Finding and correcting syntax errors using recurrent neural networks

Eddie Antonio Santos[1], **Joshua Charles Campbell**[1], **Abram Hindle**[1], **and José Nelson Amaral**[1]

[1]**Department of Computing Science, University of Alberta, Edmonton, Canada**

ABSTRACT

Minor syntax errors are made by novice and experienced programmers alike; however, novice programmers lack the years of intuition that help them resolve these tiny errors. Standard LR parsers typically resolve syntax errors and their precise location poorly. We propose a methodology that helps locate where syntax errors occur, but also suggests possible changes to the token stream that can fix the error identified. This methodology finds syntax errors by checking if two language models "agree" on each token. If the models disagree, it indicates a possible syntax error; the methodology tries to suggest a fix by finding an alternative token sequence obtained from the models. We trained two LSTM (Long short-term memory) language models on a large corpus of JavaScript code collected from GitHub. The dual LSTM neural network model predicts the correct location of the syntax error 54.74% in its top 4 suggestions and produces an exact fix up to 35.50% of the time. 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.

Keywords: syntax error correction; LSTM; JavaScript; RNN; syntax error; deep learning; neural network; n-gram; program repair; GitHub

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 programming languages for decades. Yet, plain text can be a major hurdle for novices learning how to code (Tabanao et al., 2008; Jadud, 2005; Tabanao et al., 2011). 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 (Tabanao et al., 2008), get it wrong.

The tools that interpret 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 JavaScript source code in Listing 1. A single *token* — a { at the end of line 1 — in the input is different from that of the correct file that the programmer intended to write. Give this source file to a JavaScript interpreter such as Mozilla's SpiderMonkey (Mozilla Developer Network Contributors, 2016), and it reports that there is an error, but provides misleading help to identify the location of the mistake made by the programmer.

```
1 if (process.argv.length < 3)
2   console.error("Not enough args!");
3   process.exit(1);
4 }
```

Listing 1. Syntactically invalid JavaScript code. An open brace ({) is missing at the end of line 1.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

```
broken.js:4: SyntaxError: syntax error:
broken.js:4: }
broken.js:4: ^
```

Node.js (Node.js Foundation, 2017) (using Google's V8 JavaScript engine (Google Developers, 2016)) fares even worse—stating that the syntax error is on sixth line of a four-line file:

```
/home/nobody/broken.js:6
});
^

SyntaxError: Unexpected token }
    at checkScriptSyntax (bootstrap_node.js:457:5)
    at startup (bootstrap_node.js:153:11)
    at bootstrap_node.js:575:3
```

Unbalanced braces are *the* most common error among new programmers (Brown and Altadmri, 2014). Imagine a novice programmer writing their for-loop for the first time and being greeted with `Unexpected token }` as their welcome to computer programming. SpiderMonkey suggests the programmer has *added* an extra right curly brace, despite the fact that the programmer actually forgot a left curly brace. On top of that, SpiderMonkey identified the problem as being on line 4 when the mistake is actually on line 1. 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 1!"

In this paper, we created a tool called *GrammarGuru* that seeks to find and fix **single token syntax errors**, using long short-term memory neural network (LSTM) language models trained on a large corpus of hand-written JavaScript source code. 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?

The intuition for *GrammarGuru*, a syntax error detector and corrector, is that if a tool mimics the mental model of an expert programmer *GrammarGuru* can find and perhaps correct syntax errors like an expert programmer. When expert programmers try to find a syntax error, often they read the source code and find the section of the source code that they do not understand, or code that surprises them. Then they fix that source code in order to compile or run it. The mental model of the programmer may be something like a language model for speech, but rather applied to code. *Language models* are typically applied to natural human utterances but they have also been successfully applied to software (Hindle et al., 2012; Raychev et al., 2014; White et al., 2015), and can be used to discover unexpected segments of tokens in source code (Campbell et al., 2014).

Parsers typically lack probabilistic language models and rely on formal grammar, unsuited to handle errors. Parsers sometimes get lost in a production that was unintended. A language model can tell us if a token is surprising or rare. Multiple language models can be combined to determine agreement. Thus *GrammarGuru* uses language models to capture code regularity or naturalness and then looks for irregular code (Campbell et al., 2014). Once the location of a potential error is found, code completion techniques that exploit language models (Hindle et al., 2012; Raychev et al., 2014; White et al., 2015) can be used to suggest possible fixes. Traditional parsers do not rely upon such information.

The primary contribution in this paper is the use of *two* language models working in opposite directions. One model predicts what comes *after* a sequence of tokens, and the other predicts what comes *before* a sequence of tokens. Using two recurrent neural network hidden layers—one working "forwards in time" and one working "backwards in time"—has been used in speech recognition (Hannun et al., 2014; Amodei et al., 2015); however, to our knowledge this concept has yet to be applied to source code. Using the predictions of the two language models, our command-line tool suggests the fixes for single-token syntax errors.

2 PRIOR WORK

A number of researchers have sought to improve error messages as a result of incorrect syntax. The long history of work on syntax error messages is often motivated by the need to better serve novice programmers (Tabanao et al., 2008; Jadud, 2005; Tabanao et al., 2011; Jadud, 2006; Jackson et al., 2005;

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

Garner et al., 2005; McIver, 2000; Kummerfeld and Kay, 2003; Hristova et al., 2003). More recently, Denny et al. (Denny et al., 2012) 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—such as missing semicolons, extraneous tokens, and missing close brackets. Dy and Rodrigo (Dy and Rodrigo, 2010) 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. (Nienaltowski et al., 2008) 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. (Marceau et al., 2011) developed a rubric for grading the error messages produced by compilers. Becker's dissertation (Becker, 2015) attempted to enhance compiler error messages in order to improve student performance. Barik et al. (Barik et al., 2014) studied how developers visualize compilation errors. They motivate their research with an example of a compiler misreporting the location of a fault. Pritchard (Pritchard, 2015) shows that the most common type of errors novices make in Python are syntax errors.

Earlier research attempted to tackle errors at the parsing stage. In 1972, Aho (Aho and Peterson, 1972) introduced an algorithm to attempt to repair parse failures by minimizing the number of errors that were generated. In 1976, Thompson (Thompson, 1976) 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. (Parr and Fisher, 2011) 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, replacement and predictive parsing to produce better error messages. In essence, 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 (Jeffery, 2003) 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.

