Contextual Code Completion Using Machine Learning

Subhasis Das, Chinmayee Shah
{subhasis, chshah}@stanford.edu
Mentor: Junjie Qin

Abstract

Large projects such as kernels, drivers and libraries follow a code style, and have recurring patterns. In this project, we explore learning based code recommendation, to use the project context and give meaningful suggestions. Using word vectors to model code tokens, and neural network based learning techniques, we are able to capture interesting patterns, and predict code that that cannot be predicted by a simple grammar and syntax based approach as in conventional IDEs. We achieve a total prediction accuracy of 56.0% on Linux kernel, a C project, and 40.6% on Twisted, a Python networking library.

1 Introduction

Code IDEs make the task of writing code in new languages easier by auto-completing key words and phrases. For example, when writing code in C++, an IDE may automatically close an opening parenthesis, and suggest else immediately following an if block. Such code completion has several problems:

1. Grammar based code completion requires writing down an exhaustive set of rules. This is tedious and repetitive when done for every new language.
2. Predictions do not consider the category of code. For instance, driver codes and math libraries may exhibit different patterns.
3. Predictions do not consider context such as class definition, function definition, and tabs and spaces.
4. Recommendations are often ordered lexicographically, which may not be very useful.

This report explores a learning based approach to code completion. Instead of writing grammar based rules, we use machine learning to learn structure and patterns in code. We can then not only automate code predictors for different languages, but also specialize them for different kinds of projects. We can also consider the enclosing context and rank predictions.

2 Related Work

We use natural language techniques for predicting code. In this section, we review some literature on language models and neural network based language models.

2.1 Traditional Language Models

Statistical models of languages have been used extensively in various natural language processing tasks. One of the simplest such models proposed is the n-gram model, where the frequencies of consecutive n tokens are used to predict the probability of occurence of a sequence of tokens. The frequencies of such n-grams are obtained from a corpus, and are smoothed by various algorithms such as Kneser-Ney smoothing [10] to account for sparsity of n-grams. Such models can be easily extended to code completion, by modeling each token as a word and doing n-gram analysis over a codebase. However, such models do not capture the high fraction of unknown tokens from variable and function names very well.

2.2 Word Vector Embeddings and Neural Network Language Models (NNLM)

In their seminal work, Bengio et al. [6] first propose a vector embedding space for words for constructing a language model, that significantly outperforms traditional n-gram based approaches. Their proposal maps each word to a vector and expresses the probability of a token as the result of a multi-layer neural network on the vectors of a limited window of neighboring words. T. Mikolov et al. [14] show that such models can capture word-pair relationships (such as "*king* is to *queen* as *man* is to *woman*") in the form of vector operations (i.e., $v_{king} - v_{\text{queen}} = v_{man} - v_{woman}$). To distinguish such models from *recurrent* models, we call them *feed-forward* neural network language models. We use the same concept in our token-embedding. We found singificant clustering of similar tokens, such as `uint8_t` and `uint16_t`, but were unable to fund vector relationships.

Later, such feed-forward NNLMs have been extended by Mikolov et al. [13] where they propose the continuous bag-of-words (CBOW), and the skip-gram. In CBOW, the probability of a token is obtained from the average of vectors of the neighboring tokens, whereas in skip-gram model the probabilities of the neighboring tokens is obtained from the vector of a single token. The main advantage of such models is their simplicity, which enables them to train faster on billion-word corpuses. However, in our case, the size of data was not so huge, and we could train even a 4-layer model in 3 days.

2.3 RNN Language Models

A significant drawback of feed-forward neural network based language models is their inability to consider dependencies longer than the window size. To fix this shortcoming, Mikolov et al. proposed a recurrent neural network based language model [12], which associates a cell with a hidden state at each position, and updates the state with each token. Subsequently, more advanced memory cells such as LSTM [9] and GRU [8] have been used for language modeling tasks. More sophsticated models such as tree-based LSTMs [15] have also been proposed. We experimented with using GRUs in our setup, but surprisingly did not find them to be competitive with feed-forward networks.

