Editor: **Tim Menzies**
North Carolina State University
tim@menzies.us

# Naturally!: How Breakthroughs in Natural Language Processing Can Dramatically Help Developers

Anand Ashok Sawant and Premkumar Devanbu

**From the Editor**

Do you have exciting stories about novel artificial intelligence (AI) methods for software engineering (SE)? Why not submit them to the "SE for AI" column? Articles should be 1,000–2,400 words and be practitioner focused (figures and tables count as 250 words; try to include 12 references or fewer). Submit initial ideas or finished articles to timm@ieee.org. In this issue's column, Anand Ashok Sawant and Premkumar Devanbu show us that source code is like natural language, i.e., it is highly repetitive and predictable in nature. This enables a whole range of exciting and novel new analysis methods for SE.—*Tim Menzies*

**TAKING ADVANTAGE OF** the naturalness hypothesis for code, recent development, and research has focused on applying machine learning (ML) techniques originally developed for natural language processing (NLP) to drive a new wave of tools and applications aimed specifically for software engineering (SE) tasks. This drive to apply ML and deep learning (DL) has been animated by the large-scale availability of software development data (e.g., source code, code comments, code review comments, commit data, and so on) available from open source platforms such as GitHub and Bitbucket. Commercially available tools that leverage this combination of artificial intelligence (AI) and SE include:

- Codota.ai [Figure 1(a)] and Kite: tools that use DL and NLP techniques to do better in IDE code completion
- deepcode.ai [Figure 1(b)]: performs code analysis based on learned rules to augment current static analysis tools

*Digital Object Identifier 10.1109/MS.2021.3086338*
*Date of current version: 20 August 2021*



