top-k most likely syntax error locations (as calculated in Section 5.5.1), we use a naïve "guess-and-check" heuristic to produce and test a small set of possible fixes. Using the categorical distributions produced by the models, we obtain the top-j most likely adjacent tokens (according to each model) which may be inserted or substituted at the estimated location of the fault. Each fix is tested to see if, once applied, it produces a syntactically-valid file. Finally, we output the valid fixes.

For a given syntax error location, we consult each model for the top-j most likely tokens at that position. The models often produce similar suggestions, hence we take the union of the two sets.

$$\textit{suggestions} = \operatorname{top}_j\left(\hat{q_f}\right) \cup \operatorname{top}_j\left(\hat{q_b}\right)$$

We then try the following edits, each time applying them to the syntactically-invalid file, and parsing the result with `javac` to check if the edit produces a syntactically-valid file. If it does, we consider the edit to be a *valid fix*. We use the following strategies to produce fixes:

1. Assume a token at this location was erroneously **deleted**. For each t in the set of suggestions, insert t at this location.
2. Assume the token at this location was erroneously **inserted**. Delete this token.
3. Assume the token at this location was erroneously **substituted** for another token. Substitute it with t, for each t in the set of suggestions.

We repeat this process for the top-k most likely syntax error locations. We let $k = 3$ to limit runtime. j was calibrated according to the *perplexity* of the estimated probability distribution. Perplexity is, roughly speaking, the expected number of choices that an estimated distribution $\hat{q}$ requires to model events from the true distribution p. It is directly related to cross-entropy $H(p, \hat{q})$ by $2^{H(p,\hat{q})}$. To determine j, we used the ceiling of the highest validation cross-entropy obtained while training. The highest cross-entropy was 1.5608, or a perplexity of just under 3; therefore, we let $j = 3$.

Finally, all valid fixes are output to the user. The LSTM model takes less than three seconds on a modern CPU (Intel® Core™ i5-3230M) to produce a list of suggested fixes for a single file. Every single fix suggested this way is guaranteed to produce a syntactically-valid file.

5.6 Mining Blackbox for novice mistakes

Both the n-gram and LSTM model presented in this chapter are trained on data from professional code available on GitHub. In order to properly evaluate these models for their intended purpose—detecting and correcting syntax errors in novices' code—we would ideally have access to a repository of syntax errors made by novices.

Blackbox [27] is a continually-updated repository of events from the BlueJ Java IDE [92]—an IDE aimed primarily at novices learning Java in introductory computer science classes. Blackbox has been used to analyze novice programmers before [7, 26, 142, 8]. The data contains over four years of IDE events collected from users around the world who have opted-in to anonymously share edit events. The IDE events includes the start and end of an editing session, the invocation of the compiler, whether a compilation succeeded or failed, and edits to lines of code. Importantly, one can reconstruct the source code text at the time of any compilation event.

We mined Blackbox in order to find realistic data to evaluate our syntax-correction algorithms. By using the IDE event data in Blackbox, we collected syntax mistakes "in the wild", which are accompanied by the *true* resolution to that mistake. The intended audience of Sensibility is novices, thus it is important evaluate our syntax error correction methods against actual code that a novice would write.

5.6.1 Retrieving invalid and fixed source code

For the purposes of the evaluation, we sought to retrieve pairs of revisions to a source code file: the revision directly prior to a compilation that failed due to a syntax error, and the revision immediately following which compiled successfully.

In order to collect such pairs of revisions, we iterated through every editing session from the beginning of data collection up to midnight, July 1, 2017, UTC. For each session, we iterated through each pair of compilation events and filtered the pairs wherein the former compilation had failed (the "before" revision) and the compilation immediately following had succeeded (the "after" revision). We retrieved the complete Java source code at each revision of the pair and kept only those pairs wherein the source code of the "before" revision was syntactically-incorrect (verified using `javac`).

For the purposes of our analyses, we calculated the *token-wise edit distance* between the before-and-after revisions of source code. We used Levenshtein distance [97], wherein two strings of tokens are compared for the minimum amount of single token *edits* (insertions, deletions, or substitutions) that are required to transform one string of tokens into the other. Thus, edit distance indicates how many edits, at minimum, a syntax error correcter must suggest to transform an invalid source file to a syntactically-valid source file.

Table 5.4: Edit distance of collected syntax errors

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 5.5: Summary of single token syntax-errors

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

5.6.2 Findings

In total, we collected 1,715,312 before-and-after pairs matching our criteria. Of these pairs, 984,471 (57.39 %) were single-token syntax errors (Table 5.4)[4]; This means that, even if a tool accounts *only* for single-token mistakes, the tool accounts for the majority of syntax errors.

The majority of syntax errors we found differed from the fixed source file by a **single** edit.

We further studied the single-token syntax errors. Table 5.5 breaks down the errors into each edit operation. The majority of single-token syntax errors are deletions—such as when a programmer misses a token such as a semi-colon, or a brace. Insertions account for almost one-quarter of errors, while substitutions are the least common.

In Section 5.7, we describe how we used the single-token syntax errors we mined to evaluate LSTM and n-gram methods for syntax error correction and detection.

5.7 Evaluation

To determine the practical usefulness of Sensibility, we ask the following questions:

1. How well does Sensibility find syntax errors?

[4] Syntax errors caused by invalid escape sequences within string literals were considered to have an edit distance of zero, as only the contents of the string must change, but not the actual token type.

Table 5.6: Number of tokens among partitions in the train and validation sets

	Mean	S.D.	Median	Min	Max
Train	8.16 M	596,019.30	8.06 M	7.45 M	9.00 M
Validation	3.80 M	365,337.76	3.68 M	3.47 M	4.34 M

2. How often does Sensibility produce a syntactically-valid fix?

3. If a fix is produced, how often is it the same fix used by the student to fix their own code?

To answer these questions, we created three "qualifications" for a suggested fix. The first qualification is that the fix suggestion must be at the exact *location* of the mistake in the code, down to the individual token.

To judge how well Sensibility produces syntactically-valid fixes, we created a second qualification, *valid fix*, which is stricter: the fix suggestion must be at the exact location of the mistake in the code, and applying the fix suggestion must yield a syntactically valid source file.

