Multi-token Code Suggestions using Statistical Language Models

Eddie Antonio Santos
easantos@ualberta.ca *

December 17, 2014

Abstract

I present an application of the *Naturalness* of software to provide multi-token code suggestions in GitHub's new-fangled *Atom* text editor [1]. After an error-fraught evaluation, there is not enough evidence to conclude that *Gamboge* significantly improves programmer productivity. I conclude by discussing several directions for research in code suggestion using naturalness.

1 Motivation

Code suggestion is a feature of text editors and integrated development environments that, when invoked, lists different *suggestions* for the code that might proceed the cursor position.

The amount of code in each suggestion may be one or more *tokens* long. A *token* is a group of characters that form one whole, indivisible unit of syntactic meaning in a programming language. This may also be known as a *lexeme*.

Code completion is code suggestion for one token that has already been partially typed-in.

Programmers generally use code completion and code suggestion to save keystrokes when typing code. For example, code completion may save keystrokes when typing long, descriptive, identifier names, or when typing "boilerplate" code—multi-token strings of code that appear frequently between projects that achieve a certain, common purpose.

An ideal code suggestion engine would read the mind of the programmer. Its suggestions would most often have to be exactly what the programmer needs. In [1], Hindle et al. found that code has "natural" properties. That is, code is "a natural product of human effort" [1], and is "likely to be repetitive and predictable". In fact, the cross-entropy of software against itself is lower than the

*Special thanks to J. C. Campbell and A. Wilson

[1] `https://atom.io/`

cross-entropy of English against itself, implying that the tools used in Natural Language Processing (henceforth, *NLP*) can be used successfully with software. This paper aims to exploit this fact, using an n-gram language model to generate statistically-informed *multi-token* code suggestions.

1.1 Prior Work

This concept isn't new. After finding that n-gram language models capture the regularity and repetitiveness in code, Hindle et al. [1] created an application of this property (henceforth *"naturalness"*) by writing an engine that suggests *one* token ahead of the cursor. Using a hybrid between the Eclipse suggestion engine and the n-gram language model suggestion engine, the authors were able to achieve up to a 61% improvement in keystrokes saved compared to the Eclipse suggestion engine alone. There have been several efforts since then that have tried improving the accuracy of this technique.

Nguyen et al. [2] augmented the n-gram model in [1] by attaching a wealth of semantic information to each token. The token suggestion accuracy was significantly improved over the base n-gram model. However, the extra information required, such as type information, syntactic role, the arity of methods, and topic analysis requires considerably more knowledge about the programming language and would be difficult to generalize across different programming languages—especially dynamic programming languages. Most importantly, it uses "naturalness" as merely one aspect of creating better code suggestions.

Tu et al. [3] sought to confirm that software is *localized*[2] Working on top of the fact that software is *natural*, they sought to find that there are "*local* regularities [in software] than be captured and exploited." They found, empirically, that this is the case. Their method combines a general n-gram language model trained on cross-project data with a *local* n-gram language model of higher order which was trained on a sliding window of about 5000 tokens around the cursor position. When they tried their hand at writing yet another code suggestion engine, they too found that it suggested the correct token more often than that in [1].

Other methods, such as that of Omar [4] and Raychev et al. [5] have also focused on techniques that require more information about the language such as its syntax, or its type system.

2 Implementation

In contrast to previous efforts, this paper attempts to predict *multiple* tokens in one go, using only the information that the statistical language model yields. This requires a small modification in how the n-gram language model from [1] is used. The n-gram model is relatively simple: it requires only knowing how to tokenize the input language. Thus, adapting it for a wide-range of languages should be relatively simple. Plus, it does not require static analysis that would

[2]Not to be confused with *Internationalization and localization*

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

be impossible in dynamic languages such as Python and JavaScript. Allamanis and Sutton [6] found that, with a much larger model trained on over one billion tokens, the n-gram language model performed "significantly better at the code suggestion task than previous models". This implies that all that is needed to generated better suggestions and to mitigate the problems of decreased accuracy due to predicting across project domains is to increase the corpus size. For this reason, the n-gram model was chosen, without modification.

2.1 What is an n-gram language model anyway?

An n-gram language model counts how many times a string of tokens has been seen in some training data, called its *corpus*. The n of n-gram denotes the *order* of the language model, or how many contiguous tokens are counted in the language model.