2.4 Attention Based Models

Recently, neural network models with *attention*, i.e., models that weigh different parts of the input differently have received a lot of attention (pun intended) in areas such as image captioning [16] and language translation [5, 11]. In such models, the actual task of prediction is separated into two parts: the attention mechanism which "selects" the part of input that is important, and the actual prediction mechanism which predicts the output given the weighted input. We found an attention based feed-forward model to be the best performing model among the ones we considered.

2.5 Code Completion

Frequency, association and matching neighbor based approaches [7] are used to improve predictions of IDEs such as Eclipse. Our learning approach, on the other hand, attempts to automatically learn such rules and patterns.

3 Dataset and Features

We use two different projects, Linux source [3], a C project, and Twisted [4], a Python networking library, to train and test our methods. In each case, we use half of all files for training, and the remaining half for testing. Given a sequence of tokens in an incomplete piece of code, we predict the next token. We pick these incomplete code pieces randomly from the complete codes in the training set. Tokens in these incomplete codes constitute the features, and the next token occuring in the actual complete code is the "truth". The next section describes how we model these tokens.

4 Modeling Tokens

One of our objectives of learning based code prediction is to do away with the tedious process of building grammar based rules for different languages. We treat codes in the training set in a language agnostic way. The first step is to build a dictionary of tokens or words that can occur in the code, that we can take as input to predict the next token. We build this dictionary by reading all code, and treating each consecutive set of alphanumeric characters and (_), or each consecutive set of special characters other than alphanumeric, (_), space and newline as one token. Thus, for example, the code `for (my_var = 0; my_var < foo; my_var++) {` will be tokenized into eleven tokens: `for`, `my_var`, `=`, `0`, `;`, `my_var`, `<`, `foo`, `;`, `my_var`, `++) {`. Given a sequence of tokens, we then want to predict the next token. Note that these tokens do not correspond exactly to language level tokens, e.g., `++) {` is a single token despite containing three different language level tokens `++`, `)`, and `{`.

The dictionary of tokens constructed as above is not complete, since new code may contain new tokens that are not present in the training data. Moreover, many of the tokens in the training data may be specific to a few files and may never occur again. To address these issues, we divide the tokens into two categories:

a) *Key tokens*: A subset of K most frequent tokens are categorized as *key tokens*. Not surprisingly, many of these frequent tokens are keywords for the language the project is written in, or words that tend to occur often in that particular project. For example, some of the frequent tokens in Linux kernel are: `struct`, `;`, and `dev`, out of which the first and second are keywords in C, and the third is a token frequently used to denote device objects in Linux. Note that while several language keywords do end up being part of the set of key tokens, we *do not* manually curate the list of key tokens to ensure that they contain only language specific keywords.

b) *Positional Tokens*: While key tokens occurrences constitute a major fraction of all token occurrences ($\approx 60\%$ with 2000 key tokens), the remainder of tokens are rarely seen (such as names of variables, macros etc.). However, we would like our learning algorithm to autocomplete new variable names, once they occur in a file. For example, given many examples of the form `for (int TOKEN = 0; TOKEN < n; TOKEN++)` (all with different values of `TOKEN`), our algorithm should be able to autocomplete `for (int myIterName = 0;` with `myIterName`. This capability can not be achieved by merely trying to autocomplete between a set of pre-defined key tokens. Hence, we also define *positional tokens* as follows.

Given a sequence of tokens — an incomplete piece of code, we replace each token which is not a key token by a string of the form `POS_TOK_ii`, where `ii` is the *position* of that token within that sequence. A non-keyword token which is repeated multiple times in a sequence is assigned the index corresponding to its first appearance. For example, given the sequence of tokens `[for, int, myVarName, =, 0, ;, myVarName, <, n, ;, myVarName, ++]`, where `myVarName` is not a key token, we construct the new sequence `[for, int, POS_TOK_2, =, 0, ;, POS_TOK_2, <, n, ;, POS_TOK_2, ++]`. In case the token to be predicted is not a key token but has appeared in the window, it is also replaced by the corresponding positional token string. If the target has not appeared before in the window, it is assumed to be a special token `UNKNOWN`. However, as we describe in Section 5, we ignore such windows since in many such cases the token is actually a hereto unseen token.