The third qualification is the most strict: the fix suggestion must be exactly the same token (abstracted or concrete, depending on the model) that the student used to fix their own code. A fix suggestion that matches the student's own fix is called a *true fix*. A true fix precisely reverses the mistake that introduced the error.

Then we found the highest ranking fix suggestion produced by the tool using various models and training partitions that met the above qualifications.

5.7.1 Partitioning the data

To empirically evaluate Sensibility, we repeated our experiments five times on mutually-exclusive subsets of source code files called *partitions*. Each of the five partitions are subdivided further into two mutually-exclusive sets: the **train set**, and the **validation set**, thus resulting in 10 sets total (described in Table 5.6). The validation set was used exclusively for the LSTM training procedure described in Section 5.5, but not used when estimating n-gram models.

We populated every set with Java source code from the corpus collected in Section 5.3.1. The training and validation sets were used in the training phase (Section 5.4 and 5.5).

We split our evaluation into five partitions to demonstrate how the performance of Sensibility changes when trained on completely different example source code. Keeping each corresponding partition the same size facilitates the comparison of results between partitions. This also represents the expected use case of an end-user who will never retrain Sensibility.

When assigning source code files to partitions, we imposed the following constraint to ensure the independence of training partitions: The source code files of a single repository cannot be distributed over partitions; in other words, every source code file in a given repository must be assigned to one and only one partition. The constraint's purpose is to make the evaluation more realistic, considering the expected use case of Sensibility. If a user is trying to figure out a syntax error in their own hand-written source code, it is likely that their model was trained on whole projects at once.

The test files did not come from the GitHub corpus. Instead, they came from the Blackbox repository of novice programmers' activity [27]. The same set of test files were used to evaluate each partition; Only the training and validation files changed between partitions.

5.7.2 Finding the syntax error

To quantify Sensibility's accuracy in finding the error, we calculated the mean reciprocal rank (MRR) of the ranked syntax error location suggestions. Reciprocal rank is the inverse of the rank of the first correct location found in an ordered list of syntax error locations for a file q. Mean reciprocal rank is the average of the reciprocal rank for every file q in the set of total solutions attempted Q:

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}$$

MRR is always a value in $[0, 1]$, where an MRR of 1 is the best possible score, obtained when the first suggestion is always the correct location of the error. Conversely, an MRR of 0 means that the correct location number is never found in the ranked list of suggestions. For example, if for one mistake, the correct token was given first, for another student mistake, the correct token was given third, and for yet another mistake the correct token was never found, the token-based MRR would be $\frac{1}{3}\left(\frac{1}{1} + \frac{1}{3} + 0\right) = 0.44$. MRR is quite conservative: in the case that the correct result is ranked first half of the time and ranked second the rest of the time, the MRR is only 0.75. To quantify Sensibility's ability to find syntax errors, we determined how often it finds the *exact location* of the erroneous token.

Table 5.7: MRRs of n-gram and LSTM model performance

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	Location	.52	.53	.53	.50	.50	.52
	Valid Fix	.52	.53	.52	.49	.50	.51
	True Fix	.46	.46	.46	.44	.44	.46

5.7.3 Fixing the syntax error

To evaluate the effectiveness of Sensibility to fix syntax errors, we measured how often the models produces an edit that, when applied, produces a syntactically-valid file. We report this as a *valid fix*. A stricter measure of success is how often Sensibility produces the *true fix*. The true fix is defined as the exact same fix that the student applied to their own code. For *abstract* tokens, the true fix requires that the tool applied the correct operation (insertion, deletion, substitution) at the correct location, and with the correct token type. For *concrete* tokens, the token must also match exactly, not just its type, to count as a true fix. Thus, for the 10-gram Concrete to produce a true fix it must produce the exact identifier, number literal, string, etc. that the student used to fix their code. So, if the student's fix was to insert the identifier `a`, the 10-gram Concrete model must also insert the identifier `a`, while the abstract models must only suggest inserting `identifier` at that same location.

5.8 Results

Performance of finding syntax errors is measured by mean reciprocal rank. Table 5.7 lists the MRR obtained when locating the exact token of the syntax error, generating a valid fix, and generating the true fix for each tool. As can be seen from the table, results are very consistent across partitions.

The values in Table 5.7 are all well above what would be expected from guessing by chance alone. If a file had 100 lines and 10 tokens per line, location MRR would be 0.02 just by guessing. Since fix

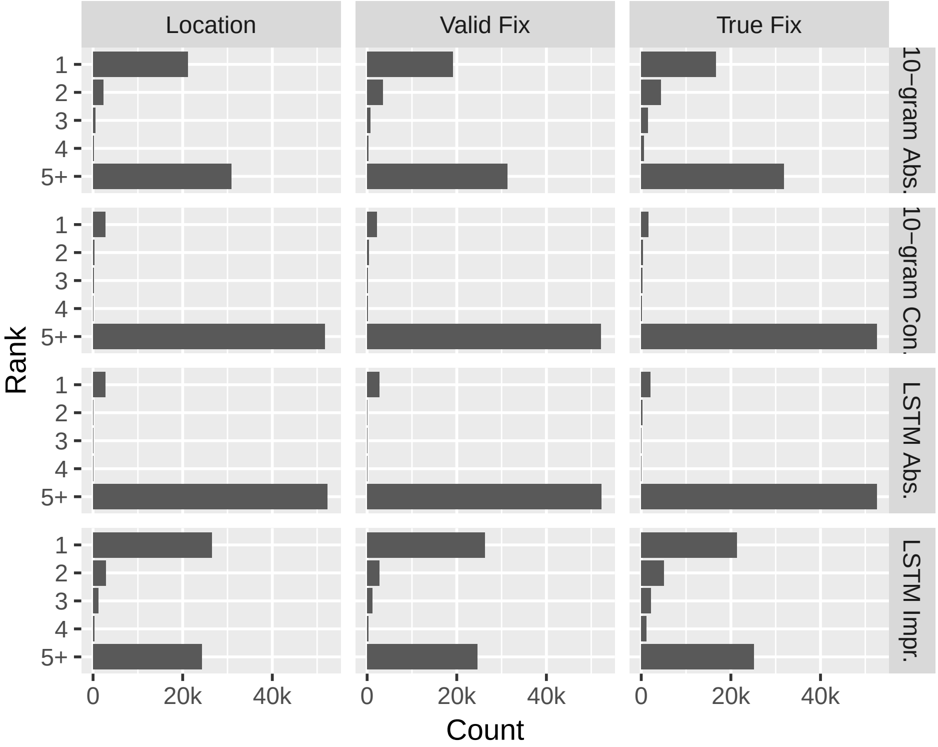


Figure 5.3: The mean reciprocal ranks of determining the exact location, a valid fix and the true fix of student mistakes for all three models.