2.2 Prediction

Since the language model is just a big fat counter, we assume that the most probable token following a prefix of tokens (sometimes known as the *history* [7]) is the token that appears most often after the prefix in our language model. That is, we choose the token with the highest count out of all other tokens with the same prefix. Mathematically this is denoted as:

$$P(w_n|w_1 \cdots w_{n-1}) = \frac{C(w_1 \cdots w_n)}{C(w_1 \cdots w_{n-1})}$$

where $C(w_1 \cdots w_n)$ denotes the amount of times the token sequence $w_1 \cdots w_n$ has been seen in the corpus, and ${}_1 \cdots w_{n-1}$ is the given prefix.

For the sake of numerical stability (floating point arithmetic on values limited to 0.0–1.0 is prone to rounding errors), and simpler mathematics (calculating the compound probability of two independent events requires an addition instead of a multiplication), we apply the following transformation to the probability, denoted I:

$$I(w_n|w_1 \cdots w_{n-1}) = -\log\left(P(w_n|w_1 \cdots w_{n-1})\right)$$

This also give us an intuitive way at looking at the next token's probability. The most probable token following some sequence is also the least *surprising* token after the sequence. Indeed, we call this measure the *surprise*[3] of the token sequence. An event that is certain to happen, would have a probability of 100%. It is "not surprising", so its surprisal value is 0. Similarly, an event that could never happen would leave us "infinitely surprised".

[3]Sometimes known as *self-information*.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

2.3 Multi-token prediction

This approach works fine for predicting a single token. To extend this to predict multiple tokens, we simply apply this method recursively: do the prediction again, this time adding *the most recently predicted token* to the prefix. The suprisal of the entire multi-token suggestion is the sum of the surprisal of finding the last token following its respective prefix, for every token in the suggestion.

However, the manner by which predictions are ranked requires thought: Keeping the goal of *multi*-token code completion in mind, we want to balance two opposing goals: suggesting shorter sequences of tokens that are, by definition, more probable but offer no improvement to previous methods; with suggesting longer sequence of tokens that are, by definition *less* probable, but may just save the programmer a whackton of keystrokes.

2.4 Suggestion sort key

Suggestions should be ranked such that longer strings of tokens are preferred to subsets of the same suggestion, that are, more probable, but most likely save less keystrokes.

Let I_i be the *surprisal* of seeing tokens sequence t_i in the corpus. Each suggestion is given a score S_i which is given by the following formula:

$$S_i = \frac{I_i}{|t_i|}$$

where $|t_i|$ is the length of suggestion i.

This metric averages the surprisal of the entire suggestion over the entire token string. Henceforth, this metric is called the *mean surprise*[4]. In the trivial case when *all* suggestions in the set of suggestions are only one token long, this scoring function simplifies to I_i, the surprisal of that one token.

The suggestions are then presented to the user in ascending order of their score.

3 Methodology

Sixteen of the most popular Python projects from GitHub (based on stars) were collected. For each project, the language model was retrained on every file in the corpus *except* the files that belong to the project itself to simulate writing the project from scratch without the knowledge of the project itself. From each file, the final 350 tokens were typed using *Gamboge*.

Using Atom's spec helpers[5], files were typed programmatically, as if a person was writing the code. A programmatically typed file was considered to be "identical" if the result of tokenizing the original input (in terms of its text and category) was identical to the result of tokenizing the "typed" output.

[4] I swear this is not a move in Pokémon.

[5] The Jasmine Behaviour-Driven Development framework is one built-in to Atom, and has support for testing Atom headless.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

That is, given an original file looking like this:

```
w=lambda x: x*2
```

And the programmatically typed output:

```
w = lambda x : x * 2
```

The files are considered identical because in both cases, tokenizing their text produces the following token sting (presented as a JSON array):

```
[{"category": "NAME", "text": "w"}, {"category": "OP", "
    text": "="}, {"category": "NAME", "text": "lambda"}, {
    "category": "NAME", "text": "x"}, {"category": "OP", "
    text": ":"}, {"category": "NAME", "text": "x"}, {"
    category": "OP", "text": "*"}, {"category": "NUMBER",
    "text": "2"}, {"category": "NEWLINE", "text": "\n"}, {
    "category": "ENDMARKER", "text": ""}]
```

The difference in the two files is syntactically-*irrelevant* whitespace. Thus, in reporting keystroke count, typing inconsequential whitespace was ignored.