Such an encoding is advantageous since now the set of prediction targets to choose from is simply the union of the key tokens and the positional token strings (of the form `POS_TOKEN_ii`). In case of a fixed window size of W and a fixed number of key tokens K, it can be seen that the total number of prediction targets is $K+W+1$ (K key tokens, W positional tokens, and 1 unknown token), i.e., a *constant*. This immediately opens up the possibility of applying simple models such as logistic/neural network based classifiers.

5 Methods

In this model, we simply take the set of $K + W$ positional and key tokens, and the set of $K + W + 1$ output tokens, and fit a model similar to word vectors [13]. The details of the model are described below.

We first assign a D dimensional vector to each one of $K + W$ different key tokens and positional tokens. We denote such token-vectors by v_i, where $1 \leq i \leq K + W$. Next, given a fixed size window of tokens $[t_1, t_2, \ldots, t_W]$, we compute a score for each possible output j, s_j as a function of the token-vectors of the tokens in the window, i.e.,

$$s_j = f_j \left(v_{t_1}, v_{t_2}, \ldots v_{t_W} \right)$$

The final loss function for this particular example is given by

$$L = \log \left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}} \right)$$

where t_o is the output token. This is the cross-entropy loss between softmax based probabilities for each output and the actual observed output. According to the actual form of the function f_j, we get different models. A few of these models are described below.

For each model, we use AdaGrad optimizer to minimize the total loss function with respect to the parameters of that model. AdaGrad was chosen because it gave the best performance in our case among other alternatives such as vanilla SGD and momentum based SGD.

5.1 Fixed Window Weight Model

In the spirit of continuous bag-of-word (CBOW) model [13], in this model we assume that a token at position i has a "weight" of w_i, and combine the token-vectors of the window according to these weights. The final score is assumed to be a linear function of this weighted token-vectors. Thus, the overall model is

$$u = \sum_{i=1}^{W} w_i v_{t_i} \tag{1}$$

$$s_j = p_j^T u \tag{2}$$

$$L = \log \left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}} \right) \tag{3}$$

The parameters in this model are the weights w_i, the word vectors v_i, and the "prediction" vectors p_j. The gradients of the loss w.r.t. the parameters are obtained by backpropagation, the details of which are omitted here for space.

5.2 Matrix Vector Model

In this case, we do not introduce any averaging as in the case of CBOW, but instead simply concatenate the token-vectors to create a larger vector which is then used to create the scores. Formally, the model is

$$u = [v_{t_1}; v_{t_2}; v_{t_3}; \ldots; v_{t_W}] \tag{4}$$

$$s_j = p_j^T u \tag{5}$$

$$L = \log \left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}} \right) \tag{6}$$

Note that since here u is DW dimensional instead of being D dimensional as in the case of Fixed Window Weight Model. Thus, the prediction vectors p_j are much higher dimensional as well, which means this model has a higher number of parameters as compared to Fixed Window Weight Model.

5.3 Feed-Forward Neural Network Model

This case differs from the Matrix Vector Model by addition of one or more non-linear transformations between the concatenated token-vectors and the final scores. We denote a specific instance of this model by NL-k, where k is the number of non-linearity layers it contains. Thus, for example, NL-0 is equivalent to the Matrix-Vector model described above, while for NL-3 the probabilities are obtained as:

$$u = [v_{t_1}; v_{t_2}; v_{t_3}; \ldots; v_{t_W}] \tag{7}$$

$$z_1 = \text{relu}(Q_1 u) \tag{8}$$

$$z_2 = \text{relu}(Q_2 z_1) \tag{9}$$

$$z_3 = \text{relu}(Q_3 z_2) \tag{10}$$

$$s_j = p_j^T z_3 \tag{11}$$

$$L = \log \left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}} \right) \tag{12}$$

Here relu(.) is the rectified linear unit, i.e., $\text{relu}(x) = x$ if $x \geq 0$, and 0 otherwise. In this work, we specifically experimented with NL-0, NL-1, NL-2, and NL-3.