MRRs depend on determining the location first, just by guessing, they would be even lower than 0.002.

Figure 5.3 is a series of histograms visualizing the ranks of each qualification. The width of each bar displays the number of observations that have a rank at that value. Figure 5.3 shows that the MRR values in Table 5.7 are dominated by either rank 1 results, that is, the tools first result was qualifying, or rank 5+ results, that is the qualifying result was far down the suggestion list, or the tools were unable to produce a qualifying result. In this case, qualifying means the result is the correct location, the result was the first valid fix, or the result that was the true fix.

5.8.1 Performance of fixing syntax errors

In all cases, there is a clustering of reciprocal ranks at 1.0, meaning that the 10-gram Abstract model tool can suggest the true fix as its first suggestion 30.22% of the time. The 10-gram Concrete model tool can suggest the true fix as its first suggestion 3.52% of the time. The LSTM model tool can suggest the true fix as its first suggestion 38.9% of the time.

A dual LSTM model can produce a fix for syntactically-invalid code by suggesting the true token type, operation, and location over one-third of the time.

5.9 Discussion

While the improved LSTM model performed the best overall—producing a valid fix in its top-2 suggestions—it has a few disadvantages compared to n-grams. Namely, extending the models to account for changes in the language specification is more costly, since n-gram models can be simply appended to, whereas LSTM models must be retrained. Additionally, n-gram models are much simpler to compute—being mostly look-ups in highly-optimized associative arrays; whereas LSTMs require a large series of floating-point arithmetic, often requiring GPU acceleration.

Using Sensibility to correct multiple mistakes at once is theoretically possible but due to the fact that, at best, its first fix suggestion is only the true fix approximately half of the time, more advanced search strategies may be required that try fixes in different combinations. Alternatively more generic forms of search can be exploited, such as SMT solvers or heuristic search to help produce more parsable and safe outputs. In this work we only explore code as a one-dimensional token stream, but hand-written code is often laid out in two dimensions on the user's screen. Spatial relationships or positioning could prove useful in future error locators and fixers.

Our work is complementary to other syntax error detection tools such as Merr [82] and Clang's "fix it" hints [35]. This work can be integrated with other tools to squash syntax errors. Sensibility is only one step in the hunt for typos. It attempts to handle any typos that would produce a definite syntax error. However, it does not address misspelled variable names, or type errors. Sensibility can help other tools that require a valid abstract syntax tree—such as type checkers—that cannot work on code with invalid syntax. Localness [158], online n-gram models, or other search-based models might be feasible candidates to help resolve identifiers. Further investigation is required into applying ensembles of learners to combine the strengths of deep learning, smoothed n-gram models, and other probabilistic models to achieve high precision in detecting and fixing syntax errors.

5.9.1 Threats to validity

Construct validity is threatened by the abstraction of identifiers. By suggesting that an identifier be inserted or modified, the choice is still up to the programmer since the code may parse, but may not necessarily be compilable.

Internal validity is threatened by our reliance on the Blackbox data set [27, 7]. There is potential bias from the curators of the dataset, and a self selection bias as the syntax errors are submitted with the consent of the students. Internal validity could be harmed by using directly adjacent compilations: the first for erroneous code and the second for the fixed source code. This could miss situations where a syntax error is made, doesn't compile, and several fixes are attempted by a human. In this case the erroneous source code we used wouldn't match the original erroneous source code a human had written, that is, their first attempt.

External validity is addressed by the use of a large number of Java source files; however these source files were only collected from GitHub, thus are not necessarily representative of all Java code or code of other languages.

5.10 Conclusions

We have described a method of exploiting n-gram and LSTM language models to locate syntax errors and to suggest fixes for them. Typically a fix is found about half of the time. The fix is often suggested based on abstract tokens, so often the end-user needs to fill in the appropriate identifier.

Abstract vocabularies overcomes two limitations of software engineering deep learning problems: vocabulary size and unseen vocabulary, and cost of training for end-users. Both n-gram and LSTM models require extensive training; we present a method for training on large corpora before deployment such that end-users of the model never have to locally train.

This work demonstrates that search aided by naturalness—language models applied to source code—can produce results with appropriate accuracy and runtime performance (two to three seconds on a modern CPU).

In summary by training language models on error-free code, we enable them to locate errors through measures of perplexity, cross-entropy, or confidence. We can then exploit the same language models bi-directionally to suggest the appropriate fixes for the discovered syntax errors. This relatively simple method can be used to aide novice learners avoid syntax-driven pitfalls while learning how to program.

Acknowledgments

We thank Neil C. C. Brown, Ian Utting, and the entire Blackbox administration team for the creation of the Blackbox repository, their continual support, and community that they foster. We thank Julian Dolby, for offering a new perspective on the problem. We would also like to thank Nvidia Corporation for their GPU grant. This work was funded by a MITACS Accelerate with BioWare and a NSERC Discovery Grant.

Chapter 6

Conclusion

In this thesis, I have demonstrated three separate applications of the *naturalness* of software.

I discussed how software engineering tasks can posed as **search problems**. I have provided evidence that:

> the *naturalness* of software is an effective tool for **reducing the search space** in software engineering tasks posed as search problems

such as "What is the most appropriate bucket for this new crash report?" (Chapter 4); "Where is the location of the true syntax error in this source code file?" (Chapter 5); "What action do I need to perform in order to fix this syntax error?" (Chapter 5).