More recent research has applied language models to syntax error detection and correction. Campbell et al. (Campbell et al., 2014) 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 would augment 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. Our work trains language models with a bounded vocabulary using deep-learning LSTM recurrent neural networks (described in Section 3). In contrast, UnnaturalCode's *n*-gram models use an unbounded vocabulary. Gupta et al. (Gupta et al., 2017) used an encoder-decoder framework with Gated Recurrent Units (GRU)—a type of recurrent neural network—to detect and correct compiler errors in C source code... Bhatia and Singh (Bhatia and Singh, 2016)...

Others have applied recurrent neural networks (RNNs) to source code. White et al. (White et al., 2015) trained RNNs on source code and showed their practicality in code completion. Similarly, Raychev et al. (Raychev et al., 2014) used RNNs in code completion to synthesize method call chains in Java code.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

Dam et al. (Dam et al., 2016) provides a good overview of using 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, *GrammarGuru* may suggest how to complete it; however, our focus is on syntax errors, rather than helping complete code as it is being written.

Others still have considered making syntax errors impossible to make, even in text-based languages, by blurring the line between the program editor and the language itself. Omar et al. (Omar et al., 2017) proposed Hazel, a new programming language and environment, which offers formal semantics between edit states in the language and defines explicit "holes" in the abstract syntax tree where the program is incomplete.

Numerous compilers have placed a focus on more user-friendly error messages that explain the error and provide solutions. Among these are Clang (Clang, 2016), Rust (Turner, 2016), Scala (Mulder, 2016), and Elm (Czaplicki, 2015). Our work is directly influenced by the "fix-it hints" provided by Clang.

3 METHODOLOGY

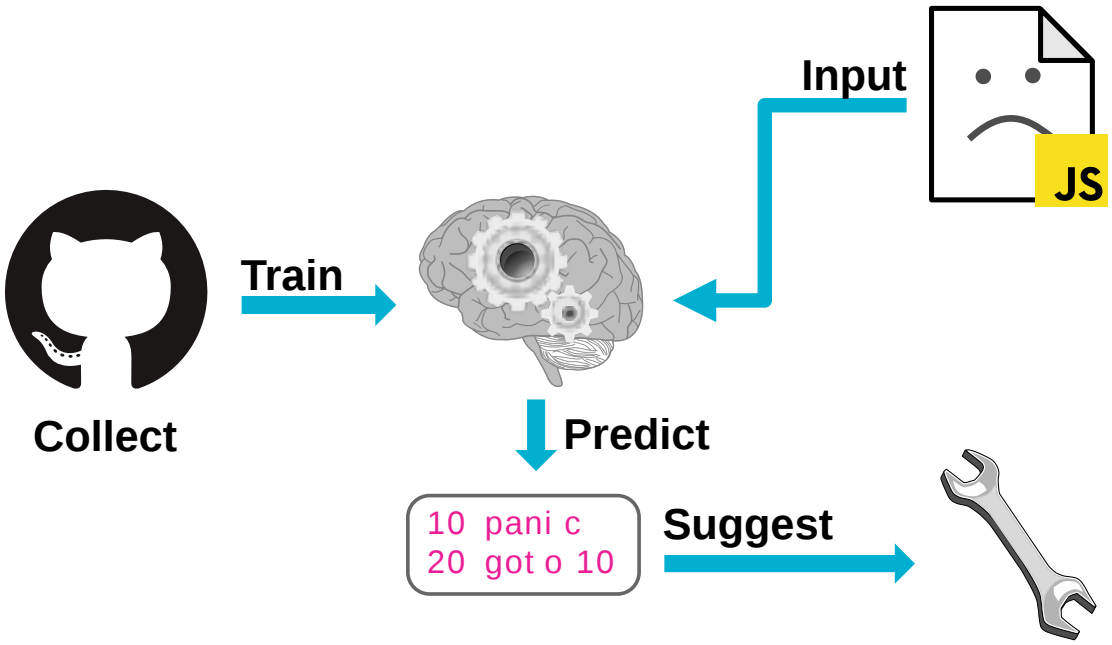

Figure 1. An overview of our methodology. First we trained two long short-term memory (LSTM) neural networks on JavaScript source code collected from GitHub. Then we used the LSTMs to predict tokens from JavaScript files with syntax errors. We applied a naïve heuristic that relies on the disagreement between the LSTMs to pinpoint the syntax error location and suggest possible fixes.

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 1).

$$P(\textit{adjacent-token}|\textit{context}) \tag{1}$$

We trained long short-term memory (LSTM) recurrent neural networks to approximate the function in Equation 1. In order to train the LSTM, we needed a vast corpus of positive examples. For this, we mined a large corpus of over nine thousand of the most popular open source JavaScript repositories from GitHub (Section 3.1). This source code was tokenized (Section 3.2) such that it could be used to train the JavaScript language model (Section 3.5). Finally, we used the approximated functions expressed in Equation 1 to detect a syntax error in a file (Section 3.6) and suggest a plausible fix (Section 3.7). Figure 1 graphically describes this process.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

3.1 Building a corpus through mining

To obtain the training data, we downloaded JavaScript source code from GitHub. Since we required JavaScript tokens from each file, other GitHub mining resources such as Boa (Dyer et al., 2013) and GHTorrent (Gousios, 2013) 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 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 `.js`, storing a SHA-256 hash to avoid storing byte-for-byte duplicate files. We used Esprima 3.1.1 (Hidayat, 2016), a high-performance parser for JavaScript, to tokenize and parse every JavaScript file downloaded. Fully-parsing the JavaScript source code allowed us to filter only the JavaScript files that were syntactically valid according to the 7th edition of the ECMAScript standard (ECMA TC39, 2016), commonly known as ES2016 or ES7. In total, we tokenized 494,352 syntactically-valid JavaScript files (as parsed by Esprima) out of 594,681 total `.js` files downloaded.

We further reduced the number of files used, both in training, validation, and testing, by using a simple heuristic to remove *minified* source code files. *Minification* is JavaScript source code obfuscation that attempts to reduce the file size of the source code. Often, minification eliminates any whitespace that does not make a difference to the syntax of the code. However, some minification strategies, such as replacing `undefined` (9 bytes) with the semantically-equivalent expression `void 0` (6 bytes), alter the token stream of the code, and hence do not represent "handwritten JavaScript". Our simple heuristic simply discarded all files whose filename ends with `.min.js`, as this is a common convention for indicating minified JavaScript code (Rutter, 2013). Finally, after finding and removing 14,354 minified files, we were left with 479,998 JavaScript source code files available for training, model validation, and evaluation.

