# 6.864 Proposal

Matthew Kilgore

2015-12-11

## 1 Overview

We have begun to examine and continue the work in [1], "Code completion with statistical language models." This paper, as its title implies, examines the following question:

> *Given a program with some designated areas removed, how well can a natural language model predict the removed code?*

They solve this by converting code into an execution, and then applying a trigram or RNN model, which is used to score "sentences" generated by a bigram model.

### 1.1 Shortcomings

While the above model works, it has some shortcomings. The model suffers from issues with loops.

- The model is unable to handle arbitrary loops, instead requiring a bound on the number of iterations. This eliminates many possible programs.
- Moreover, the method of constructing executions counts every occurrence of a function call in a loop once for every iteration. In the general case, this could skew results towards occurrences common in loops.
- Finally, the implementation in [1] sets the loop upper bound to two iterations. While this does somewhat handle the concern of the previous point, it again is incredibly restrictive.

The Natural Language Model similarly suffers from defects:

- The tested models are standard $n$-gram (specifically trigram) and RNN models. While they demonstrate that this is effective, it is not ideal for programs. More specifically, it does not ideally capture non-linear control flows. While fully using such features would defeat the purpose of using Natural Language Methods, ignoring them entirely is counterproductive.

- The method of generating and scoring candidate completions is inconsistent. Namely, while sentences were scored with trigram and RNN models, completions were generated with a bigram model. While this is not necessarily an issue, it would seem as if a more consistent method may yield better results.

Finally, the accuracy of [1]'s model is reported in an unusual manner. Specifically, the paper lists the location of the accurate code completion within the ordered list of suggested completions—for example, "appearing in the top 3." However, the actual assigned probability is not given. This removes a potentially crucial piece of data.

# 2 Areas We Explored

Our changes focused on two areas:

## 2.1 Support of Alternative Control Flows

We changed the method by which training sentences were generated—whereas before they were generated by a complex analysis of the supplied programs, they are now generated directly from a parse tree:

- Programs were represented as trees of "Code Units." Intuitively, these are a simplified parse tree, and represent a block of the code, such as a method declaration.
- Each code unit has a name, corresponding to the nonterminal in the parse tree that generated the code unit.
- Each code unit has a body consisting of statements and other, nested, code units.

This allowed two generation schemes:

- `levels`. Every code unit's body (excluding nested code units) was transformed into a sentence.
- `cfs`. The code units were transformed into a rough, acyclic version of the control-flow graph; the various potential paths were made into the sentences.

## 2.2 Alternative Completion Generation Schemes

As we noted, we found it curious that, while sentences were scored with trigram and RNN models, completions were generated with a bigram model. We expanded on this, breaking completion into two steps:

1. A function to call is selected. We call this the *code model*.
2. Parameters to pass to the chosen function are selected. We call this the *variable model*.

We expanded this as follows:

- Both models were allowed to use unigram, bigram, and trigram models.
- The code model was given support for a MEMM.

# 3 Implementation Details

Our implementation was written in Python, using the following libraries:

- `PLYJ`, a Java program parser.
- `sklearn`, which supports MEMM training.

The code can be found at `https://github.com/KleinFourGroup/NLP_PL`.

## 3.1 Code Analysis

Much of the time spent on this work went into writing a code analysis tool. This tool did two things:

- Convert the parse tree to code units.
- Convert the Java statements to a simpler "IR."

Some basic type resolution was also included.

## 3.2 Corpora

Since we were not able to obtain the corpus used by [1], we created a corpus of our own. This consisted of Java files from 66 GitHub projects, detailed at `https://github.com/KleinFourGroup/NLP_PL/blob/master/git_dl/source_list.txt`. This totalled roughly 22 MB worth of code. In addition, two of the extracted files were "blacklisted" for crashing the `cfs` model—sequences of if-statements can cause an exponential blow-up.

## 3.3 Baselines

In addition to the models described in 2.2, we also implemented a random baseline, which worked as follows:

- *Code Model*: Pick a random function that has been imported.
- *Variable Model*: Return the first completion that passes a type-check.

## 3.4 Shortcomings

Unfortunately, our implementation lacked support of the following:

- *Expressions as arguments for function calls*: While our IR does express this, we are not able to support it within the language model. (Neither does [1].)
- *Type-casts*: We simply ignore type-casts, as we lack a proper type-resolution system.
- *Non-static fields*: Again, while our IR does express this, we are not able to support it within the language model.

Our implementation has poor support of the following:

- *General type resolution*: Subclasses proved too difficult to implement in time.
- *Static fields*: Again, while our IR does express this, we treat them as constants.

# 4 Results

The full data is available at `https://github.com/KleinFourGroup/NLP_PL/blob/master/LM/results.csv`. Below is a summary:

| levels, max | | | | |
|---|---|---|---|---|
| call model | 3 | 1.3801695908 | 1.4446105766 | 1.471347531 |
| | 2 | 1.3801695908 | 1.4446105766 | 1.471347531 |
| | 1 | 1.3794449235 | 1.4479395829 | 1.4694658828 |
| | MEMM | 1.2973448348 | 1.3617000792 | 1.3816577024 |
| | | 1 | 2 | 3 |
| | | | var model | |

| levels, random baseline | | | | |
|---|---|---|---|---|
| call model | 3 | 1.3502175602 | 1.3991302954 | 1.4090822628 |
| | 2 | 1.3675182533 | 1.3854038403 | 1.4029202412 |
| | 1 | 1.3671579219 | 1.3908003662 | 1.4049350739 |
| | MEMM | 1.2811909286 | 1.3039491778 | 1.3234735757 |
| | | 1 | 2 | 3 |
| | | | var model | |

| CFS, max | | | | |
|---|---|---|---|---|
| call model | 3 | 1.2526546072 | 1.2952245955 | 1.310383277 |
| | 2 | 1.2526546072 | 1.2930782423 | 1.3067884675 |
| | 1 | 1.252504953 | 1.2877275064 | 1.2984496871 |
| | MEMM | N/A | N/A | N/A |
| | | 1 | 2 | 2 |
| | | | var model | |

| CFS, random baseline | | | | |
|---|---|---|---|---|
| call model | 3 | 1.2376532236 | 1.2616989967 | 1.2590966415 |
| | 2 | 1.2263363548 | 1.2873724219 | 1.2797650841 |
| | 1 | 1.2234551087 | 1.2464570138 | 1.2671625827 |
| | MEMM | N/A | N/A | N/A |
| | | 1 | 2 | 2 |
| | | | var model | |

**Average Per-event LL of Completed Code / Average Per-event LL of Actual Code**

## 4.1 Remarks

In most cases, we usually didn't get the right completion:

- We had difficulty with static fields—as mentioned, without detailed type analysis, we had to treat them like constants. As a result, we couldn't dynamically place them, instead selecting them only where they had appeared.
- The model was weighted towards short sentences—with the exception of fields, many variables only generate one or two events. As a result, the model assigned greater likelihood to completions that only used constants.

The MEMM was also severely hamstrung by `sklearn`'s MEMM training:

- Training on sentences generated by `levels` took several minutes.
- Training on sentences generated by `cfs` never finished.

We are still trying to identify how to resolve this?

# References

[1] Raychev, Veselin, Martin Vechev, and Eran Yahav. "Code completion with statistical language models." ACM SIGPLAN Notices. Vol. 49. No. 6. ACM, 2014.