Suggesting Comment Completions for Python using Neural Language Models

Ciurumelea, Adelina ; Proksch, Sebastian ; Gall, Harald C

DOI: https://doi.org/10.1109/SANER48275.2020.9054866

Suggesting Comment Completions for Python using Neural Language Models

Adelina Ciurumelea
University of Zurich
Switzerland
ciurumelea@ifi.uzh.ch

Sebastian Proksch
University of Zurich
Switzerland
proksch@ifi.uzh.ch

Harald C. Gall
University of Zurich
Switzerland
gall@ifi.uzh.ch

***Abstract*—Source-code comments are an important communication medium between developers to better understand and maintain software. Current research focuses on auto-generating comments by summarizing the code. However, good comments contain additional details, like important design decisions or required trade-offs, and only developers can decide on the proper comment content. Automated summarization techniques cannot include information that does not exist in the code, therefore fully-automated approaches while helpful, will be of limited use. In our work, we propose to empower developers through a semi-automated system instead. We investigate the feasibility of using neural language models trained on a large corpus of Python documentation strings to generate completion suggestions and obtain promising results. By focusing on confident predictions, we can obtain a top-3 accuracy of over 70%, although this comes at the cost of lower suggestion frequency. Our models can be improved by leveraging context information like the signature and the full body of the method. Additionally, we are able to return good accuracy completions even for new projects, suggesting the generalizability of our approach.**

***Index Terms*—source code comments, naturalness, language models**

I. Introduction

Comments are an essential part of any software project, they support the understandability and maintainability of a project. As mentioned by John Ousterhout [37], *"they should describe things that aren't obvious from the code"* and should be written at a different level of abstraction than the code they accompany. The goal of comments is to lower the cognitive load of a developer while comprehending a piece of code and document important information that is necessary for the evolution and maintenance of a project. Nevertheless, writing comments is time consuming and cumbersome, the developer has to carefully summarize what a piece of code does while documenting relevant decisions and trade-offs they have made, additionally the value of good comments, as for tests, is not as obvious as working features. Developers often postpone them until the end of the development process or do not write them at all [37]. Furthermore, modern IDEs offer limited support in creating them.

Researchers have analyzed the contents of comments [19] and concluded that these contain more than just summaries of the code being written [39]. This observation is also corroborated by practitioners [30], [6]. However, most research focuses on automatically generating source code comments [44], [24], while helpful, these approaches are limited to the information available in the code. They still need the contribution of the developer to write helpful and complete comments. We believe a more promising direction is to support the developer while composing comments by providing accurate completion suggestions generated using a neural language model. This support should be twofold: on one hand we would like to reduce the time and effort required to write comments, on the other hand we would like to help developers write more natural comments.

Generating text completions is a well known problem in natural language processing [26] and can be solved using statistical language models. These have already been applied to a wide variety of problems in software engineering research: source code completion [22] [8], finding bugs [42] [28], type inference [20], and evaluating open source contributions [21] among others. However, to the best of our knowledge, there is only one paper [35] that investigates the use of language models for source code comments completion. Their approach is based on n-gram models and focuses on completing words that the developer has started typing. As newer language models can even help users compose emails [9], we explore their application to support developers with writing Python documentation string (docstrings) through the generation of complete word suggestions. In this paper, we focus on the following research questions:

RQ1: Are docstring comments natural? Do they show similar repetitiveness and predictability characteristics as natural language and source code?

RQ2: Will context information improve the predictions of neural language models trained on Python docstrings?

RQ3: How well can neural language models, trained on Python docstrings, generate completion suggestions for projects not seen during training?

By comparing language models trained on English text and Python docstrings we can confirm that docstrings show similar repetitiveness and predictability characteristics as natural language. The use of context information (the method signature or body) does, indeed, improve the prediction accuracy. We observed that by only considering high probability completion suggestions, we can significantly increase the prediction accuracy, at the cost of reducing the suggestion frequency.

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

Finally, using our language model for projects not seen during the training, leads to similar suggestion accuracies as for the projects that are part of the training set, confirming the generalizability of our approach.

In summary, these are the main contributions of this paper:

- investigating the feasibility of generating completion suggestion for Python docstrings using neural language models;
- integrating context information into neural language models to improve the suggestion accuracy.

II. Related Work

Previous work has studied the contents of comments to understand the types of comments developers write, what kind of programming constructs they accompany and their frequency. Additionally, researchers have acknowledged that developers need support in documenting code and investigated approaches to automatically generate comments from source code. In the next paragraphs, we describe several papers addressing these topics and how they relate to our work.