5.4 Feed-Forward Model with Soft Attention

Motivated by the recent successes of attention based models, we explore an attention mechanism for our problem as well. We experiment with a "soft"-attention model, i.e., a weight a_i between 0 and 1 is assigned to each position i. The attentions are assumed to be the result of a sigmoid function on a linear combination of the concatenated word-vectors. Subsequently, the word vector for the i^{th} token is weighted by a_i, and the aforementioned NL-k model is applied on the concatenation of these weighted word vectors instead. Thus, for example, an attention based NL-3 model takes the form of:

$$u = [v_{t_1}; v_{t_2}; v_{t_3}; \ldots; v_{t_W}] \tag{13}$$

$$a = \sigma(Au) \tag{14}$$

$$z = [a_1 v_{t_1}; a_2 v_{t_2}; a_3 v_{t_3}; \ldots; a_W v_{t_W}] \tag{15}$$

$$s_j = \text{NL-3}(z; Q_1, Q_2, Q_3, p_j) \tag{16}$$

$$L = \log \left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}} \right) \tag{17}$$

Here NL-3$(x; Q_1, Q_2, Q_3, p_j)$ is the function going from u to s_j in the case of a NL-3 model, described in Section 5.3.

5.5 GRU based Recurrent Model

Recurrent neural network models based on cells like LSTM [9] and GRU [8] have recently been shown to achieve state-of-the-art performance in language models. Inspired by these results, we also attempted to use a GRU based recurrent model for our prediction task. Specifically, our model was the following:

$$[g_1; g_2; g_3; \ldots; g_W] = \text{GRU}([v_{t_1}; v_{t_2}; v_{t_3}; \ldots; v_{t_W}]) \tag{18}$$

$$s_j = p_j^T [g] \tag{19}$$

$$L = \log \left(\frac{e^{s_{t_o}}}{\sum_j e^{s_j}} \right) \tag{20}$$

Here, g_i is the output of the i^{th} GRU cell, and we have a dense layer after that to get the scores for each output token. As we did not achieve competitive performance with this model as compared to NL-1, we did not experiment further with deeper GRU models.

6 Experiments and Results

In this section, we evaluate the methods outlined in Section 5, on Linux kernel and Twisted library. We first present the accuracy of predictions on the test set, and then present some interesting predictions that come out from recurring patterns. We implemented the matrix vector model in Python, and the feed-forward and recurrent models in Keras [2]. Our code is available on github [1].

6.1 Accuracy

Method	Known	Abs	Top 3	Key	Pos
Win, #Keys	acc.	acc.	acc.	acc.	acc.
NL-3, 40, 2000	64.4	53.4	81.7	78.0	43.7
NL-4, 64, 2000	63.2	50.6	82.0	72.8	41.8
Attn, 40, 1000	67.5	56.0	83.6	79.9	48.4
BestKW-RandPos	49.8	41.3	–	79.9	7.0

Table 1: Test accuracy (%) of predictions for Linux project

Method Win, #Keys	Known acc.	Abs acc.	Top 3 acc.	Key acc.	Pos acc.
NL-3, 40, 500	46.3	39.2	64.8	64.1	14.2
NL-4, 40, 500	44.8	37.7	62.2	63.9	10.8
Attn, 40, 500	47.9	40.6	66.5	64.6	17.6
GRU, 40, 500	41.6	35.2	59.1	64.4	4.9
BestKW-RandPos	43.6	37.5	–	64.4	5.5

Table 2: Test accuracy (%) of predictions for Twisted project

Table 1 shows results for Linux source, a C project, and Table 2 for Twisted, a Python networking library. As mentioned in Section 3, we use half the files for training and half for testing. Column 1 lists the learning method, window size and number of key words. NL-3 is a feed forward model with 3 non-linearity layers, and NL-4 is a feed forward model with 4 non-linearity layers. Attn. computes a weighing parameter for each input token in the window, and uses 3 non-linearity layers. GRU is a recurrent model. We do not report numbers for GRU in the Linux codebase since it gave poor results in the smaller Twisted codebase. Additionally, we report numbers for a fictional random predictor that achieves the accuracy of the best predictor in case of keywords, but in case of a non-keyword token predicts one randomly from the non-keyword tokens in the window, to show the significance of positional tokens. We name this BestKW-RandPos.