A suggestion was only completed if the cost of selecting the appropriate selection (in keystrokes) was less than simply typing out the token. For example, selecting the third suggestion in the list would require two keystrokes to move the selection down to the third element, and one more keystroke to select it, making three keystrokes total. A typer that only uses Atom's built-in editing features (such as automatic indentation) was also developed and tested. Its keystrokes per file were measured.

Additionally, for each prediction request, the following information was gathered:

- Were the desired tokens in the suggestion list
- If so, at what rank in the list?
- How long was the list of suggestions?
- How many keystrokes total did it take to choose the appropriate suggestion?

4 Results

Due to data generation being fraught with errors, there was not enough data generated to make a proper conclusion. Due to a bug in Atom, the tests would only run on small files. After working around this, only the final few tokens (under 300) were typed by the typers. This was marginally more succesful in generating data. However, in my rush to generate data in time, a mistake in the data collection code meant that accurate keystrokes counts for the control could not be collected. It counted both keystrokes for the prefix tokens and what was to be compared against the Gamboge typer. Despite the data being

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

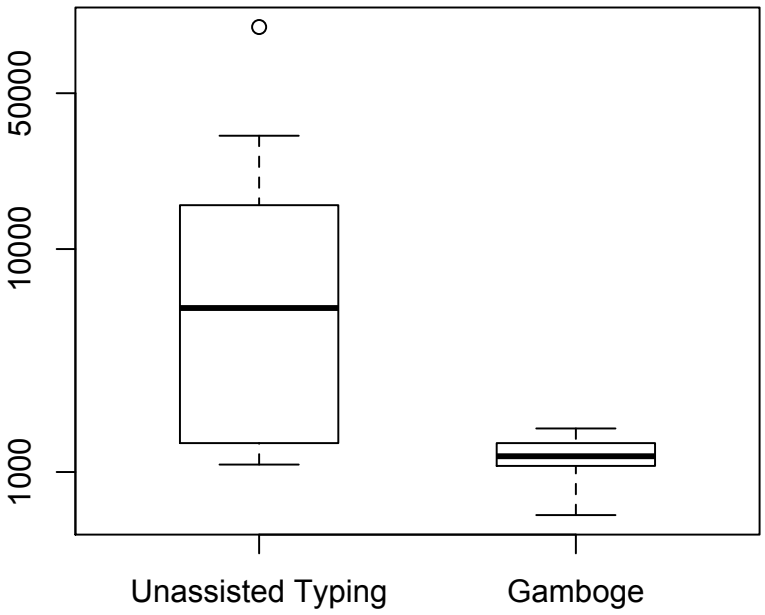

Figure 1: Amount of keystrokes per typer. Note that the Y scale is logarithmic.

erroneously biased in favour of Gamboge, a paired t-test showed no significant improvement in keystrokes for the files tested (95% significance level, p-value = 0.078). However, the assumptions for the t-test are completely invalidated, since both distributions have radically different variation, making the anaylsis invalid. Figure 1, shows the side-by-side box plots of the bad data.

5 Discussion

5.1 Retrospective

Originally, I had intended to use the tokenizer built into Atom that is used for its syntax highlighting feature. This would mean that any language Atom can highlight, *Gamboge* could generate predictions. This was based on the assumption that the tokenizer would be accurate enough to generate tokens

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

that could be passed to the language's grammar. Unfortunately, this is not the case. Its tokens would sometimes include multiple *language* tokens, or include unnecessary whitespace. As well, the syntactically-significant whitespace for languages like Python are completely absent from the token list. As such, these tokens were unusable for prediction, so I used the Python tokenizer provided by *UnnaturalCode* [8].

Atom itself is in early, rapid development. Since I pushed the first commit to `atom-gamboge`[6] in October 2014 to the time of this writing, Atom has been through https://github.com/atom/atom/releases, with *several* API-breaking changes. The documentation for Atom have been needing, at best. Since Atom packages are written in a dynamically-typed language (JavaScript or CoffeeScript), using the rapidly changing API is troublesome, because type errors often appear far away from where they originate. This means a lot of my time was spent in the debugger, reading Atom's actual *code*, and how other Atom packages are written. The test framework, although promising in that it is possible to fully script editor interactions, is also fraught with issues. Not everything is setup by default, and most of the editor must be manually put together from their component pieces. The other troublesome aspect during testing was attempting to interact synchronously over `stdin` with an external script, used for tokenizing Python (you know you're doing something wrong when it's easier to do it in C) Due to the intentionally asynchronous event-loop based design of Node.JS (on which Atom is built on), synchronous interaction with outside processes is cumbersome, and error-prone. A simple for-loops must be decomposed into awkward continuation passing style, with many, *many* callbacks. For this reason, Atom was invoked by an external Python program for each file that had to be typed.