All relevant data—repository metadata, source code, and repository licenses—were stored in an SQLite3 database.[1]

3.2 Tokenization

Min	Max	Median	Skew	Kurtosis
0	1,809,948	248.00	33.25	1,721.98

Table 1. A summary of the number of tokens per file

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.

The set of all possible **unique** tokens tracked by a language model is called the *vocabulary*. Every new source file likely contains novel variable names or string literals that have never been seen before. Thus, to keep a generally small, bounded vocabulary that has enough unique tokens to faithfully represent the syntax and regularity of handwritten JavaScript, we present the concept of *open and closed classes*.

3.3 Open and closed classes

A natural-language analogy provides the motivation behind open and closed classes. In English, there are different word classes such as nouns, verbs, adjectives, determiners, prepositions, and conjunctions. Some of these word classes are *open*, such as nouns, verbs, and adjectives. They are open classes because it is quite easy to coin new nouns, adjectives, and verbs, and have them become words that are widely used by a community of people. However, other word classes in English are *closed* such as determiners ("the", "that", "this"), conjunctions ("and", "or", "but"), and prepositions ("of", "for", "with"). It is much more difficult to invent a new determiner with its own meaning independent of the existing determiners and furthermore convince all your friends to use it.

Similarly, tokens in JavaScript source code can be classified as either belonging to an open or to a closed class. The goal of the proposed method is to merely learn the syntax of JavaScript and does not

[1] Available: https://archive.org/details/sensibility-replication-package

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

Esprima kind	Class	Examples
Boolean	Closed	`true`, `false`
Keyword	Closed	`if`, `for`, `with`, `new`, `this`
Null	Closed	`null`
Punctuator	Closed	`{`, `+`, `;`, `**=`
Identifier	Open	`undefined`, `$`, `buñuelo`
Numeric	Open	`42`, `0xC0FFEE`, `0755`
RegularExpression	Open	`/a*|(ab)+/gi`
String	Open	`'hello'`, `"hello"`
Template	Open	`` `hello` ``, `` `hello ${}!` ``

Table 2. Esprima token kinds. Any token text belonging to closed classes are used verbatim, whereas tokens from open classes were abstracted.

require predicting the *exact* next token text of an identifier, or a literal. Therefore, we can abstract each open class with a single vocabulary entry. For closed classes, there is a small and finite number of unique tokens that belong to each class. Thus, we entered these items verbatim into the vocabulary.

Esprima provides tokens as the triple: (kind, text, location). We can determine whether a token belongs to an open or a closed class by simply examining its kind. Table 2 shows every Esprima token kind, whether its tokens belong to an open or a closed class, and some illustrative examples.

The vocabulary used to build the LSTM models was discovered by exhaustively iterating through all 1.58 billion tokens in the corpus (Table 1), and then adding each unique token to the vocabulary. The result was 98 unique tokens from the ECMAScript 2016 standard (ECMA TC39, 2016). In addition to these tokens, we added synthetic `/*<START>*/` and `/*<END>*/` tokens such that we could encode beyond the start and end of a source file, respectively. Thus, the total size of our vocabulary was 100 unique tokens.[2]

3.4 Tokenization Pipeline

Original source code	`var greeting = 'hello'`
Esprima tokenization	Keyword(`"var"`), Identifier(`"greeting"`), Punctuator(`"="`), String(`"'hello'"`)
Vocabulary normalization	`var` Identifier = String
Vectorization	$\begin{bmatrix}3\\0\\2\\1\end{bmatrix}$
One-hot vectorization	$\begin{bmatrix}0&1&0&0\\0&0&0&1\\0&0&1&0\\1&0&0&0\end{bmatrix}$

Table 3. The series of token transformations from source code to one-hot matrices suitable for training the LSTM models. The simplified vocabulary indices are "Identifier" = 0, "String" = 1, "=" = 2, and "`var`" = 3.

To convert a source file to a form that is suitable for training the LSTMs, we performed a series of transformations (Table 3). The raw source code was tokenized in a form that is suitable as input for a parser. As mentioned in Section 3.1, this was done for each JavaScript file using Esprima. Then, we normalized the token stream such that each token was an entry of the vocabulary; consequently, the exact text of tokens belonging to open classes was discarded. Each token in the vocabulary is assigned a non-negative integer index. Each file was converted into a numeric vector, representing the tokens of an entire file. Each subsequent token in the file was replaced with its corresponding numeric index into the vocabulary. 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.

[2] Available: https://github.com/eddieantonio/training-grammar-guru/blob/icsme2017/sensibility/js_vocabulary.py

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

```
1 if (window === undefined) {
2   this.isBrowser = false;
3 }
```

Listing 2. Syntactically-Valid Javscript

Thus, the matrix has as many columns as tokens in the file and one row for each vocabulary entry. 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.

3.5 Training the LSTMs

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. (Campbell et al., 2014). However, to fix errors, it is also necessary to know what tokens actually *are* likely for each given context. We rephrase Equation 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 2).

$$adjacent\,[context] = \begin{cases} P(\texttt{if}|context) \\ P(\texttt{else}|context) \\ P(\text{Identifer}|context) \\ P(\text{Number}|context) \\ \dots \\ P(\}|context) \end{cases} \tag{2}$$

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 3.7).

To approximate such a function, we used machine learning to map contexts to categorical distributions of the adjacent token. For this task, we employed long short-term memory (LSTM) recurrent neural networks, as they were successfully used by prior work in predicting tokens from source code (Raychev et al., 2014; White et al., 2015). 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 (Hannun et al., 2014; Amodei et al., 2015), 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 using just these two models.

As an example, consider the syntactically-valid JavaScript snippet in Listing 2. Figure 2 illustrates the contexts—both prefix and suffix—when estimating the likelihood of the open brace ({) on the first line of Listing 2.