The main unit of meaning in natural language models and information retrieval systems is the *token*; thus, in order to exploit the these tools for software engineering tasks, one must first **create a mapping from software artifacts to tokens** in a process called *tokenization*. In this thesis, I have shown there is no "one-size-fits-all" approach to tokenization; I have provided evidence that

> **choices made in tokenizing software artifacts** may have drastic repercussions on the effectiveness of search

such as when tokenizing jargon-heavy commit messages (Chapter 3), tokenizing crash reports the task of deduplication (Chapter 4), and when locating a syntax error using *n*-gram language models (Chapter 5).

Finally, I have provided evidence that:

investigating **unlikely events** can be a useful tool in finding mistakes

in certain circumstances such as when pinpointing the location of a syntax error (Chapter 5). In other circumstances, the results require care to interpret, such as when trying to predict when a commit will break the build (Chapter 3).

6.1 Future research directions

The introduction (Chapter 1) and Chapter 3 touched upon how disparate software repositories are *interrelated* (traceability). An interesting research direction is to learn *how* various software artifacts are interrelated, and furthermore automatically mine these relationships. One could exploit the *naturalness* of software to find implicit links between software artifacts. Since source code, commits, issues, crash reports, and discussions are related, often created by software developers, one could construct an ontology linking software artifact to each other across several projects source code repositories, which one could query. This ontology could use Semantic Web [21] principles and be queried using SPARQL [144] to answer questions such as "who has fixed an energy bug?", "how do developers fix `UnicodeDecodeError` crashes?", and other queries that require the interaction of several repositories of artifacts to answer.

Another direction to research is, rather than modeling source code as a one-dimensional series of tokens, explore the ways in that source code is multi-dimensional. For example, source code is often written in a two-dimensional text editor, where line breaks often carry meaning to the programmers that maintain the code base. The line breaks and indentation serve a clue as to the intention of the programmer. When the code is syntactically-correct, one could model the levels of declaration (scope) as yet another dimension.

6.2 Summary

Previous results by Hindle et al. [74] have provided evidence that natural language modeling may be an effective tool when applied to software source code. In this thesis, I have demonstrated some applications of the *naturalness* of software. I have shown how to map software artifacts to language model and information retrieval systems through the process of application-specific tokenization. I have demonstrated that *naturalness* can efficiently prune the search space when solving problems such as, "*what cluster does this crash belong to?*" (Chapter 4) and "*what is the correct token to insert here?*" (Chapter 5). I have shown that it is worth investigating events that are *unnatural*, such as when trying to predict if a commit will break the build (Chapter 3) or when pinpointing the source of a syntax error (Chapter 5). This thesis demonstrates early work exploiting the *naturalness* of software. I suggest that future research may augment *naturalness* with multi-dimensional modeling (such as using line numbers and indentation levels), and augmenting the field of software traceability by exploring the explicit and implicit links between software repositories.

Bibliography

[1] Iftekhar Ahmed, Nitin Mohan, and Carlos Jensen. The Impact of Automatic Crash Reports on Bug Triaging and Development in Mozilla. In *Proceedings of The International Symposium on Open Collaboration*, OpenSym '14, pages 1:1–1:8. ACM, 2014.

[2] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, 1972.

[3] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? A characterization of open source software repositories. In *16th IEEE International Conference on Program Comprehension*, ICPC 2013, pages 182–191, 2008.

[4] Anahita Alipour, Abram Hindle, and Eleni Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 183–192. IEEE Press, 2013.

[5] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.

[6] Miltiadis Allamanis and Marc Brockschmidt. Smartpaste: Learning to adapt source code. *CoRR*, abs/1705.07867, 2017.

[7] Amjad Altadmri and Neil CC Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527. ACM, 2015.

[8] Amjad Altadmri, Michael Kölling, and Neil CC Brown. The Cost of Syntax and How To Avoid It: Text versus Frame-Based Editing. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[9] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information retrieval*, 12(4):461–486, 2009.

[10] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in English and Mandarin. http://arxiv.org/abs/1512.02595, 2015.

[11] Craig Andrews. Bug #126558 “evolution-exchange-storage crashed with sigsegv in ...” : Bugs : evolution-exchange package : Ubuntu. https://bugs.launchpad.net/ubuntu/+source/evolution-exchange/+bug/126558, July 2007. (Accessed on 02/08/2018).

[12] Martin Anthony. *Discrete mathematics of neural networks: selected topics*. Society for Industrial and Applied Mathematics, 2001.

[13] Apple Inc. Technical Note TN2123: CrashReporter. https://developer.apple.com/library/mac/technotes/tn2004/tn2123.html, 2016.

[14] Appveyor Systems Inc. About us | AppVeyor. https://www.appveyor.com/about/, 2017. (Accessed on 05/11/2017).

[15] Atlassian Pty Ltd. Bitbucket | the Git solution for professional teams. https://bitbucket.org/product, 2017. (Accessed on 05/09/2017).

[16] Atlassian Pty Ltd. JIRA software - issue & project tracking for software teams. https://www.atlassian.com/software/jira, 2017. (Accessed on 05/10/2017).

[17] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval*. Addison-Wesley, 2 edition, 1999.

[18] Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill. How developers visualize compiler messages: A foundational approach to notification construction. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, pages 87–96. IEEE, 2014.

[19] Kevin Bartz, Jack W. Stokes, John C. Platt, Ryan Kivett, David Grant, Silviu Calinoiu, and Gretchen Loihle. Finding similar failures using callstack similarity. In *SysML*, 2008.

[20] Brett A. Becker. An exploration of the effects of enhanced compiler error messages for computer programming novices. Master’s thesis, Dublin Institute of Technology, 2015.

[21] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific American*, 284(5):28–37, 2001.

[22] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. Availlable: http://arxiv.org/abs/1603.06129, 2016.

[23] Alain Biem. Neural networks: A review. In Charu C. Aggarwal, editor, *Data Classification: Algorithms and Applications*, chapter 8, pages 205–235. CRC Press, Boca Raton, FL, 2014.

[24] Neil C Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. Analyzing the effects of test driven development in github. *Empirical Software Engineering*, pages 1–28, 2017.