Atom's documentation on testing is so underdeveloped, that I created a pull request to finish it. However, due to the needs of this project and other schoolwork, I did not complete all the work I wanted accomplish on documenting the magic monkey-patching that the Atom developer's did in test contexts.

What *is* nice about Atom is the fact that it's just a JavaScript app running in Chromium. This means that anything Chromium can do, I can exploit. What was especially simple was providing the ricey "ghost text" affect that was a trivial use of CSS 3 animations and opacity effects. Plus, Atom's choice of using dynamic LESS styles allowed me to use the color scheme of the user's chosen syntax theme to colour my spiffy ghost text effect. Atom Shell is a project worth looking out for. Another nice thing about this is that I already knew how to do an HTTP request—it's just Chromium—so `XMLHttpRequest` works out of the box. Plus, Atom is bundled with jQuery, so I could just use its more friendly AJAX API instead.

In addition, I took was very undisciplined in refactoring. Instead of having code that works, and passes tests that prove that they work, I had code that seemed to be working—but all in one massive god object. When it came time to separate the concerns, then and *only then* did I write tests. I wrote tests for

[6] `https://github.com/eddieantonio/atom-gamboge`

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

my *re factored design* before I had programmed it, but without verifying that the original worked. The result was that every class that I factored out of the original "view" (which turned into the controller) worked, and passed its tests (eventually). However, what's left of the original god object is tremendously buggy and weird, does not pass all its tests. It is currently pending a rewrite (since its functionality is comparably minimal to what it originally was).

I spent some time of this project working on existing code bases, such as the forked version of MITLM and UnnaturalCode. Particularly, I spent time figuring out how to extend the existing interprocess communication protocol between UnnaturalCode (in Python) and MITLM (in C++) to support both cross-entropy *and* prediction. Learning enough about both code bases took some of my time, and I balanced this time poorly while between writing code for the Atom package, and doing my literature review. However, the protocol was extended, and UnnaturalCode's existing tests covered any faults I might be introducing into either code base, with the exception of testing predictions. Much of the time spent on UnnaturalCode itself was spent writing tests for the new or modified functionality, rather than on writing the new code itself.

One thing that UnnaturalCode was missing was an "inline" mode for tokenizing Python code. As previously mentioned, Atom's tokenizer was ill-suited for my purposes, so I had to use Python's. UnnaturalCode has a forked version of Python's built-in tokenize module. However, after *every* input, it would emit one `DEDENT` token for every `INDENT` token left on the tokenizing stack. As well, the last token would always be an `ENDMARKER`. Thus, what my code would end up predicting on was often a bunch of `DEDENT` tokens followed by an `ENDMARKER`, which, naturally, did not often yield any good results. Getting rid of the extraneous `ENDMARKER` was good enough, but it was a better idea to modify the tokenizer with an "inline" mode that would simply stop before performing this finalizing step.

The manner by which *Gamboge* ultimately talked to the n-gram language model provided by MITLM went through the following stages: Atom would issue a `XMLHttpRequest` to *UnnaturalCode-HTTP*, which is a thin HTTP wrapper to UnnaturalCode. UnnaturalCode would handle tokenizing the input and passing it to MITLM via ZeroMQ. MITLM would do its MITLM thing, and pass back its predictions to UnnaturalCode, back to its HTTP front-end and finally to Atom. The only part missing in this strange bridge of processes was the HTTP interface to UnnaturalCode. I wrote a small Python library using the Flask web framework to accomplish this. Since it was intended to be small, and mostly just forward requests to UnnaturalCode, its interaction with the rest of UnnaturalCode is untested. After this project, however, it may be expanded and become a fully-tested component of UnnaturalCode, at the request of UnnaturalCode's author.

There were many, many workarounds. For example, I was running into `MemoryError` in my corpus downloading script. Since it only wrote files that were syntically valid in Python, it would have to call `compile` on each and every file downloaded. However, CPython implicitly keeps a cache of Python bytecode once `compile()` has been called. This cache is not kept alive by

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

references. In order to stop leaking memory, I changed my `syntax_ok()` function to `fork()`, and `compile()` in the child process, which would exit normally if the file compiled, or exit abnormally if the file failed to syntax check. The parent process would then return `True` or `False` based on the child's exit code.