0740-7459/21©2021IEEE

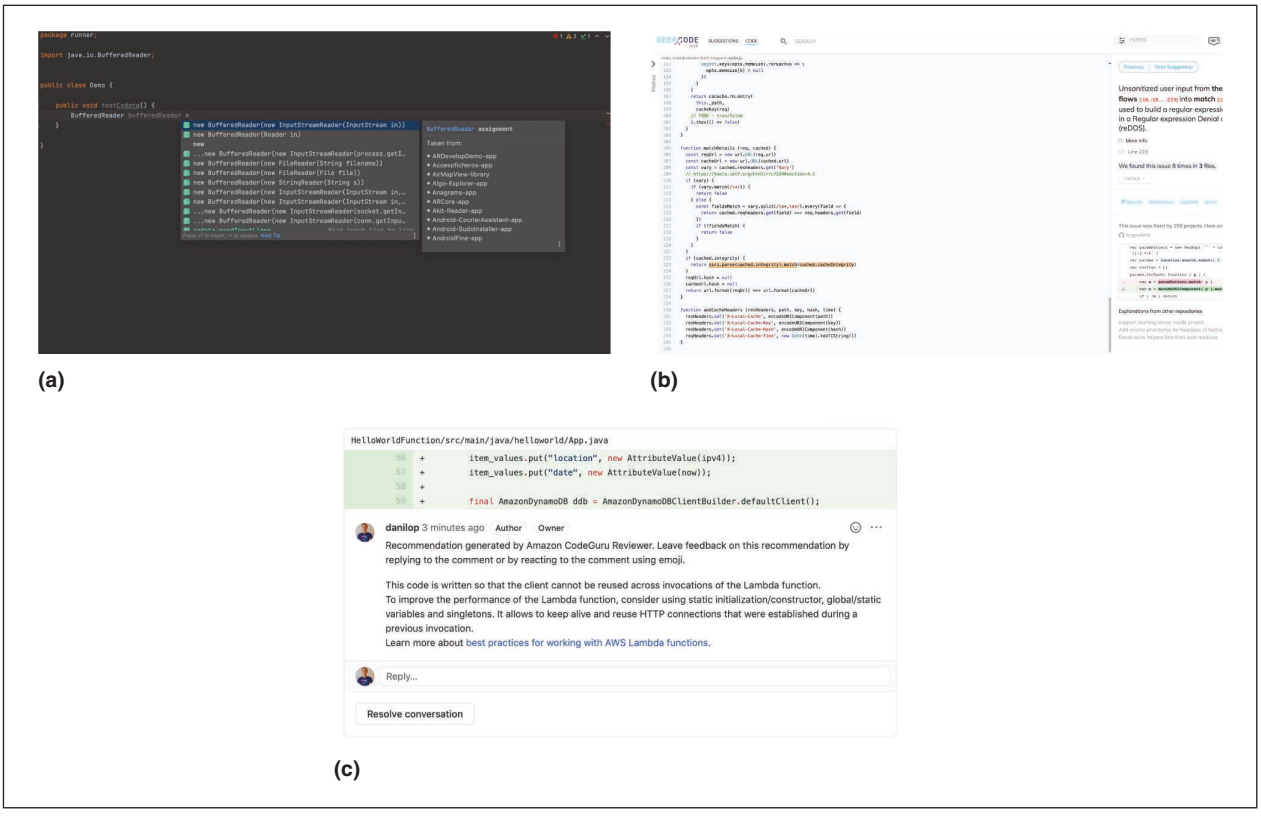


**FIGURE 1.** Commercial tools taking advantage of the naturalness hypothesis. (a) PROBLEM: Code completion at line level that includes all variables is hard. SOLUTION: Using naturalness, we can learn complete code patterns from other projects to suggest relevant code completions. (b) PROBLEM: Finding potential issues such as duplicate code or security issues using static analysis can result in a lot of false positives or negatives. SOLUTION: Using naturalness, we can learn potential issues from token relationships from past issues to identify issues in previously unseen code. (c) PROBLEM: Manually reviewing code written by others to find bugs/issues is not completely reliable. SOLUTION: Using naturalness, we can learn a mapping of potential code issues to a natural language description to generate code review comments.

- CodeGuru [Figure 1(c)]: developed by Amazon to perform automated code reviews on customer code that use Amazon's own Amazon Web Services APIs.

## Why Does This Work Well? Parallels Between AI and SE and NLP Tasks

The naturalness hypothesis for code states that "though software, in theory, can be very complex, in practice, it appears that even a fairly simple statistical model can capture a surprising amount of regularity in 'natural' software." Source code, like natural language, is a form of communication between programmers. Much like natural language, more "natural" code—code that has lower per-token entropy—has a direct impact on its readability (subsequently maintainability). Source code, like natural language, is predictable—in fact, it might be even more predictable due to the repetitiveness of code.

Source code is often much more repetitive than texts written in languages such as English (assuming no plagiarism). Studies have shown that code tokens are easier to predict than normal text.[1] This is due not only to the simpler syntax of code[1] but also to an adherence to similar programming styles across projects. This would indicate that code is often written in such a way that another human can understand it (and not just for execution by a machine). In terms of modeling code, this can mean that code language models might be just as, or even more, effective than their equivalent for NLP.

A growing body of evidence[1–3] supporting the naturalness hypothesis indicates that language models that have been commonly used in the NLP world applicable to SE problems as well. It's not just the models that transfer over to the SE world, but there are also similar tasks that are applicable, for example, text summarization in NLP that parallels code comment generation. We see those language models for source code are used in much the same ways as those in NLP. A partial list of SE tasks that take advantage of the naturalness hypothesis include:

1. code comment generation
2. code retrieval
3. code deobfuscation
4. defect prediction
5. API mapping
6. code translation
7. automated code review
8. code completion
9. type recovery.

To achieve any one of these tasks, SE research has to take advantage of NLP models, some examples of these are as follows:

1. *Code2Vec*: these approaches utilize a bag-of-words approach similar to Word2Vec. Essentially, the vector representation for each token is calculated based on the surrounding tokens. Code2Vec[4] learns a vector representation of a piece of code based on paths in its abstract syntax tree. Figure 2 depicts an example where a Code-2Vec based model can label the semantics of a piece of code, that is, predict the expected name of a function.
2. *RNN-based language models*: recurrent neural networks (RNNs) can be used to capture local sequence dependence between tokens. They have been shown to be effective for text prediction in the NLP world. In the SE world, modeling the source code using RNNs has been shown to be effective for code completion-based tasks. One such example is by Raychev et al., where they developed a model and tool called SLANG. This tool can predict based on what a developer has (partially) typed to guess what the complete form should be (see the example in Figure 3).
3. *Long short-term memory (LSTM)-based language models*: simple RNNs have one major shortcoming where the context for which it learns an embedding of a token (i.e., the length of the preceding tokens that contribute) is limited. LSTMs, on the other hand, can store token information over long-term temporal dependencies. This has made LSTM's effective for NLP classification problems. Similarly, LSTMs are especially useful in the case of source code as variable usage and declaration might often be separated by a lot of tokens. One effective use case in the SE world of an LSTM is a tool called NL2Type[5] (seen in Figure 4) to predict a JavaScript function parameter's type based on the function body.
4. *Graph neural network approaches*: all of the aforementioned approaches only consider the lexical nature of source code, that is, the models are based on

```
Object f (int target) {
    for (Object elem : this.elements) {
        if (elem.hashCode().equals(target)) {
            return elem;
        }
    }
    return this.defaultValue();
}
```