1) Content of Comments: Haouari et al. [18] conducted two studies to better understand source code comments: the first one looked at the distribution and frequency of comments depending on different programming constructs and noticed that method comments are the most common. During the second study, human participants looked at the content and relevance of comments and observed that most of them refer to the subsequent code and that method declarations are well explained. Their findings support our decision to first focus on method and function comments, as developers seem to prioritize them. Steidl et al. [45] developed a semi-automatic method for analysing the quality of comments. One of their metrics computes the coherence between code and comments based on the Levenshtein distance between the words in the method name and in the comment: if the two are too similar then the comment is likely trivial and not useful; if they are too different then either the identifier name needs to be refactored or the comment is not sufficient. They believe that the method name is likely to be relevant while writing the comment. This encourages our approach to use language modeling on a large corpus of comments for generating completion suggestions, but also to condition the suggestions on context information, such as the method signature.

Pascarella et al. [39] built a detailed taxonomy taking into account the purpose of Java comments after analysing 6 OSS Java projects. With an automated approach for classifying comments according to their categories, they noticed that even though *summary* comments are quite common, they only represent 24% of the overall comments. In particular, two of the categories they identified: *expand* (provides more details on the code itself) and *rationale* (explains the reason behind some choices, patterns, and options) further support our belief, that comments contain more than just summaries of code.

2) Automatic Generation of Comments: Previous approaches focused on automatically generating summaries for Java methods [17], [44] and classes [34]. For source code without documentation, generating accurate and concise natural language summaries is going to be helpful. Nevertheless, such approaches can only include information that is contained in the code, important design decisions and details cannot be retrieved. Moreover, automatically generated summaries are unlikely to have the same quality as those written by humans.

Iyer et al. [25] uses an approach based on neural attention models to generate high-level summaries of source code snippets in natural language, interestingly, they use StackOverflow data to train their model. They do not focus specifically on generating comments and their approach could also be used for code retrieval. Oda [36] employs a traditional machine translation system to solve a related, but different problem: generating pseudo-code for source code. Their goal is to aid beginner programmers, or programmers that are not familiar with the programming language of the project they are trying to comprehend. Hu et al. [24] uses neural machine translation to automatically generate comments for Java methods, they model the problem of comments generation as a translation problem between source code and natural language text. They only focus on generating the first sentence of a Javadoc, which is usually a summary of the method. Similarly, Liang et al. [29] generate descriptive comments for Java code blocks using a recursive neural network.

All the related work we described focuses on the automated generation of summaries or pseudo-code to aid the comprehension of software projects for either experienced developers that need to quickly understand the main goal of a method/class or beginner developers that require fine grained natural language descriptions of the code. By contrast, we focus on an approach to support developers in the process of writing comments, to reduce the required time and effort necessary to write them. Our approach is semi-automated, as it uses the feedback of the developer, what they have typed so far, to generate full word completions, and it is not limited to only summaries.

III. Approach

This section introduces background information related to Python docstrings and language models for readers not familiar with them. We then describe our approach for generating docstring completion suggestions.

A. Python Documentation Strings

A Python documentation string (docstring) is a string literal that occurs as the first statement in a module, function, class or method definition [3]. They are typically surrounded by triple quotes and can be written on one or multiple lines. The first line is usually a summary and if other lines exists, they elaborate on the summary and document arguments, return values, exceptions or side-effects. Figure 1 contains a screenshot of a docstring from a Github project. Various formatting guidelines are used in practice, like, Numpy/Scipy, Pydoc, Epydoc, Google docstrings [1]. However, there is no consensus on which kind of format should be used. We noticed that in our dataset, some projects, utilize several conventions.

```
def check_query_status(self, query_execution_id):
    """
    Fetch the status of submitted athena query. Returns None or one of valid query states.

    :param query_execution_id: Id of submitted athena query
    :type query_execution_id: str
    :return: str
    """
    response = self.conn.get_query_execution(QueryExecutionId=query_execution_id)
    state = None
    try:
        state = response['QueryExecution']['Status']['State']
    except Exception as ex:
        self.log.error('Exception while getting query state', ex)
    finally:
        return state
```