Another issue that may have affect results is UnnaturalCode's treatment of tokens. Sometimes the text of tokens with the category of `NAME` come back from the langauge mode as `<NAME>` instead of the actual token text. String tokens were always returned with the text `<STRING>`. Similarly dedent tokens were returned with the text `DEDENT`, making them indistinguishable from an indentifier with the same text. The latter problem was fixed, but the former remained, and was used in the evaluation.

Fortunately, judicious use of source control saved my bacon several times. Since the latest commits the my development branch of my plugin were broken, I branched from the `v0.1.1` tag which I created for a demo-ready version of the plugin. The API changed between the release of `v0.1.1` and the time I needed it, so I just changed what needed to be changed in that branch and merged it into my evaluation branch. Without my use of tags, I would have been even more setback in tools to automate my evaluation.

5.2 Future Work

There are several ways to improve *Gamboge* in general. There are also several assumptions made in this paper that should be empirically verified.

5.2.1 Implementation concerns

The scoring method in this paper was chosen arbitrarily. It seems to work "well enough". Data is needed to confirm that this does work well enough compared to other possible scoring methods. That, or it may need to be theoretically proven in using a strong background in information theory.

An issue for the general adoption of *Gamboge* as the preferred text editor is whitespace. Currently, *Gamboge* naïvely separates tokens using a single space, as noted in the implementation section. This is not ideal, as not all tokens are separated using only a single space. Sometimes, as is often the case with the opening parenthesis of a function call, there is no space separating the identifier from the parenthesis (e.g., `foo()`). A simple possibility is to invoke the language's automatic program such as *autopep8* for Python, go fmt for Go. However, this method relies on the language having such a tool to begin with, plus, depending on the implementation, the cost of invoking an external program for every suggestion may be too slow for interactive use. A more ambitious method for formatting the tokens produced by the suggestion engine is to embed formatting information in the language model, in addition to the standard . This is based on the assumption that the code in the corpus largely abides by one language style guide. For languages that do not have one canonical automatic formatter, it would be unnecessary to write it; the language model would inherently know how to format code for this language, based on the

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

existing corpus. This proposal yields several research questions: Is this approach worth the effort? Is it too slow to invoking an external formatter for each suggestion in interactive use? To what extent do language communities abide by one universal coding style?

Despite its modest needs for knowing only how to tokenize the source code of a particular language, *Gamboge* is restricted to languages for which it knows how to tokenize, and create tokens that would ultimately be used in parsing the language's syntax. As noted previously, using a text editor's syntax highlighter's lexer for tokenization is troublesome, since its needs are only to *highlight* certain tokens; it is safe to completely disregard, for example, syntactically-significant whitespace that *Gamboge* needs in order to make decent predictions. The solution could be a parameterized lexer—since languages tend to have similar ways of lexing the broad categories of *identifiers*, *operators*, *delimiters*, and in some cases *syntactically-significant whitespace*, a general lexer might be possible. This would make the job of training a new language model and predicting from it quite easy: set the parameters required for lexing the new language and start training.

Currently, *Gamboge* only provides the preceding tokens, or *history* to its underlying language model for prediction. However, a possible improvement is sending both prefix and *postfix* tokens. The predictions would then have to match their surrounding context, rather than to be based off the tokens that precede it.

Previously, this paper made a distinction between code *completion* and code *suggestion*. *Gamboge* is a code *suggestion* engine, but its approach may be used to complete partially input tokens. Instead of showing what tokens may follow after the current token (always assuming that the token is complete), the engine may use the current identifier as a hint for filtering the suggestion list. For example, if the user types "`for key in d.`", *Gamboge* suggests complete tokens following "`.`". Once the user types "`i`", the suggestion list would be filtered and perhaps the most probable suggestion starting with "`i`" would be "`items ()`", which is listed.

5.2.2 Combination with other code suggestion engines

Gamboge is complementary to many existing code suggestion engines, and can be used as one part of a larger code suggestion effort.

The *Gamboge* prediction results can be used as a *suggestion provider* for the *AutoComplete+*[7] package. This is a popular plugin that makes improvements to the interface of the *AutoComplete* plugin such as automatically displaying suggestions (much like the ghost text feature of *Gamboge*). It also allows for other Atom packages to supply suggestions that will be displayed alongside its own.