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 empirical work done by White et al. (White et al., 2015) we chose a context length τ of 20 tokens. This corresponds to an n-gram length of 21 tokens, as an n-gram traditionally 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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

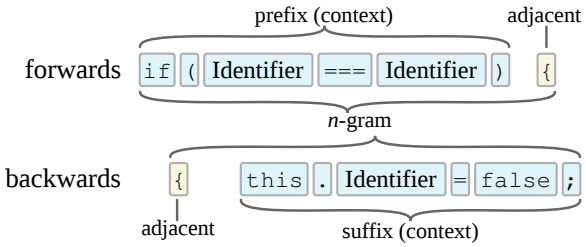

Figure 2. The relationship between an n-gram, the contexts, and the adjacent token. In this diagram, $n = 7$, and the adjacent token is the { in both cases. Thus, the contexts (both prefix and suffix) are $n - 1$ or 6 tokens long.

start and end of the file, we inserted synthetic `/*<START>*/` and `/*<END>*/` tokens, collectively called *padding* tokens. This means that the first prefix context in the file is comprised entirely of 20 `/*<START>*/` tokens; likewise, the last suffix context in the file is comprised of 20 `/*<END>*/` tokens.

Input	One-hot matrix, dimensions = $\tau \cdot \lvert V \rvert$			
	Type	Output dim.	Activation	Recurrent activation
Layers	LSTM	$\tau \cdot 300$	tanh	hard sigmoid
	LSTM	300	tanh	hard sigmoid
	Dense	$\lvert V \rvert$	softmax	
Output	Categorical distribution, size = $\lvert V \rvert$			
Loss	Categorical cross-entropy			
Optimizer	RMSprop, initial learning rate = 0.001			

Table 4. Summary of the neural network architecture we trained. $\lvert V \rvert = 100$ is the size of the vocabulary (Section 3.3) and $\tau = 20$ is the length of each context in number of tokens.

We used Keras 1.2.2 (Chollet, 2015), a Python deep neural network framework, to define our model architecture, using the Theano (Theano Development Team, 2016) backend to train the models proper. A summary of the precise architecture that we used is given in Table 4. Each LSTM layer has 300 outputs, based on the observation by White et al. (White et al., 2015) that recurrent neural networks with 300 outputs with 20 tokens of context or 400 output with 5 tokens of context have the lowest perplexity with respect to the corpus of source code. We used the RMSprop (Tieleman and Hinton, 2014) 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 three consecutive epochs (its *patience* parameter) that yield no improvement to the categorical cross-entropy with respect to the validation examples.

We arrived at this neural network architecture by starting with the work of White et al. (White et al., 2015) and using LSTMs rather than plain recurrent neural networks (RNNs). We tried models in which we varied the amount of neurons in the hidden layer, as well as keeping the number neurons per layer constant while adjusting the number of layers. We settled on the architecture of two LSTM layers of 300 neurons each due to it consistently achieving the lowest loss across validation partitions.

Once each model was trained, it was serialized in Hierarchical Data Format 5 (HDF5). In total, the weights and biases of each individual model resulted in 9.4 MiB of data.[3] Section 5.1 discusses how files were chosen for the training, validation, and testing sets.

3.6 Detecting syntax errors

We use the output of the two models trained in Section 3.5 to find the likely location of the syntax error. Given a file with one syntax error, we tokenize it using Esprima. Esprima is able to tolerate erroneous

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

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

input in the tokenization phase, which produces a token stream (as opposed to the full 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 use the two independent probability distributions to determine which tokens are quantifiably unlikely, or "unnatural" (Campbell et al., 2014). We use two measures—total variation distance and the joint probability—and combine them using an average for each token in the file. Once the naturalness of every single token in the file is computed, 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.

To obtain a "global" measure of the difference at a particular token location, we use the *total variation distance* to compare the difference of the two independent categorical distributions. For two discrete distributions n and p, both with cardinality $|V|$, the total variation distance is defined as half the ℓ_1 norm of the pairwise difference of elements in the distribution (Gibbs and Su, 2002), written as:

$$global_t = \frac{1}{2}\sum_{i=1}^{|V|} |n_i - p_i|$$

This returns a value in the $[0,1]$ where 0 indicates that the two distributions are equal, and 1 which indicates that the two distributions are completely different. Here, n is the categorical distribution produced by the LSTM working in the forwards direction given the prefix; likewise, p is the categorical distribution produced by the LSTM working in the backwards direction given the suffix. Notice that this measure ignores the value of the token under inspection.

To obtain a "local" measure of the naturalness of a given token, we consider both distributions as independent events. Given a token t in the erroneous file, whose vocabulary index is i, we calculate the joint probability of the adjacent token being t given the prefix, and the probability of the adjacent token t given the suffix.

$$\begin{aligned} local_t &= P(t|prefix \text{ and } t|suffix) \\ &= P(t|prefix)P(t|suffix) \\ &= next[prefix]_i \, prev[suffix]_i \\ &= n_i p_i \end{aligned}$$

This returns a value in $[0,1]$ where 0 indicates that the token witnessed is unlikely ("unnatural"), and 1 indicates that the token witnessed is very likely ("natural").

We then combine the two measures using an arithmetic average. Notice that total variation distance and joint probability have the same range, but opposite semantics. For this, we simply take the complement of one measure—total variation distance—and take the average. The average is in $[0,1]$, but the semantics follow that of probability, where 0 is unlikely, and 1 is certain. Finally, the naturalness for a single token in a source code file is given as follows:

$$score_t = \frac{1}{2}\left((1 - global_t) + local_t\right)$$

We calculate the $score_t$ of each token t in the erroneous file, and then sort tokens in the file in order of increasing naturalness. The first elements in the list are the least natural, and thus the most likely locations of the syntax error. Thus, we produce a *ranked list* of possible syntax error locations.

3.7 Fixing syntax errors

Given the top-k most likely syntax error locations (as calculated in Section 3.6), 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 most likely adjacent token 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 location, we obtain the most likely next token t_{next} (an entry of the vocabulary V) from the distribution returned by the forwards LSTM:

$$t_{next} = \text{argmax}_{i \in V} \, next[prefix]_i$$

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

```
1 if (name)) {
2   $form.addClass('highlight');
3   return;
4 }
```

Listing 3. Invalid JavaScript with one extraneous token

Likewise, we obtain the most likely previous token t_{prev} from the distribution returned by the backwards LSTM:

$$t_{prev} = \text{argmax}_{i \in V}\ prev[suffix]_i$$

Using t_{next} and t_{prev}, we try the following five edits, each time applying them to the incorrect file, and parsing the result with Esprima 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:

1. Assume the token at this location was erroneously **inserted**. Delete this token.
2. Assume a token at this location was erroneously **deleted**. Insert t_{next} at this location.
3. Assume a token at this location was erroneously **deleted**. Insert t_{prev} at this location.
4. Assume the token at this location was erroneously **substituted** for another token. Substitute it with t_{next}.
5. Assume the token at this location was erroneously **substituted** for another token. Substitute it with t_{prev}.