[25] M. Brodie, Sheng Ma, G. Lohman, L. Mignet, M. Wilding, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Second International Conference on Autonomic Computing, 2005. ICAC 2005. Proceedings*, pages 101–110, 2005.

[26] Neil Christopher Charles Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the tenth annual conference on International computing education research*, pages 43–50. ACM, 2014.

[27] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 223–228. ACM, 2014.

[28] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 252–261. ACM Press, 2014. Available: http://dl.acm.org/citation.cfm?doid=2597073.2597102.

[29] Joshua Charles Campbell, Eddie Antonio Santos, and Abram Hindle. The unreasonable effectiveness of traditional information retrieval in crash report deduplication. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 269–280. ACM, 2016.

[30] Canonical Ltd. Launchpad. https://launchpad.net/, 2004.

[31] Marco Castelluccio, Carlo Sansone, Luisa Verdoliva, and Giovanni Poggi. Automatically analyzing groups of crashes for finding correlations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 717–726. ACM, 2017.

[32] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 1999.

[33] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[34] CircleCI. About us - CircleCI. https://circleci.com/about/, 2017. (Accessed on 05/11/2017).

[35] Clang—Expressive Diagnostics, October 2016. Available: http://clang.llvm.org/diagnostics.html.

[36] Kevin S. Clarke. added init.d test to travis config · ksclarke/solr-jetty-maven@80b627f. https://github.com/ksclarke/solr-jetty-maven/commit/80b627f6327afc178e9b0d1a14c9821a197bc76d, November 2013. (Accessed on 02/08/2018).

[37] Evan Czaplicki. Compiler errors for humans. http://elm-lang.org/blog/compiler-errors-for-humans, 2015.

[38] Mike Dalessio. Updating manifest.txt. · sparklemotion/nokogiri@f60f78d. https://github.com/sparklemotion/nokogiri/commit/f60f78dff19043c09dc3c6ff7bd975edfff0730f, March 2013. (Accessed on 02/08/2018).

[39] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. Preprint, 2016.

[40] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. ReBucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1084–1093. IEEE Press, 2012.

[41] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 75–80. ACM, 2012.

[42] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 333–342, 2011.

[43] Thomas Dy and Ma. Mercedes Rodrigo. A detector for non-literal java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 118–122, New York, NY, USA, 2010. ACM.

[44] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *35th International Conference on Software Engineering*, ICSE 2013, pages 422–431, 2013.

[45] Edgewall Software. The Trac project. https://trac.edgewall.org/, 2017. (Accessed on 05/10/2017).

[46] Elasticsearch BV. Elasticsearch. https://www.elastic.co/products/elasticsearch, 2016.

[47] Elasticsearch 1.6 documentation: Pattern analyzer. https://github.com/elastic/elasticsearch/blob/1.6/docs/reference/analysis/analyzers/pattern-analyzer.asciidoc, May 2015. (Accessed on 2018-03-28).

[48] Michael Fairley. Cargo-cult maven · 1000memories/photon-core@f6e0fe6. https://github.com/1000Memories/photon-core/commit/f6e0fe6f8a4bf03a044c83dd691517ac1edfc35c, August 2012. (Accessed on 02/08/2018).

[49] Michael D. Feist, Eddie Antonio Santos, Ian Watts, and Abram Hindle. Visualizing project evolution through abstract syntax tree analysis. In *Software Visualization (VISSOFT), 2016 IEEE Working Conference on*, pages 11–20. IEEE, 2016.

[50] David Freedman and Persi Diaconis. On the histogram as a density estimator: L_2 theory. *Probability theory and related fields*, 57(4):453–476, 1981.

[51] Sandy Garner, Patricia Haden, and Anthony Robins. 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*, pages 173–180. Australian Computer Society, Inc., 2005.

[52] GitHub Inc. About GitHub. https://github.com/about, 2017. (Accessed on 05/09/2017).

[53] GitHub Inc. About issues - user documentation. https://help.github.com/articles/about-issues/, 2017. (Accessed on 05/10/2017).

[54] GitHub Inc. Autolinked references and URLs—user documentation. https://help.github.com/articles/autolinked-references-and-urls/#issues-and-pull-requests, 2017. Accessed on 2017-04-24.

[55] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 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, pages 103–116. ACM, 2009.

[56] Google Inc. Google Code. https://code.google.com/, 2017. (Accessed on 05/09/2017).

[57] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Oracle Corporation, 8 edition, February 2015. Available: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[58] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. Lexical structure. In *The Java Language Specification*, chapter 3. Oracle Corporation, 2015. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html.

[59] Claire Le Goues, ThahnVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.

[60] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236. IEEE Press, 2013.

[61] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[62] Diego Güemes-Peña, Carlos López-Nozal, Raúl Marticorena-Sánchez, and Jesús Maudes-Raedo. Emerging topics in mining software repositories. *Progress in Artificial Intelligence*, pages 1–11, 2018.

[63] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process*, 25(6):575–599, 2013.

[64] Lisong Guo, Julia Lawall, and Gilles Muller. Oops! Where did that code snippet come from? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 52–61. ACM, 2014.

[65] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common C language errors by deep learning. In *AAAI*, pages 1345–1351, 2017.

[66] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep Speech: Scaling up end-to-end speech recognition. Preprint, 2014.

[67] Mark Harman and Afshin Mansouri. Search based software engineering: Introduction to the special issue of the ieee transactions on software engineering. *IEEE Transactions on Software Engineering*, 36:737–741, 11 2010.

[68] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[69] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[70] Aslak Hellesøy. I'm sloppy · cucumber-attic/bool@33af59d. https://github.com/cucumber-attic/bool/commit/33af59d3dac87491c0b22b6248e54ba7b02a367e, March 2013. (Accessed on 02/08/2018).

[71] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[72] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780, 2014.

[73] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 197–207. ACM, 2017.

[74] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference On*, pages 837–847, 2012.

[75] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[76] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[77] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156, 2003.

[78] Bo-June Paul Hsu and James R. Glass. Iterative language model estimation: efficient data structure & algorithms. In *INTERSPEECH*, pages 841–844, 2008.