Function f is get

**FIGURE 2.** Code2Vec used to identify semantics of a function definition.

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
        ArrayList msgList = smsMgr.divideMsg(message);
    ?   ⟹  smsMgr.sendMultipartTextMessage(...msgList...);
} else {
    ?   ⟹  smsMgr.sendTextMessage(...message...);
}
```

**FIGURE 3.** RNN-based model used to complete snippets of code.

token sequences. However, source code has semantic properties which can be represented as a graph (or tree). Graphical representations of code semantics can be modeled using a graph neural network approach. One task for which it has been shown to be effective is the variable misuse task, where the model by Allamanis et al.[6] learns from the relationships in the program graph as to what variable needs to be used where and where it should not be used (see Figure 5), that is, it learns some information about the semantic flow of the source code.

5. *Transformer-based approaches*: the current state of the art in NLP is 1) pretraining a larger transformer-based language model using billions of tokens such that it captures the underlying statistical relationships between all the token in the corpus and 2) fine-tuning this pretrained model for a specific task such as sequence labeling. Adoption of this approach has recently been on the uptick, where we see pretrained models, such as CodeBert,[7] CUBert,[8] and PLBART,[9] which pretrain models using billions of token of source code. These models are then applied to a variety of tasks such as variable misuse detection, identifying the wrong binary type, and detection of the exact exception type.

## What Are the Differences, Though?

Most of the AI + SE work parallels that of the NLP world. This operates under the assumption that code is the same as any natural language. However, there are a few differences between source code and natural language and that has certain implications.

1. *Vocabulary growth*: the vocabulary used in source code is much larger than that of natural language. Developers can freely create identifier names; this means that even a model trained on billions of tokens might encounter novel vocabulary. This can severely hinder the performance of a model when it comes to generalizing performance to more than a small set of projects (out-of-vocabulary problem). One way to overcome this obstacle is using byte pair encoding where each token is split into subtokens (e.g., <LinkedList > → <Linked> <List >) and embeddings are derived for these subtokens. This ensures that even for unseen tokens, at least part of the token might be part of the vocabulary. Since the subtoken vocabulary includes the character set, even dinovo tokens could be theoretically handled. Byte pair encoding works very well in practice for code.[10]
2. *The "span" or "scope" of relevant text is much wider for code*: language models learn some relationships between tokens that are in their immediate proximity. While this works well for natural language, in the case of source code this can be especially tricky. For example, in a class, a field can be declared at the top of the file and the field is then used (read or written to) later on in one of the methods that might be at the bottom of the file and 1,000 tokens away. This seldom occurs in NLP, where the subject and object of a piece of text can mostly be found in a smaller context window. Even proper nouns tend to be reused in fairly proximate contexts. In code, learning this relationship between the declaration of the field and its usage can be limited by current hardware constraints based on the size of sequences and batches that can be fit into a GPU. Typically, on an Nvidia V100 GPU,

```
/** Utility function to ensure that object properties are
    * copied by value, and not by reference
    * @private
    * @param {Object} target Target object to copy
    *                        properties into
    * @param {Object} source Source object for the
    *                        properties to copy
    * @param {string} propertyObj Object containing
    *                        properties names we
    * want to loop over
    */
function deepCopyProperties(target, source, propertyObj) {
    ...
}
```

Parameter Should Be of Type Object

**FIGURE 4.** The LSTM model used to detect the type of JavaScript parameters.