We repeat this process for the top-k most likely syntax error locations. k should be calibrated according to the expected value of finding the syntax error in the ranked list (discussed in the evaluation, Section 5.3).

Finally, all valid fixes are output to the user. These are suggestions, but every single fix is guaranteed to produce a syntactically-valid file. Ultimately, it is up to the programmer to determine whether a fix is appropriate.

4 PROTOTYPE IMPLEMENTATION

We created a prototype of *GrammarGuru* that implements the algorithms described in Sections 3.6 and 3.7. It is available on GitHub.[4]

Given the program from Listing 1, *GrammarGuru* produces the following output:

```
broken.js:4:38: try inserting '{'
    if (scorer.nextDoc() == NO_MORE_DOCS)
                                         ^
                                         {
```

Given the full program from Listing 3, *GrammarGuru* produces similar output:

```
insertion.js:5:14: try removing ')'
    if (name)) {
              ^
              )
```

The formatting of the output is inspired by that of Clang's "fix it" hints (Clang, 2016). It uses coloured command line output to emphasize the error. The concise wording of the message tells the user of the tool what they should try in order to fix the error, and the caret tells the user precisely which token should be removed, or where the token should be inserted.

[4] Available: https://github.com/eddieantonio/training-grammar-guru/blob/icsme2017/bin/detect.py

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

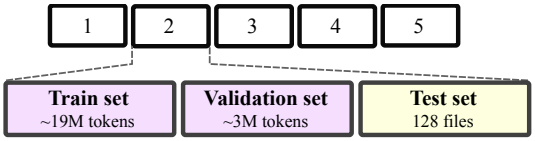

Figure 3. Each of the five partitions are further divided into three mutually-exclusive sets

5 EVALUATION

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

1. How well does *GrammarGuru* **find** syntax errors?
2. How often does *GrammarGuru* produce a **valid fix**?
3. If a fix is produced, how often is it the **true fix**?

A "true fix" is the edit that precisely reverses the mistake that introduced the error.

To answer these questions, we have adapted the evaluation methodology used by Campbell et al. (Campbell et al., 2014). We discuss how we partitioned the data collected in Section 3.1 into mutually-exclusive subsets in order to evaluate *GrammarGuru* (Section 5.1); we describe how we procedurally altered syntactically-correct JavaScript to synthesize a corpus of labelled syntax errors, called *mutants* (Section 5.2); finally, we describe the metrics we used to answer the evaluation questions (Sections 5.3, 5.4).

5.1 Partitioning the data

To empirically evaluate *GrammarGuru*, 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 three mutually-exclusive sets: the **train set**, the **validation set**, and the **test set**, thus resulting in 15 sets total (Figure 3). We populated every set with JavaScript source code from the corpus collected in Section 3.1. The training and validation sets were used in the training phase (Section 3.5), whereas the test sets were used in the mutation testing phase (Section 5.2).

When assigning source code files to sets, we imposed the following constraints to ensure the independence of training and test sets:

1. The source code files of a single repository cannot be distributed over multiple sets; in other words, every source code file in a given repository must be assigned to one and only one set.
2. Each corresponding set of each partition must be approximately the same size.

The first constraint is to make the evaluation more realistic, considering the expected use case of *GrammarGuru*. If a user is trying to figure out a syntax error in their own hand-written source code, it is unlikely that their source code belongs to a well-established open source repository that our models have already been trained on. Therefore, we do not mix files from other repositories between sets, to simulate the prediction of syntax errors in a file from a totally unknown source code repository.

We split our evaluation into five partitions to demonstrate how the performance of *GrammarGuru* changes when trained and evaluated on completely different data. Keeping each corresponding set 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 *GrammarGuru*.

	Mean	S.D.	Median	Min	Max
Train	19.23 M	814,607.66	18.89 M	18.76 M	20.68 M
Validation	3.05 M	715,868.73	2.76 M	2.51 M	4.24 M

Table 5. Number of tokens in partitions in the train and validation sets

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

5.2 Mutation testing

In order to evaluate *GrammarGuru*, we need a set of labelled syntax errors—JavaScript source files with a single syntax-error whose cause is known prior to evaluation. One option to collect invalid source code files is to collect them from mistakes made by real programmers; however collecting such a corpus is time-consuming, and the process of manually labelling the syntax errors is error-prone and potentially subjective. Instead, we adapted the evaluation methodology used by Campbell et al. (Campbell et al., 2014). Recall that all files that are retained in the corpus (and thus, all files assigned to the test sets) are syntactically-valid JavaScript files. We sampled the test sets of valid source code files to synthesize invalid files, called *mutants*, through a process called *mutation*. To mutate a source code file, we applied one of the following three edit operations:

Insertion A token is chosen randomly in the file.[5] A random token from the vocabulary[6] is inserted before this token.

Deletion A token is chosen randomly in the file. This token is removed from the file.

Substitution A token is chosen randomly in the file. This token is replaced with a random token from the vocabulary.[6]

For each file in the test set, we recorded at most 120 unique mutations for each of the three edit operations (insertion, deletion, and substitution), resulting in a total of 360 unique mutations per file. However, applying random edit operations to a file does not guarantee that the mutant will be syntactically incorrect. Consider inserting the unary negation operator (`!`) before a boolean expression, or substituting a variable name for a numeric literal. These examples show that a single random edit operation may still produce a syntactically-valid file. Thus, after a file is mutated we check its syntax using Esprima. We discard any mutants that are syntactically valid and try a different mutation instead.

5.3 Finding the syntax error

To quantify *GrammarGuru*'s accuracy in finding the error, we calculated the *mean reciprocal rank* (MRR) of the ranked syntax error location suggestions, by line number. Reciprocal rank is the inverse of the rank of the first correct line number found in an ordered list of syntax error locations for a mutant q. Mean reciprocal rank is the average of the reciprocal rank for every in mutant q in the set of total mutants 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 has the correct line number of the error. Conversely, an MRR of 0 means that the correct line number is never found in the ranked list of suggestions. For example, if for one mutant, the correct line was given first, for another mutant, the correct line was given third, and for yet another mutant the correct line was never found, the MRR would be $\frac{1}{3}\left(\frac{1}{1}+\frac{1}{3}+0\right) = 0.44$. MRR is considered quite conservative: in the case that the correct result is first half of the time and second the rest of the time, the MRR score is only 0.75.