[79] James Jackson, M. J. Cobb, and Curtis Carver. Identifying top Java errors for novice programmers. In *Frontiers in Education Conference*, volume 35, page T4C. STIPES, 2005.

[80] Matthew C. Jadud. A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1):25–40, 2005.

[81] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, pages 73–84. ACM, 2006.

[82] Clinton L. Jeffery. Generating LR Syntax Error Messages from Examples. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):631–640, 2003.

[83] JetBrains s.r.o. intellij-community/psielement.java at master · jetbrains/intellij-community. https://github.com/JetBrains/intellij-community/blob/fc280fb011cd526fcda2ed2664e5b02b70a0c415/platform/core-api/src/com/intellij/psi/PsiElement.java, March 2018. (Accessed on 2018-05-04).

[84] Huzefa Kagdi, Jonathan I Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 145–154. IEEE, 2007.

[85] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101. ACM, 2014.

[86] Foutse Khomh, Brian Chan, Ying Zou, and Ahmed E. Hassan. An entropy evaluation approach for triaging field crashes: A case study of Mozilla Firefox. In *2011 18th Working Conference on Reverse Engineering (WCRE)*, pages 261–270, 2011.

[87] Dongsun Kim, Xinming Wang, Sunghun Kim, A. Zeller, S.C. Cheung, and Sooyong Park. Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

[88] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pages 486–493, 2011.

[89] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. Available: http://arxiv.org/abs/1412.6980, 2014.

[90] Nathan Klein, Christopher S. Corley, and Nicholas A. Kraft. New features for duplicate bug detection. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 324–327. ACM, 2014.

[91] Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995.

[92] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[93] Sarah K. Kummerfeld and Judy Kay. The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20*, pages 105–111. Australian Computer Society, Inc., 2003.

[94] Lamdu. http://www.lamdu.org/, 2017. (Accessed on 09/26/2017).

[95] Alina Lazar, Sarah Ritchey, and Bonita Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 308–311. ACM, 2014.

[96] Johannes Lerch and Mira Mezini. Finding duplicates of your yet unwritten bug report. In *2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 69–78, 2013.

[97] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[98] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

[99] Chao Liu and Jiawei Han. 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, pages 46–56. ACM, 2006.

[100] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 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, pages 286–295. ACM, 2005.

[101] TFIDFSimilarity (Lucene 4.9.0 API). https://lucene.apache.org/core/4_9_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html. (Accessed on 05/24/2017).

[102] Lucene's practical scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/1.x/practical-scoring-function.html, January 2015. (Accessed on 2018-03-28).

[103] Marco Lui and Timothy Baldwin. langid.py: An off-the-shelf language identification tool. In *Proceedings of the ACL 2012 system demonstrations*, pages 25–30. Association for Computational Linguistics, 2012.

[104] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing.* The MIT Press, Canbridge, MA, 1 edition, May 1999.

[105] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 499–504. ACM, 2011.

[106] Bernard Marr. 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, March 2015.

[107] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, pages 131–140, 2009.

[108] Apache Maven project. https://maven.apache.org/, March 2018. (Accessed on 2018-03-28).

[109] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[110] Linda McIver. The effect of programming language on error rates of novice programmers. In *12th Annual Workshop of the Psychology of Programming Interest Group*, pages 181–192. Citeseer, 2000.

[111] Natwar Modani, Rajeev Gupta, Guy Lohman, Tanveer Syeda-Mahmood, and Laurent Mignet. Automatically identifying known software problems. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 433–441, 2007.

[112] Akira Moroo, Akiko Aizawa, and Takayuki Hamamoto. Reranking-based crash report deduplication. http://ksiresearchorg.ipage.com/seke/seke17paper/seke17paper_135.pdf, 2017.

[113] Mozilla Corporation. Mozilla Crash Reports. http://crash-stats.mozilla.com, 2012.

[114] Mozilla Corporation. mozilla/socorro: Socorro is a server to accept and process Breakpad crash reports, 2016.

[115] Mozilla Corporation. About :: Bugzilla. https://www.bugzilla.org/about/, 2017. (Accessed on 05/10/2017).

[116] Felix Mulder. Awesome error messages for Dotty, October 2016. Available: http://scala-lang.org/blog/2016/10/14/dotty-errors.html.

[117] Tim Muller, Dongxia Wang, and Audun Jøsang. Information theory for subjective logic. In Vicenc Torra and Torra Narukawa, editors, *Modeling Decisions for Artificial Intelligence*, pages 230–242, Cham, 2015. Springer International Publishing.

[118] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 455–458. IEEE, 2017.

[119] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? *SIGCSE Bull.*, 40(1):168–172, March 2008.

[120] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. Toward semantic foundations for program editors. Preprint, 2017.

[121] Gs-803 build version advanced to 11501-11 · openspaces/openspaces@26e900e. https://github.com/OpenSpaces/OpenSpaces/commit/26e900e32d07ab6747d9092b929194076a575632, February 2014. (Accessed on 02/08/2018).

[122] Oracle Corporation. The Java programming language compiler group. http://openjdk.java.net/groups/compiler/, 2017. (Accessed on 08/04/2017).

[123] Klérisson VR Paixão, Crícia Z Felício, Fernanda M Delfim, and Marcelo de A Maia. On the interplay between non-functional requirements and builds on continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 479–482. IEEE Press, 2017.

[124] Terence Parr and Kathleen Fisher. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[125] Gustavo Pinto, Luiz Fronchetti Dias, and Igor Steinmacher. Who gets a patch accepted first? comparing the contributions of employees and volunteers. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'2018)*, 06 2018. https://www.researchgate.net/publication/323693078_Who_Gets_a_Patch_Accepted_First_Comparing_the_Contributions_of_Employees_and_Volunteers.

[126] Tom Preston-Werner. Semantic versioning 2.0.0. http://semver.org/spec/v2.0.0.html, 2017. Accessed on 2017-04-24.

[127] David Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 1–8. ACM, 2015.

[128] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[129] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428. ACM, 2014.

[130] Gregorio Robles, Jesús M. González-Barahona, Daniel Izquierdo-Cortazar, and Israel Herraiz. Tools and datasets for mining libre software repositories. *Multi-Disciplinary Advancement in Open Source Software and Processes*, page 24, 2011.