Known acc. is the accuracy for predictions for the cases where the next token is a key token, or a positional token from the window being considered. Abs acc. is the accuracy for predictions for all cases, including cases where the next token is neither a key word nor a seen positional token. These are often new function names or variable names, or variables outside of our window. Top 3 is the percentage of cases where the next token is within top 3 predictions, ignoring the unknown cases (unseen variable and function names). We report this number, because generally, we are interested in the top few suggestions as opposed to the top suggestion with a code recommendation system. Key acc. is the accuracy with which we predict key tokens correctly, given the next token is a key token, and pos. acc. is the accuracy with which we predict positional tokens correctly, given the next token is a positional token. These give us insight into how well our methods predict tokens that are not key tokens.

We found that attention gives the best results among all the methods we tried. This makes sense because it is also the most general of all models — it directly connects all inputs to the output, (unlike GRU where computed states are chained), and it has an additional attention parameter, that is just a constant in the simple feed-forward case. We also found that the feed forward model with 4 non-linearity layers tends to overfit the training data, giving a slightly lower accuracy on test data. Increasing the word vector dimensions also tends to result in overfitting. Though regularization can help with overfitting, we found that reducing vector dimensions and number of layers was also sufficient to reduce high variance from overfitting.

The accuracy of predicting non-key words, that is, positional tokens is quite high for Linux kernel, a C project, indicating there is a lot of redundancy/ recurring patterns. The positional token accuracy is lower for Python. We think this is because Python is a dynamically typed, high level language with less redundancy, and more terse syntax. However, in both cases, the top 3 prediction accuracy is still much higher than a random predictor, indicating that the model does extract some patterns out of the code.

A plot of the word vectors reveals a significant clustering of similar words — `uint8_t` and `uint16_t`, different locking and unlocking calls, and integers. We have omitted these plots from the report for space. Unlike natural language texts, however, we did not observe any significant vector relationships between different tokens.

6.2 Examples

Another experiment we ran was to evaluate the accuracy of the classifier in a setting where, if the correct token is not within the top few predictions, the user types in one or more characters to "home in" on the correct prediction. Figure 1 shows the results for this experiment. In this experiment, a character is colored in green if the token is correctly predicted (i.e., top prediction) before that character is typed. It is colored in yellow if the token is among the top 5 before typing that character, and colored in red if the token is not among the top 5. So, for example, in the first line in the token `struct`, the character `s` is in yellow since before typing `s`, the token `struct` was among the top 5. After typing `s`, the top prediction was `struct`, and thus the characters `truct` are all in green. We can see that, for example, entire sequences of code such as `{ PCI_VDEVICE(SUNDANCE,` are predicted correctly without typing anything.

```
static const struct pci_device_id ipg_pci_tbl[] = {
        { PCI_VDEVICE(SUNDANCE, 0x1023), 0 },
        { PCI_VDEVICE(SUNDANCE, 0x2021), 1 },
        { PCI_VDEVICE(DLINK,    0x9021), 2 },
        { PCI_VDEVICE(DLINK,    0x4020), 3 },
        { 0, }
};

MODULE_DEVICE_TABLE(pci, ipg_pci_tbl);
```

Figure 1: Code example with per-character predictions

6.2.1 Importance of Positional Tokens

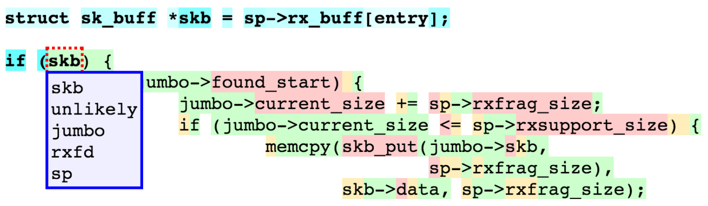
From Figure 1 we can also see the importance of positional tokens. The tokens `PCI_VDEVICE` and `SUNDANCE` are both not keywords (i.e., not frequently observed in the codebase), and yet were predicted correctly by our method the second time they were used (in line 3). If we chose to ignore such tokens instead, we would not have been able to predict such tokens correctly in a majority of cases.

6.2.2 Learning Frequently Occurring Patterns

We also found several interesting patterns that are predicted correctly by our system. An example of such a pattern is given in Figures 2a & 2b. In these figures, each character is color coded as before, and some additional information is also shown. The top 5 predictions at the position of interest (shown in red border), are shown in a blue box. Also, the value of attention for each token in the prediction window are shown color coded in sky-blue, where deeper blue is indicative of more attention (for reference, the first `if` tokens in both of these two cases have an attention value of $\approx$ 1).