5.4 Fixing the syntax error

To quantify *GrammarGuru*'s ability to fix syntax errors, we determined how often it finds the correct location and applies the correct fix. We report the success rate as a percentage, as well as the raw number of mutations for which *GrammarGuru* produced a valid fix. We report the success rate per each mutation kind (i.e., insertion, deletion, and substitution). We also report the mutation kind that *GrammarGuru* provided to fix the given syntax error. We expect that, if the mutation inserted an extraneous token into the file, then *GrammarGuru* would produce a deletion; if the mutation deleted a token from the file, we expect *GrammarGuru* to produce an insertion; similarly, if the mutation substituted a random token, then we would expect that *GrammarGuru* produces a fix that substitutes a token. In any case, if *GrammarGuru* produces an edit that, when applied, produces a syntactically-valid file, we report it as a *valid fix*. A stricter measure of success is how often *GrammarGuru* produces the *true fix*. That is, does the generated fix exactly undo the original mutation edit?

[5] The end of the file is also considered a "token" for the purposes of the insertion operation.

[6] Excluding the synthetic `/*<START>*/` and `/*<END>*/` tokens.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

6 RESULTS

6.1 Performance of finding syntax errors

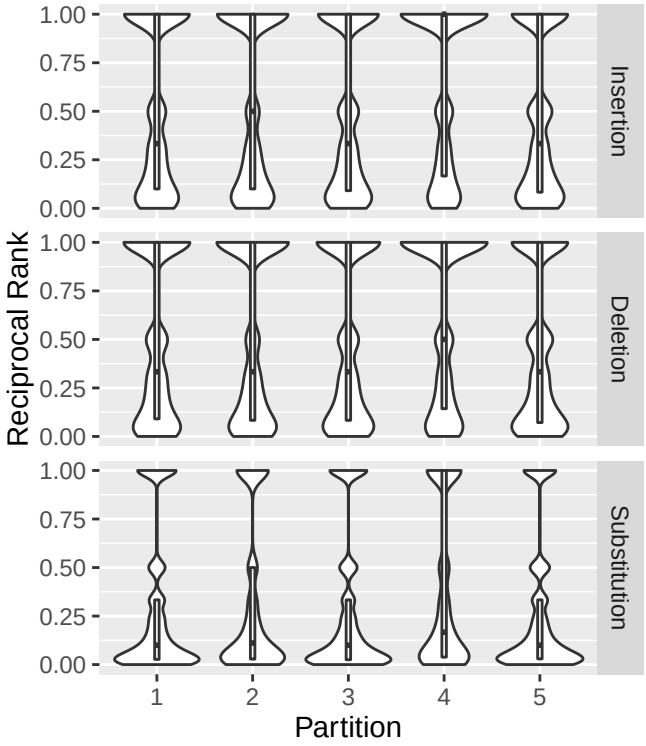

Figure 4. The mean reciprocal rank of determining the location of three kinds of mutation kinds, by partition

Table 6 lists the mean reciprocal rank obtained when finding the syntax error for each mutation kind and each partition. Both the kind of mutation and the partition have a significant effect on the reciprocal rank of finding syntax errors ($F = 4318.35$, $p \approx 0$ and $F = 545.41$, $p \approx 0$, respectively).

Partition	Insertion	Deletion	Substitution
1	0.51	0.49	0.28
2	0.54	0.52	0.32
3	0.48	0.47	0.28
4	0.62	0.60	0.38
5	0.46	0.44	0.26
Mean	0.52	0.50	0.30
Top-4	62.40%	61.01%	37.53%

Table 6. Mean reciprocal rank of mutation locations across each partition

Figure 4 is a series of violin plots with embedded interquartile ranges visualizing the reciprocal ranks when varying the mutation kind and partition. This gives a more granular view of the results in Table 6. The width of each "violin" displays the density of observations that have a reciprocal rank at that value. In all cases, there is a clustering of reciprocal ranks at 1.0, meaning that *GrammarGuru* can suggest the correct line number of a syntax error as its first suggestion 32.77% of the time. *GrammarGuru* can suggest the correct line number in its top-4 suggestions 54.74% of the time.

However, it is apparent that *GrammarGuru* especially struggles with finding substitutions. Notice that in all cases other than substitutions, the interquartile range of reciprocal ranks (inset in Figure 4) extends

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

	Fix created by *GrammarGuru*						Summary					
	Insertion		Deletion		Substitution		No fix		Valid fix		True fix	
Mutation	Total	%	Total	%	Total	%	Total	%	Total	%	Total	%
Insertion	499	1.04%	18790	39.21%	42	0.09%	28587	59.66%	19331	40.34%	17012	35.50%
Deletion	7795	16.90%	9939	21.55%	358	0.78%	28035	60.78%	18092	39.22%	7182	15.57%
Substitution	305	0.66%	1024	2.21%	475	1.03%	44473	96.10%	1804	3.90%	277	0.60%

Table 7. Percent correctness of finding and fixing syntax errors within mutated JavaScript files.

to 1.0. This is true for only one partition—partition 4—in the substitution case.

Since fixing the syntax error is directly dependent on *GrammarGuru*'s ability to find the error first, we should expect the results of fixing substitutions to suffer due to the relatively poor results in finding the substitutions in the first place. Despite this, the correct syntax error location is in the top-3 49.19% of the time, and in the top-4 53.74%.

Comparing the to prior work (Campbell et al., 2014), the n-gram model of order 10 outperforms *GrammarGuru* when it is trained on the same repository as it is being tested on, achieving an MRR of up to 0.99 on insertion, 0.88 for deletions, and 0.98 for substitutions. However, when evaluated against a different repository than its training data (as is the case in all of our tests), *GrammarGuru* yields better MRR for insertions and deletions. Compare the results in Table 6 to UnnaturalCode's MRRs of 0.36 on insertion, 0.20 on deletion, and 0.36 on substituion as shown in Table 2 on pp. 256 of Campbell et al. (Campbell et al., 2014). Similarly, *GrammarGuru* outperforms UnnaturalCode on insertions and deletions even when the new files come from the same repository as its training data: 0.47 on insertions, 0.29 on deletions, and 0.48 on substitutions. In all cases, *GrammarGuru* struggles with substitutions.