```
public ArraySegment<byte> ReadBytes(int length) {
    int size = Math.Min(length, _len - _pos);
    var buffer = EnsureTempBuffer(length);
    var used = Read(buffer, 0, size);
```

Incorrect variable used, should be size

**FIGURE 5.** A graph neural network is used to detect incorrect variable usage.

## ABOUT THE AUTHORS

**ANAND ASHOK SAWANT** is a postdoctoral scholar at the University of California, Davis, Davis, California, 95616, USA. Further information about him can be found at https://anandsaw.github.io/. Contact him at asawant@ucdavis.edu.

**PREMKUMAR DEVANBU** is a distinguished professor in the Department of Computer Science at the University of California, Davis, Davis, California, 95616, USA. Further information about him can be found at https://web.cs.ucdavis.edu/~devanbu/. Contact him at ptdevanbu@ucdavis.edu.

the maximum sequence size is 512 tokens and 16 batches, thereby reducing the chances that the field and its usage will be in the same sequence. To alleviate the issues caused by this, researchers have often resorted to only using input sequences that might fit on the GPU with all relationships preserved, thus discarding any data that might be too long. This does mean that most AI and SE models are restricted by the size (in terms of the number of tokens) of the code and do not effectively scale to larger programs.

3. *Code is bimodal*,[11] *admitting both formal and "natural" reading*: during this entire article we have operated under the assumption that source code is "natural." One might assume that reading code is the same as reading a paragraph from Harry Potter, especially since most code appears to have been written to communicate with other humans in addition to the computer; clearly, it is not. This has led to a dual-channel hypothesis for code. One channel is formal or algorithmic, where the execution of the code is specified. The other channel is a natural language channel, where the code is a medium of communication with another human developer. A functional magnetic resonance imaging (fMRI) study[12] has confirmed this to an extent where neuroscientists have found that when reading code, humans engage both the parts engaged with mathematics and natural language. All language models used in AI and SE take advantage of this natural channel; static analysis tools and compiler optimizations exploit the algorithmic channel. These "bimodal" property enables several new directions of research, for example, static analysis methods could selectively perform approximations when a language model indicates it is unlikely to lead to inaccuracies; algorithmic approaches could be used as training signals to learn noise-resistant representations that could be used to fix errors;[13] recovering original code from algorithmically obfuscated code could be used as a training signal to train autoencoders.[14]

## Going Forward

This research field of AI and SE has taken off as an increasing number of papers at top SE conferences utilizing state-of-the-art AI approaches. This does appear to reinforce this notion that treating code as natural language does work. But what does the future hold? As mentioned earlier, code is a bit more than just a series of tokens there is richer information available in code (the algorithmic channel). Designing models and approaches that take this into account along with the natural channel is the next frontier for the field of AI and SE. We look forward to the new waves of research that are sure to follow and welcome the many new researchers to this field!

## References

1. W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021, arXiv:2103.06333.
2. T. Ahmed, P. Devanbu, and V. J. Hellendoorn, "Learning lenient parsing & typing via indirect supervision," *Empirical Softw. Eng.*, vol. 26, no. 2, pp. 1–31, 2021. doi: 10.1007/s10664-021-09942-y.
3. M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017, arXiv:1711.00740.
4. U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Programming*

*Languages*, vol. 3, no. POPL, pp. 1–29, 2019. doi: 10.1145/3290353.
5. C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., New Ideas Emerg. Results*, 2020, pp. 25–28.
6. C. Casalnuovo, K. Lee, H. Wang, P. Devanbu, and E. Morgan, "Do programmers prefer predictable expressions in code?" *Cogn. Sci.*, vol. 44, no. 12, p. e12921, 2020. doi: 10.1111/cogs.12921.
7. C. Casalnuovo, E. Morgan, and P. Devanbu, "Does surprisal predict code comprehension difficulty," in *Proc. 42nd Annu. Meeting Cogn. Sci. Soc.*, Toronto, Canada, 2020, pp. 564–570.
8. C. Casalnuovo, K. Sagae, and P. Devanbu, "Studying the difference between natural and programming language corpora," *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 1823–1868, 2019. doi: 10.1007/s10664-018-9669-7.
9. Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages," 2020, arXiv:2002.08155.
10. A. A. Ivanova et al., "Comprehension of computer code relies primarily on domain-general executive brain regions," *Elife*, vol. 9, p. e58906, 2020. doi: 10.7554/eLife.58906.
11. A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. Int. Conf. Machine Learning*, 2020, pp. 5110–5121.
12. R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Open-vocabulary models for source code," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., Companion Proc.*, 2020, pp. 294–295.
13. R. S. Malik, J. Patra, and M. Pradel, "Nl-2type: Inferring javascript function types from natural language information," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 304–315. doi: 10.1109/ICSE.2019.00045.
14. B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," 2021, arXiv:2102.07492.



*Digital Object Identifier 10.1109/MS.2021.3099682*