CS846 Week 3 Reviews

Felix Wang

Problem Being Solved

The most central conjecture of this paper is that "Most software is also natural (like other natural languages)", in the sense that it contains attendant constraints and limitations of human work and thus is repetitive and predictable. The author of this paper wants to use a statistical language model called n-gram to model the coding languages in Java, C, and Python and compare it with the results trained on natural language corpora like Gutenberg and Brown. They also want to utilize the n-gram model to build an Eclipse Plug-in suggestion engine and evaluate whether it's useful to developers.

New Idea

The author wants to use a statistical language model to prove the repetitiveness and predictability of code, and because the repetitiveness and predictability appear on the lexical level, syntactic level, and semantic level, statistical language models are capable of doing so.

The author used Brown and Gutenberg corpora for natural languages, and Java projects, Ubuntu applications in C, after removing all human comments for the code, and then trained them by 10-fold cross-validation training, which randomly selected 90% of the data for training and 10% of the data for validation. One drawback of such a model is the sparsity, which the author used a fall-back to the (n-1)-gram technique if the probability of an n-gram is 0, and used a smoothing method. The metric used by this paper is the cross-entropy loss, which measures how confident the language model is able to guessing the next word.

By training such an n-gram model on Java projects, they found that Java contains a lot of local repetitiveness, and software is far more regular than English. And by training a trigram model on 10 different Java projects, respectively, they found that each project has its own type of local and non-Java-specific regularity. And by training the n-gram model on Ubuntu applications across 10 domains, they found that a lot of local regularity is repeated within application domains, and much less so across domains.

They designed a heuristic algorithm called MSE, which combined their n-gram model suggestion (NGSE) engine with Eclipse's built-in suggestion engine(ECSE). If ECSE has long tokens within the top n, then take ECSE's top n offers, or if not, then half NGSE, half ECSE. And in practice, this is proven to save more keystrokes than ECSE.

Positive Points

One positive point about this paper is that I like the choice of the n-gram model. It's deterministic and explainable, but many language models based on optimization algorithms are not. Since the scope of this paper is to prove that software is repetitive and explainable, it's good to choose something deterministic and explainable.

The second thing is that this paper has very strong empirical evidence. It used some large and real-world datasets which contain 10 major Java projects, 10 Ubuntu domains of C applications, and 2 large-scale natural language corpora (Brown: ~1 million words, Gutenberg: ~2.5 million words), and used a rigorous procedure of 10-fold cross-validation, which provides robust and strong empirical evidence.

Negative Points

One thing I don't like is that though repetition does mean regularity, it's not always a good sign in software engineering. The paper lacks preprocessing on the codebase that ensures the quality of code, including whether it contains legacy code, whether it contains code that follows poor coding practices that cause redundancy.

The MSE algorithm is too superficial. We already have ECSE, which is based on type information, and NGSE, which has statistical patterns, we are able to build something better on top of them, not simply choose one or another.

The next thing is the limitations of N-gram. Although the n-gram model is able to catch some local grammatical structures, it's unable to catch any long grammatical structures, and cannot give any long suggestions.

Also, the author illustrated "most software is also natural", and inferred that "it's also likely to be repetitive and predictable" from there. But this is not a sufficient and necessary condition, we are not able to infer backwards from "proving that software is repetitive and predictable" to "proving that software is natural".

Future Work

A Probabilistic Context Free Grammar (CFG) is a concept related to traditional NLP, aiming to capture the grammatical structure of natural languages and construct an abstract syntax tree of what grammar is allowed in the training data, along with the training process, and label the probability of each possibility. The n-gram model can work with PCFG in the field of traditional NLP, and what I'm proposing is to incorporate these two together and let the n-gram model be capable of catching complicated grammar in code, and repeat the procedure of this paper again to see if the model catches something new.

Rating

3.5/5. I think it's a good paper, but not life-changing. The idea of naturalness of this paper is novel, and the work being done is rigorous, but the reliance on the n-gram model limits its ability to catch something more in-depth.

Discussion Points

1. With the transformer architecture coming out, everything about language models, computer vision, speech recognition, and machine translation changes fundamentally. For the future directions section (section 6), which direction do you think becomes irrelevant, and which direction do you think becomes more relevant and realistic?

One student that this paper talks about deep learning very briefly and doesn't really capture how powerful language models were going to be in the future, and he believes that this paper underestimated the strength of corpus-based learning.
Prof. Godfrey talked about how this paper was the first one that used a language-model-based technique of that kind, and it worked. The n-gram was simple and easy to prove, and the data showed that it's possible. This technique was simple and straightforward, and this idea is solid for its time. It proved the possibility of using AI to do software tasks.
For me, I think direction 6.1 is still solid, the trade-off between model capacity and sparsity is still an issue even now. Even advanced models like GPT-5 need to do a lot of downstream fine-tuning for sparsity issues. 6.1-6.3 is not relevant anymore with the advancement of deep learning. These are basically "solved" or "nearly solved" problems by using deep learning, and have no point in using the Naïve Bayes proposed in this paper. 6.4 is the current trend, although it went way beyond what Dr. Hindle could have imagined back then. This paper provided a perfect theoretical foundation for the emergence of later models.

2. The N-gram-based Eclipse plug-in showed 27–61% improvement in code suggestion. And the measure used is the number of keystroke savings. Do you think this result convincingly demonstrated practical benefits, or is it more of a proof-of-concept? What makes you think it indicates practical benefits? Or what follow-up study could be done to make it more convincing?

One student mentioned that she believes that, over time, even if they are small, productive jumps, they could show improvements more significantly in productivity.
Another student talked about how he would have liked to see optimized code versus just code completion coming from the n-gram. This would be more helpful than simply code suggestions.
My take on this is that the measure of keystrokes, though not convincing enough, is definitely better than a proof-of-concept. But a follow-up case study with a controlled experiment on the result of work by professional developers using this tool, or without using this tool, is more convincing.

3. Should "Quality of Code" play into our proof for naturalness? If yes, how can we incorporate it? If no, please justify it.

One student argues that the regularity in the project was more obvious, even if it's not the "best" code. The n-gram picking up on that would be a good thing, potentially, regardless of code quality. Another student believes code consistency is far more important than other quality measures.
The professor explained how the industry standards help to standardize the way developers write code. He suggested that in the project, you would expect there to be even more regularity.
My opinion is that we don't necessarily need to let the quality of code be a metric when proving the naturalness of software. But if we are actually applying it in practice, it's better to ensure the model is learned on training data with good coding practices.

4. If we now have Eclipse's built-in suggestion engine and N-gram model suggestion engine (NGSE) both in hand, how could we build something potentially better than MSE?

Prof. Godfrey talked about throwing out special cases and just going hard with statistical analytics. He talked about how the data is really what can drive better performance.
A student mentioned that the linguistics community still don't have a definition of what a word is.
My personal opinion on this is that I think first we could combine the PCFG context tree that I introduced in the future works section, and combine it with some search engine or recommendation engine kind of thing built with deep learning algorithms on top of these two.

5. Any disagreement or agreement with my critiques before? Do you have other positives or negatives that you want to share?

One student talked about his confusion with token length and questioned whether the length of a token would affect its probability in the n-gram model. Another student shared his explanation of what he thought the tokenization technique could entail, but I don't have enough expertise to keep this conversation going further.