GrammarGuru outperforms Campbell et al. (Campbell et al., 2014) on insertions and deletions when finding syntax errors in unknown files.

As mentioned in Section 3.7, parameter k sets the amount of syntax error locations which are inspected to try to synthesize a fix. This value should be calibrated to the expected value of the rank that locates the true syntax error. The expected value of the rank is simply the reciprocal of the MRR. That is, $k = \lceil MRR^{-1} \rceil$. Plugging in the lowest MRR value obtained, 0.26, we get $k = 4$.

6.2 Performance of fixing syntax errors

Table 7 is a summary of how often *GrammarGuru* was able to fix syntax errors, for every given mutation. Each row of Table 7 corresponds to the mutation kind; the first three columns correspond to the valid fix that *GrammarGuru* generated to fix the given mutation. The cells that are highlighted show the *dual* of the mutation—that is, if a token was erroneously inserted, then the expected fix should be a deletion; if a token was erroneously substituted, then the expected fix should also be a substitution. The final three columns summarize the results, by how many times *GrammarGuru* was unable to produce a valid fix; how many times *GrammarGuru* did produce a valid fix; and how many of the total number of fixes were the *true* fix that exactly reverses the original mutation.

As expected, since the MRR of finding substitutions is limited, so too will be the results of fixing the substitutions. Curiously, *GrammarGuru* produces more fixes that *delete* a token rather than substitute a token to fix the substitution. In general, our tool seems to be biased towards deleting tokens rather than adding or substituting new tokens.

However we can evaluate the performance of *GrammarGuru* shown in Table 7 against a randomly selected fix as a baseline. First we select a random type of fix, either deletion, insertion or substition, and then we select a random token from the 98 tokens in the token vocabulary. Combining the baseline fix and the top-4 accuracy of *GrammarGuru* in finding the correct location, we calculate that the baseline would perform the true fix only 20.80% of the time for insertions, 0.13% of the time for substitutions, and 0.21% of the time for deletions. In all cases *GrammarGuru* outperforms the baseline for providing true fixes without taking into account fix location.

We investigated one of these cases, when a valid fix was produced by deleting a token. We investigated this snippet in Google Cloud's Node.JS client (Google Inc., 2016):

```
nextQuery = extend(/* ... */);
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

The mutation substitutes the variable `nextQuery` with the keyword `new`, making the statement syntactically-invalid:

```
new = extend(/* ... */);
```

GrammarGuru calculates the score of all 5151 tokens in the 2485 line file and determines that the token the = beside `new` has the lowest score at 8.14×10^{-5} in the file, making it the top syntax error location. By comparison, `new` has a score of 0.27, quite a lot higher (and thus more "natural") than that of the = immediately beside it. Hence, *GrammarGuru* deletes the unnatural = sign, turning what was originally an assignment to `nextQuery` into the invocation of a constructor, which is syntactically-valid—if dubious—JavaScript code.