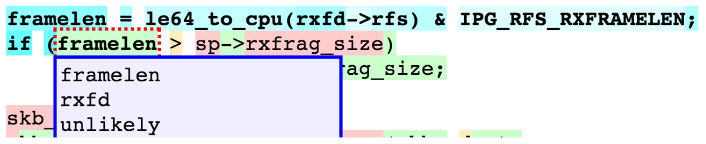
Here, a variable is assigned to right before an `if` statement, and the prediction is that the variable will be immediately used inside the `if` condition. This is a common pattern in Linux code, where if the condition inside `if` is too long, it is broken up into an assignment and a subsequent conditional.

Also, in the figure the top 5 predictions for each case is given in the blue box. We can see that `unlikely` is a common prediction, since `if (unlikely(condition))` is a common pattern in



```
struct sk_buff *skb = sp->rx_buff[entry];

if (skb) {
   skb       umbo->found_start) {
   unlikely     jumbo->current_size += sp->rxfrag_size;
   jumbo        if (jumbo->current_size <= sp->rxsupport_size) {
   rxfd                 memcpy(skb_put(jumbo->skb,
   sp                                         sp->rxfrag_size),
                              skb->data, sp->rxfrag_size);
```

(a) Example 1



```
framelen = le64_to_cpu(rxfd->rfs) & IPG_RFS_RXFRAMELEN;
if (framelen > sp->rxfrag_size)
    framelen                             ag_size;
    rxfd
skb_unlikely
```

(b) Example 2

Figure 2: Definition before `if` pattern

Linux code to check for unlikely edge cases such as error conditions.

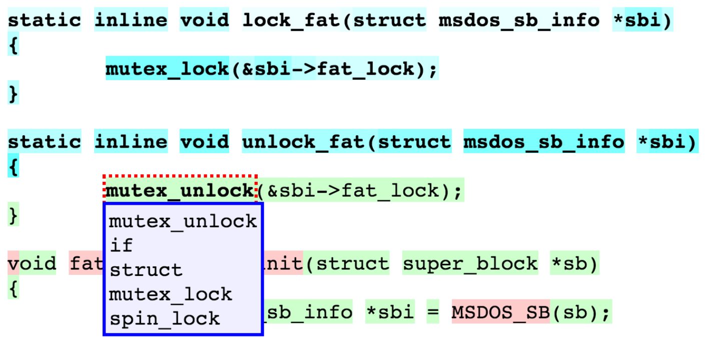
We can also see that the attention of the models is relatively high at the correct predictions (`skb` and `framelen` in the first and second cases respectively). This shows that even a single layer model can be expected to fairly reliably predict the next token (since the attention is derived from a single non-linear layer).

A few more interesting examples are shown in Figure 3. A description of why each of the cases are interesting are given in the captions.

7 Conclusions and Future Work

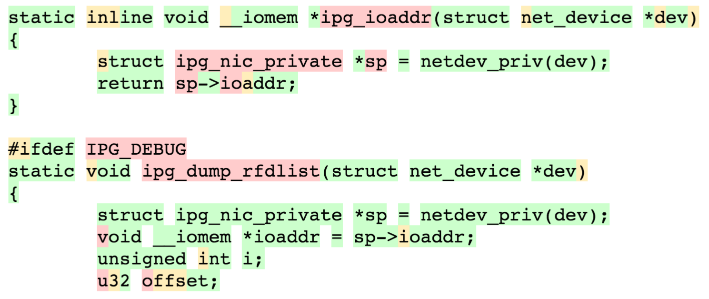
We found that a learning based approach extracts some interesting patterns, that cannot be captured by language rules. We model tokens using dense word vectors and use natural language processing methods such as a simple matrix-vector model, feed forward neural network models, attention models and recurrent models. These models do not use any information about the language grammar or syntax. The attention based model performs the best, because it directly connects each input token to output, and also computes weights for important tokens. It is the most general of all models, with the most parameters. We found that by adjusting the vector dimensions and number of layers, we could reduce overfitting, and get close accuracies for training and test data. Our top 3 predicted tokens give an accuracy of over 80% for the cases where the next token is known (a key or a postitional token), for Linux, and over 60% for Twisted. A study of these vectors reveals a significant clustering of related tokens.