An especially interesting addition to *Gamboge* is combining it with mined source code idioms or *snippets* described in the work of Allamanis and Sut-

[7] `https://atom.io/packages/autocomplete-plus`

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

ton [9]. The language model used in *Gamboge* could be modified to suggest where a snippet is likely to appear after a string of tokens in addition to its own suggestions. In this way, it can suggest both a string of linear tokens, *and* an entire snippet which, just like the results of the n-gram model, are mined from source code, rather than being arbitrarily hard-coded into the text editor.

Since Gamboge uses a simple n-gram model, extending its prediction backend with the cache language module developed by Tu et al. [3] may be beneficial. Since they have already shown that in single-token contexts, it surpasses a bare n-gram language model in suggestion performance, it may also improve suggestion performance for multi-token predicting.

A potential improvement is to compare all of the suggestions generated by the language model, and test each suggestion in its prediction context against a syntax checker. Then, it would eliminate any suggestions that are syntactically invalid. I hypothesise that due to the results of naturalness, there may not be a significant improvement to the mean reciprocal rank of the results. However, if P600, a processing delay when the mind encounters syntactic anomaly [10], holds for programming, removing all syntactically anomalous suggestions might eliminate any undue cognitive load.

5.2.3 Other work

Finally, *Gamboge* may be used in an effort to promote the usefulness of naturalness techniques. As mentioned in [1] and [3], naturalness and localness have several applications that are being lost on the crowd that are trying to make better code suggestion engines. However, in order to maintain large, and useful language models for the most popular languages, the storage and computation of such models may be used as part of an online service that, ostensibly, is intended only for providing robust code completion. This may entice programmers to contribute to naturalness efforts, while improving the experience of the many, *many* other applications of naturalness.

6 Conclusions

In its current state, *Gamboge* is not ready for widespread adoption by programmers. Reasons for this include its cumbersome dependencies on UnnaturalCode and MITLM, and the lack of a ready-made corpus for any language that a programmer would actually want to write code in. That said, I hope that this document enlightens anybody following the same path. Despite the challenges, putting more work into *Gamboge* may not only make it a useful suggestion engine, but establish it as a platform for others to exploit the virtues of the naturalness of software.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1597v1 | CC-BY 4.0 Open Access | rec: 19 Dec 2015, publ: 19 Dec 2015

References

[1] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software", in *Software Engineering (ICSE), 2012 34th International Conference on*, bibtex:Hindle2012, Jun. 2012, pp. 837–847. DOI: 10.1109/ICSE.2012.6227135.

[2] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code", in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, New York, NY, USA: ACM, 2013, pp. 532–542, ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491458. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491458 (visited on 09/19/2014).

[3] Z. Tu, Z. Su, and P. Devanbu, *On the localness of software*, 2014. [Online]. Available: http://zptu.net/papers/fse2014_localness.pdf.

[4] C. Omar, "Structured statistical syntax tree prediction", in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, ser. SPLASH '13, New York, NY, USA: ACM, 2013, pp. 113–114, ISBN: 978-1-4503-1995-9. DOI: 10.1145/2508075.2514876. [Online]. Available: http://doi.acm.org/10.1145/2508075.2514876 (visited on 09/19/2014).

[5] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models", in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, New York, NY, USA: ACM, 2014, pp. 419–428, ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594321. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321 (visited on 09/19/2014).

[6] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling", in *2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*, bibtex:Allamanis2013, May 2013, pp. 207–216. DOI: 10.1109/MSR.2013.6624029.

[7] C. Manning and H. Schuetze, *Foundations of Statistical Natural Language Processing*, 1 edition. Cambridge, Mass: The MIT Press, May 28, 1999, 720 pp., ISBN: 9780262133609.

[8] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models", in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM Press, 2014, pp. 252–261, ISBN: 9781450328630. DOI: 10.1145/2597073.2597102. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2597073.2597102 (visited on 09/13/2014).

[9] M. Allamanis and C. Sutton, "Mining idioms from source code", *arXiv:1404.0417 [cs]*, Apr. 1, 2014. arXiv:1404.0417. [Online]. Available: http://arxiv.org/abs/1404.0417 (visited on 09/19/2014).

[10] P. Hagoort, "How the brain solves the binding problem for language: a neurocomputational model of syntactic processing", *NeuroImage*, vol. 20, Supplement 1, no. 0, S18 –S29, 2003, Convergence and Divergence of Lesion Studies and Functional Imaging of Cognition bibtex: Hagoort2003S18, ISSN: 1053-8119. DOI: http://dx.doi.org/10.1016/j.neuroimage.2003.09.013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1053811903005251.