[131] Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval.* McGraw-Hill computer science series. McGraw-Hill, 1983.

[132] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. Syntax and Sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322. IEEE, March 2018.

[133] Eddie Antonio Santos and Abram Hindle. Judging a commit by its cover: correlating commit message entropy with build status on Travis-CI. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 504–507. ACM, 2016.

[134] Andrian Schröter, Nicholas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 118–121, 2010.

[135] Hyunmin Seo and Sunghun Kim. Predicting recurring crash stacks. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 180–189. ACM, 2012.

[136] Claude E. Shannon. Prediction and entropy of printed English. *Bell Labs Technical Journal*, 30(1):50–64, 1951.

[137] Henry Maurice Sheffer. A set of five independent postulates for boolean algebras, with application to logical constants. *Transactions of the American Mathematical Society*, 14(4):481–488, 1913.

[138] Stanley Shyiko. [maven-release-plugin] prepare for next development iteration · shyiko/mappify@0bfac70. https://github.com/shyiko/mappify/commit/0bfac7039a7951bcc47c0ae4e4c6f8e201161a0e, November 2013. (Accessed on 02/08/2018).

[139] Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449. ACM, 1992.

[140] Slashdot Media. About SourceForge. https://sourceforge.net/about, 2017. (Accessed on 05/09/2017).

[141] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR ’05, pages 1–5. ACM, 2005.

[142] Stewart D. Smith, Nicholas Zemljic, and Andrew Petersen. Modern goto: Novice programmer usage of non-standard control flow. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling ’15, pages 171–172, New York, NY, USA, 2015. ACM.

[143] Software in the Public Interest. Jenkins. https://jenkins.io/, 2017. (Accessed on 05/11/2017).

[144] SPARQL—Semantic Web standards. https://www.w3.org/2001/sw/wiki/SPARQL, 2013. (Accessed on 05/26/2017).

[145] StackOverflow—where developers learn, share, and build careers. https://stackoverflow.com/, 2017. Accessed on 2018-04-04.

[146] Antonin Stefanutti. Add test for visibility modifiers · astefanutti/metrics-cdi@bde5636. https://github.com/astefanutti/metrics-cdi/commit/bde563637902237aca2fc6a7e49b4b57099b11b1, November 2013. (Accessed on 02/08/2018).

[147] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. Identifying at-risk novice Java programmers through the analysis of online protocols. In *Philippine Computing Science Congress*, 2008.

[148] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. 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, pages 85–92, New York, NY, USA, 2011. ACM.

[149] Ole Tange. GNU parallel—the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[150] The Eclipse Foundation. 4.7 - Eclipse project downloads. http://download.eclipse.org/eclipse/downloads/drops4/R-4.7-201706120950/#JDTCORE, 2017. (Accessed on 08/09/2017).

[151] The Unicode Consortium. Unicode standard annex #15, “Unicode normalization forms,” an integral part of The Unicode Standard. http://unicode.org/reports/tr29, 2015.

[152] The Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, Mountain View, CA, 2016.

[153] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. Preprint, May 2016.

[154] Richard A. Thompson. Language correction using probabilistic grammars. *IEEE Transactions on Computers*, 100(3):275–286, 1976.

[155] T. Tieleman and G. Hinton. RMSprop gradient optimization. Course slides, April 2014.

[156] Travis CI GmbH. Customizing the build—Travis CI. https://docs.travis-ci.com/user/customizing-the-build/#Breaking-the-Build, 2017. (Accessed on 2017-04-24).

[157] Travis CI GmbH. Travis CI—test and deploy with confidence. https://travis-ci.org/, 2017. (Accessed on 2017-04-24).

[158] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM, 2014.

[159] Jonathan Turner. Shape of errors to come—the Rust programming language blog, August 2016. Available: https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html.

[160] Shaohua Wang, Foutse Khomh, and Ying Zou. Improving bug localization using correlations in crash reports. In *2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 247–256, 2013.

[161] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[162] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345, 2015.

[163] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214. ACM, 2014.

[164] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

Appendix A

Searching for LSTM hyperparameters using grid search

In Chapter 5, we presented a long short-term memory (LSTM) model that aided in the task of finding and suggesting fixes to syntax errors. In this appendix, we describe the process we used to find its *hyperparameters*.

Neural networks, including LSTMs, are not trainable models in-and-of-themselves, but rather a *framework* for creating trainable models. Thus, to train a neural network, one must first decide:

- What kind and how many neural network layers to use
- How many neurons in each hidden layer
- Whether to use any kind of *regularization*
- How many *epochs*—full iterations of the corpus—to train on
- How to update the weights given each batch of training data

These decisions and many others, made prior to training, are called *hyperparameters*—parameters of the neural network framework itself. These are not to be confused with a neural network's *parameters*—the weights and biases that must be found once a network architecture is finalized.

Our initial models adapted hyperparameters pertaining to model architecture from the empirical work of White et al. [162] on training recurrent neural networks on source code. This meant using one recurrent neural network layer with 300 nodes in the hidden layer and 20 time steps of context. However, there are many difference between our modelling goals and those of White et al. The models trained in the prior work were intended for the task of code completion. Consequently, the models trained by White et al. kept all concrete tokens, whereas our approach abstracts certain tokens. How our work and the prior work measure what is optimum also differs: White et al. reported the best hyperparameters in terms of the minimizing the per-token perplexity of the models; our work is interested in maximizing the amount of syntax errors fixed in erroneous files (described in greater detail in Section 5.7). Another critical difference is that White et al. trained traditional recurrent neural networks, whereas we opted for the more modern LSTMs. As a result of these differences, we set off to find more appropriate hyperparameters.