Though we did not use any information about language grammar and syntax, it would be interesting to combine language rules with our learning based approach. For example, we could prune our predictions to those that are only syntactically valid. We could also increase our window size, and jointly process `.h` and `.c` files, to improve the context. Such large contexts may not scale well with feed-forward model, but recurrent models are known to perform well with large inputs. We could also try to use language rules to list out all possible options for next token, and then chain that to a learning based model to improve the predictions. We could also try to learn these language rules instead of listing them, by processing a large number of different projects.



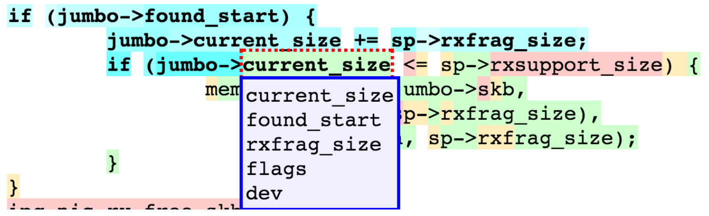
```
static inline void lock_fat(struct msdos_sb_info *sbi)
{
        mutex_lock(&sbi->fat_lock);
}

static inline void unlock_fat(struct msdos_sb_info *sbi)
{
        mutex_unlock(&sbi->fat_lock);
}       mutex_unlock
        if
void fat struct     nit(struct super_block *sb)
{       mutex_lock
        spin_lock   sb_info *sbi = MSDOS_SB(sb);
```

(a) `mutex_lock` and `mutex_unlock` pair on same lock variable



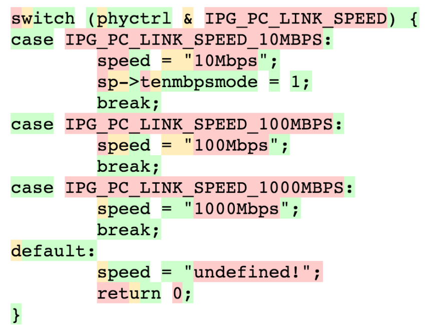
```
static inline void __iomem *ipg_ioaddr(struct net_device *dev)
{
        struct ipg_nic_private *sp = netdev_priv(dev);
        return sp->ioaddr;
}

#ifdef IPG_DEBUG
static void ipg_dump_rfdlist(struct net_device *dev)
{
        struct ipg_nic_private *sp = netdev_priv(dev);
        void __iomem *ioaddr = sp->ioaddr;
        unsigned int i;
        u32 offset;
```

(b) Duplicate code after function definition predicted correctly



```
if (jumbo->found_start) {
        jumbo->current_size += sp->rxfrag_size;
        if (jumbo->current_size <= sp->rxsupport_size) {
                mem current_size  umbo->skb,
                    found_start   p->rxfrag_size),
                    rxfrag_size   , sp->rxfrag_size);
        }           flags
}                   dev
```

(c) Member variables predicted correctly



```
switch (phyctrl & IPG_PC_LINK_SPEED) {
case IPG_PC_LINK_SPEED_10MBPS:
        speed = "10Mbps";
        sp->tenmbpsmode = 1;
        break;
case IPG_PC_LINK_SPEED_100MBPS:
        speed = "100Mbps";
        break;
case IPG_PC_LINK_SPEED_1000MBPS:
        speed = "1000Mbps";
        break;
default:
        speed = "undefined!";
        return 0;
}
```

(d) `case` token predicted correctly after each `break;`

Figure 3: Interesting code patterns predicted correctly

References

[1] Contextual code completion. `https://github.com/subhasis256/ml_code_completion`.

[2] Keras. `http://keras.io/`.

[3] Linux source code. `https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.2.3.tar.xz`.

[4] Twisted source code. `https://github.com/twisted/twisted.git`.

[5] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

[7] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceed-*

ings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 213–222. ACM, 2009.

[8] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995.

[11] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[12] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048, 2010.

[13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[14] T. Mikolov, W.-t. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751, 2013.

[15] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[16] K. Xu, J. Ba, R. Kiros, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.