```
new extend(/* ... */);
```

7 DISCUSSION

Our work is complementary to other syntax error detection tools. We do not intend to make a replacement for existing tools that work, such as Merr (Jeffery, 2003) and Clang's "fix it" hints (Clang, 2016). However, it can be used as one part of many to squash syntax errors.

The most pressing limitation is the speed of producing suggestions. There is a several second delay when initializing Keras (Chollet, 2015) and Theano (Theano Development Team, 2016). Using two models simultaneously is also computationally expensive. On one of our personal laptops (2.6 GHz Intel® Core™ i5–3230M, 8 GiB RAM, Intel HD Graphics 4000 1536 MiB), it takes around eight seconds for the tool to produce results for a relatively small file (46 tokens). The scalability issues are a real concern if this tool were to be used as a developer's companion. However, this concern can be mitigated by keeping the models "warm" and memoizing results. We employed both techniques when performing the evaluation: we initialized the models once per partition, and kept a least-recently-used (LRU) cache of prediction results. The LRU cache is beneficial when evaluating several consecutive iterations of the same file, such as when a developer is making incremental changes to one file. Each iteration of the file only affects a few tokens, leaving most tokens in the file untouched since the last iteration—hence, there is no need to compute predictions for these already-seen contexts (in other words, *memoizing* already computed predictions). We suggest that any practical implementation of our technique use a persistent process that initializes the models once, and maintains a prediction cache such that every subsequent run can benefit from the prediction cache.

In order to achieve acceptable accuracy in both detecting syntax error locations, and correcting them, one needs a large corpus with copious examples of all tokens in the vocabulary. However, this is not always the case. This problem is at its worst when the language definition evolves, as is the case for JavaScript. The latest versions of the JavaScript standard, ECMAScript 2015 and ECMAScript 2016, have introduced new syntax, and new keywords. However, these new features are not witnessed uniformly in the corpus; we have anecdotally observed that changing just one token to use newer syntax (such as substituting a `var` to a `let`) will cause the tool to fail to find the appropriate syntax error where it otherwise was able to produce a valid fix.

7.1 Threats to validity

Construct validity In Section 3.1, we obtained training data from the most popular repositories on GitHub. This code is more likely to be written by experienced programmers and use advanced idioms. As such, training on professional code to help fix novice code may be unrealistic.

The evaluation described in Section 5 chooses errors in a uniformly random manner, and as such, may not accurately represent the mistakes that novices actually make. For example, it is unlikely that a novice would substitute the `===` operator with the keyword `const`, yet our random evaluation created a mutant that did just that. In order to make a more realistic evaluation, we would need a corpus of errors made by actual programmers. An evaluation informed by a distribution of errors that novices actually make would be far more realistic and informative.

External validity is addressed by the use of a large number of JavaScript source files; however these source files where only collected from one source (GitHub), thus are not necessarily representative of all JavaScript code. External validity is also harmed by only addressing one programming language.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

8 FUTURE WORK

To resolve even more syntax errors, a deep learning model may be trained to both detect and suggest fixes for an arbitrary code location, rather than simply trained to predict the adjacent token as is in our case. There is also the possibility of using ensemble models that combine the strengths of deep learning, smoothed *n*-gram models, and other probabilistic models to achieve high precision in detecting and fixing syntax errors.

9 CONCLUSIONS

We described a novel method of detecting and correcting arbitrary syntax errors. Our tool is able to pinpoint the location of syntax errors where a standard LR parser would report a misleading error location. We describe how to train our model on a large corpus of open-source code, and how to use the raw text to train the model. We describe a two-LSTM system, and its syntax error detection strategy. We then describe our prototype, and evaluate it on synthetic syntax errors.

Our prototype *GrammarGuru* can successfully locate many syntax errors (54.74% in the top-4) and can repair many of them. The *GrammarGuru* models had difficulty with certain kinds of errors such as substitutions, but could address insertion errors with modest success.

We believe that *GrammarGuru* is a good step toward lowering the anxiety novices may face when dealing with syntax errors by providing find syntax errors suggesting fixes.

ACKNOWLEDGMENTS

The authors would like to thank Nvidia Corporation.

REFERENCES

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

Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J., Fan, L., Fougner, C., Han, T., Hannun, A. Y., Jun, B., LeGresley, P., Lin, L., Narang, S., Ng, A. Y., Ozair, S., Prenger, R., Raiman, J., Satheesh, S., Seetapun, D., Sengupta, S., Wang, Y., Wang, Z., Wang, C., Xiao, B., Yogatama, D., Zhan, J., and Zhu, Z. (2015). Deep speech 2: End-to-end speech recognition in english and mandarin. Preprint.

Barik, T., Lubick, K., Christie, S., and Murphy-Hill, E. (2014). 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.

Becker, B. A. (2015). An exploration of the effects of enhanced compiler error messages for computer programming novices. Master's thesis, Dublin Institute of Technology.

Bhatia, S. and Singh, R. (2016). Automated correction for syntax errors in programming assignments using recurrent neural networks.

Brown, N. C. and Altadmri, A. (2014). 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.

Campbell, J. C., Hindle, A., and Amaral, J. N. (2014). 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. Available: http://dl.acm.org/citation.cfm?doid=2597073.2597102.

Chollet, F. (2015). Keras. https://github.com/fchollet/keras.

Clang (2016). Clang—Expressive Diagnostics. Available: http://clang.llvm.org/diagnostics.html.

Czaplicki, E. (2015). Compiler errors for humans. http://elm-lang.org/blog/compiler-errors-for-humans.

Dam, H. K., Tran, T., and Pham, T. (2016). A deep language model for software code. Preprint.

Denny, P., Luxton-Reilly, A., and Tempero, E. (2012). 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.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

Dy, T. and Rodrigo, M. M. (2010). 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. ACM.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *35th International Conference on Software Engineering*, ICSE 2013, pages 422–431.

ECMA TC39 (2016). *ECMA-262: ECMAScript® 2016 Language Specification*. Ecma International, Geneva, Switzerland, 7th edition.

Garner, S., Haden, P., and Robins, A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education-Volume 42*, pages 173–180. Australian Computer Society, Inc.

Gibbs, A. L. and Su, F. E. (2002). On choosing and bounding probability metrics. *International statistical review*, 70(3):419–435.

Google Developers (2016). Chrome V8. https://developers.google.com/v8/. (Accessed on 02/23/2017).

Google Inc. (2016). index.js at 48a82a9fe1ae36139d910e7cb471da69d4fab546—GoogleCloudPlatform/google-cloud-node. https://github.com/GoogleCloudPlatform/google-cloud-node/blob/48a82a9fe1ae36139d910e7cb471da69d4fab546/packages/compute/src/index.js#L646-L650. (Accessed on 04/05/2017).

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

Gupta, R., Pal, S., Kanade, A., and Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. In *AAAI*, pages 1345–1351.

Hannun, A. Y., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., and Ng, A. Y. (2014). Deep Speech: Scaling up end-to-end speech recognition. Preprint.

Hidayat, A. (2016). Esprima. Avaiable: http://esprima.org/.

Hindle, A., Barr, E., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference On*, pages 837–847.

Hristova, M., Misra, A., Rutter, M., and Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153–156.

Jackson, J., Cobb, M. J., and Carver, C. (2005). Identifying Top Java Errors for Novice Programmers. In *Frontiers in Education Conference*, volume 35, page T4C. STIPES.

Jadud, M. C. (2005). A first look at novice compilation behaviour using bluej. *Computer Science Education*, 15(1):25–40.

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

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

Kummerfeld, S. K. and Kay, J. (2003). 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.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011). 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.

McIver, L. (2000). 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.

Mozilla Developer Network Contributors (2016). SpiderMonkey—Mozilla. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. (Accessed on 02/23/2017).

Mulder, F. (2016). Awesome error messages for Dotty. Available: http://scala-lang.org/blog/2016/10/14/dotty-errors.html.

Nienaltowski, M.-H., Pedroni, M., and Meyer, B. (2008). Compiler error messages: What can help novices? *SIGCSE Bull.*, 40(1):168–172.

Node.js Foundation (2017). Node.js. https://nodejs.org/en/. (Accessed on 07/04/2017).

Omar, C., Voysey, I., Hilton, M., Sunshine, J., Le Goues, C., Aldrich, J., and Hammer, M. A. (2017). Toward semantic foundations for program editors. Preprint.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017

Parr, T. and Fisher, K. (2011). LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA. ACM.

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

Raychev, V., Vechev, M., and Yahav, E. (2014). 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.

Rutter, T. (2013). What's the difference between jquery.js and jquery.min.js? Available: http://stackoverflow.com/a/3475058. Stack Overflow answer.

Tabanao, E. S., Rodrigo, M. M. T., and Jadud, M. C. (2008). Identifying at-risk novice Java programmers through the analysis of online protocols. In *Philippine Computing Science Congress*.

Tabanao, E. S., Rodrigo, M. M. T., and Jadud, M. C. (2011). Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*, ICER '11, pages 85–92, New York, NY, USA. ACM.

Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. Preprint.

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

Tieleman, T. and Hinton, G. (2014). RMSprop gradient optimization. Course slides.

Turner, J. (2016). Shape of errors to come—the Rust programming language blog. Available: https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html.

White, M., Vendome, C., Linares-Vasquez, M., and Poshyvanyk, D. (2015). Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3123v1 | CC BY 4.0 Open Access | rec: 3 Aug 2017, publ: 3 Aug 2017