A.1 Grid Search

Nodes in LSTM layer	50, 100, 200, 300, 400, 1000
Tokens of context	5, 10, 15, 20
Dropout	None, 0.75
Patience	10, 20
Optimizer	Adam, RMSprop

Table A.1: Hyperparameters and chosen values for the initial grid search

To search for potentially more optimal hyperparameters for our LSTM neural network, we employed *grid search*. Grid search (also known as a *parameter sweep*) involves exhaustively evaluating every possible combinations of hyperparameters out of a set of manually-specified values. Table A.1 lists all the hyperparameters we manipulated, along with the values we attempted. We wanted to determine how many nodes were required to learn syntax in the hidden layer; we were also interested in knowing how many tokens of context are required to reasonably learn the syntax of the language. As for the training process itself, we were interesting in whether adding a dropout “layer” (for regularization) improved results at all; as such, we tried adding a dropout after the LSTM layer with a holding parameter of 0.75, and we tried training models without any form of regularization at all. We were interested in how many training epochs were required before the models overfit to the training data. For this, we adjusted the patience of early stopping—that is, how many epochs to wait after a lull in training progress before terminating. Finally, we wanted to ascertain which optimizer—the method of tweaking model parameters after each training batch—would find the best model parameters. The optimizers we evaluated were RMSprop [155]—which is commonly suggested as the way to optimize recurrent neural network (RNN) weights—and Adam [89].

The Cartesian product of the manipulated variables in Table A.1 resulted in the evaluation of 192 configurations. For each configuration, we trained five different pairs of models for each partition (Section 5.7.1) of the Java training corpus, to determine how well the results generalize given different training data. Thus, we trained 960 distinct pairs of models in total.

We fixed the number of training files to 32 files per partition, the learning rate to 0.001, and the batch size to 32. To rank models, we compared their ability to find the *exact location* of an error, and their ability to find the *true fix* of a location (both metrics are further described in Section 5.7). We chose to evaluate on fixing 32 files from the mistake corpus (Section 5.6).

A.2 Results

Hidden layer	Context length	Dropout	Patience	Optimizer	Location MRR	Valid fix MRR	True fix MRR
200	5	None	20	Adam	0.35	0.35	0.34
50	20	0.75	10	Adam	0.37	0.37	0.34
300	20	None	20	Adam	0.36	0.36	0.34

Table A.2: The top two model configurations and the original configuration, ordered by true fix MRR (higher is better)

The top two best performing model configurations according to highest valid fix mean reciprocal rank (MRR) are listed in Table A.2. Interestingly, a configuration similar to that found by White et al. with 300 nodes in the hidden layer and 20 tokens of context is the sixth best performer (the last row of the table).

We fit a linear model to determine the effect that various factors had on the mean reciprocal rank of finding the true fix and valid fix. In both cases, we found that the training partition was a significant factor ($p < 0.001$ for all partitions, treated as a categorical variable). We concluded that, with such a small data size, variations in the types of source code files in the five sets of 32 training files affected how models learned Java syntax. The other significant factors were the choice of optimizer, and the presence of dropout.

We found the models trained with Adam performed better than the equivalent model trained with RMSprop in 94 out of 96 configurations (97.9%), with a 95% confidence interval of the mean improvement to true fix MRR in (0.091–0.11). We concluded that training with Adam unambiguously converges faster than RMSprop, hence, in all larger experiments, we used the Adam optimizer exclusively.

Dropout is a commonly suggested way to perform *regularization*—a way to reduce overfitting when training neural networks. However, prior work [164] has claimed that, without special consideration with regards to recurrent connections, dropout is ineffective when training RNN or LSTM networks. Our evaluation provides support for this claim: we found that models trained *without* any kind of regularization outperformed equivalent models with 75% dropout, in 71 out of 96 configurations (74.0%), with a 95% confidence interval of the mean improvement to true fix MRR in (0.017–0.032).

To understand the effect of training on a larger dataset, we graduated the models listed in Table A.2 to a set with 512 training files. We also increased the amount of mistakes to evaluate on to 512 to match. The results of these graduated models is listed in Table A.3.

Hidden layer	Context length	Location MRR	Valid fix MRR	True fix MRR
50	20	0.45	0.45	0.39
300	20	0.46	0.45	0.39
200	5	0.42	0.42	0.36

Table A.3: The model configurations trained on a 512 file training set, ordered by true fix MRR (higher is better)

As expected, the effects of the partition were diminished: only one partition had a coefficient $p < 0.05$ on valid fix MRR, and none had a significant coefficient for true fix MRR. In general, all models performed better than their counterparts on smaller data.

We further graduated the top-two models from Table A.3 to the highest training set size of 11,000 files. In an effort to decrease training time, we decided to only graduate the models with the highest mean true fix MRR and mean valid fix MRR; thus, we eliminated the $200, 5$ configuration from further experiments.

We evaluated the $300, 20$ and $50, 20$ configurations on a training set size of 11,000 files. Table A.4 shows the results for each partition; the results seem to suggest that the $300, 20$ may yield a better True Fix MRR. To provide evidence for this, we performed a paired, one-tailed t-test comparing the true fix MRR of the two configurations. The null hypothesis is that the difference in MRR between both configurations is zero. We obtained a p-value of 0.0046. Using a confidence level of 0.01, we have to reject the null hypothesis, and conclude that the difference in true fix MRR is greater than zero, with the $300, 20$ configuration having a higher MRR. Thus, we chose to use the $300, 20$ configuration

Hidden layer	Context length	Partition	Location MRR	Valid fix MRR	True fix MRR
300	20	2	0.53	0.53	0.47
300	20	1	0.52	0.52	0.46
300	20	3	0.53	0.52	0.46
50	20	2	0.51	0.51	0.45
300	20	5	0.50	0.50	0.44
300	20	4	0.50	0.49	0.44
50	20	3	0.49	0.49	0.44
50	20	1	0.48	0.47	0.42
50	20	4	0.46	0.46	0.41
50	20	5	0.44	0.43	0.39

Table A.4: The model configurations trained on the full 11,000 file training set. ordered by true fix MRR (higher is better). Results are shown by partition.

as our final LSTM configuration. Interestingly, it is quite similar to the model described by White et al., with a few exceptions:

- We used the LSTM instead of regular RNN networks
- We used the Adam optimizer, instead of RMSprop.
- We increased the patience to 20 epochs, of which only 13 epochs were typically required to converge.

The evaluation of the chosen $300, 20$ configuration is described in greater detail in Section 5.8.