Fig. 1. Python Documentation String Example

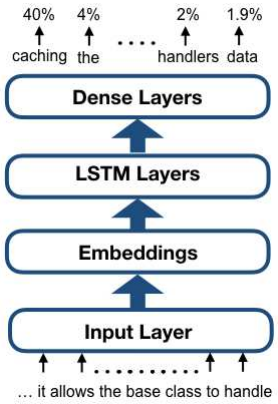

Fig. 2. Sequential Model

B. Language Models

Statistical language models learn the regularities of the corpus they have been trained on. Such models can be used to calculate the probability for arbitrary sentences that they come from the same corpus [15], [11]. Additionally, a language model can compute the likelihood that a prefix sequence is followed by a specific word. By iterating over the complete vocabulary of the language model, it is possible to identify the most likely continuation of an input sequence. For example, given the sentence *"how are you"*, the likelihood of the word *"doing"* is high. This property solves the problem described in this paper, that is generate relevant next words to developers while they are writing comments.

Traditional approaches for training language models are based on n-grams, which assume the Markov independence property: the probability of the *n-th* word in a sentence is only dependent on the last *n-1* words:

$$P(w_{i+1} = m|w_{1:i}) \approx P(w_{i+1} = m|w_{i-(n-1):i})$$

These probabilities can be estimated using the training corpus counts. While efficient and easy to use, n-gram models have several drawbacks: they require back-off and smoothing techniques to account for the sparsity of data and scaling to larger n-gram sizes is very expensive in terms of memory requirements. Moreover, they cannot generalize across contexts, e.g. seeing a *"red bicycle"* and a *"black bicycle"* does not influence the estimates for the sequence *"blue bicycle."*

Parts of the disadvantages of n-gram models can be tackled using neural networks, in particular Recurrent Neural Networks (RNNs) that are capable of conditioning the next word on previously seen words in the corpus. RNNs represent arbitrarily sized sequential inputs as fixed-sized vectors while capturing their statistical and structural properties. This allows developing language models that do not have to make the simplifying Markov assumption and can take an input sequence of arbitrary length when predicting the next word. Nevertheless, vanilla RNNs encounter problems with exploding and vanishing gradients during training. Therefore in practice the best results are obtained using gated architectures such as LSTMs [23] and GRUs [10]. Language models are typically evaluated using the intrinsic metric *perplexity*, or its log-transformed version *cross-entropy*. Both are measurements from information theory that indicate how surprised a model is by a sentence. Good models should return low values for sentences from the corpus. The cross-entropy H_{LM} of a language model LM with respect to a corpus is formally defined as:

$$H_{LM} = -\frac{1}{n}\sum_{i=1}^{n} log_2 P_{LM}(w_i|w_{1:i-1})$$

This metric is useful for comparing models, but to understand how well a model can solve a task, it is common to define task specific metrics. In our case, we are interested in understanding how well a language model is able to generate completion suggestions. Therefore, we measure and report the top-1, top-3, top-5 and top-10 accuracy, which measures how often the correct suggestion is contained in the first 1, 3, 5 and 10 suggestions. We define accuracy as the ratio of correct predictions with respect to all predictions being made.

C. Models for Generating Completion Suggestions

We only focus on RNN-based models for generating completion suggestions, as they have been shown to be better at modelling language than n-gram based models [32], [33] and [27]. Our models learn, given a training corpus of Python documentation strings, to predict what is the most likely word following a particular sequence. We use sequences extracted from docstrings for training. Additionally, we include context information from the corresponding method body for a particular comment, to help the model generate better suggestions. Next, we describe the architecture of the two models we developed: a Sequential and a Context Model.

1) Sequential Model: this is an LSTM language model [23] that – as the name suggests – follows a sequential architecture with an input layer, an embedding layer, one or more LSTM layers and finally two or more fully connected layers separated by Dropout layers as presented in Figure 2. The *input layer* receives as data the sequence of words for which we would like to generate a completion. Then the *embedding layer* is able to associate each word in the input with a low-dimensional floating-point vector. At the beginning, these vectors are

initialized with random values; during training, the network learns to pair each word with a meaningful representation that maps semantic relationships between words into a geometric space [31] [11]. For example, synonyms will be mapped to very similar vectors, while words with different meanings will be mapped to points further away from each other. The output of the *embedding layer* is still sequential, therefore we use an *LSTM layer* that is able to processes sequences and to keep an internal state while processing the individual words. At each timestep (after processing a single word in the sequence), this layer returns an output, which is the representation of the input it has received so far. We only use the results returned after the last word, which contain the representation of the entire sequence. These are then passed to a first *dense (fully connected) layer* that is able to extract meaningful features from this representation and further sends its output to a final *dense softmax layer*. Between these two layers we add a *dropout* layer to decrease overfitting. The last layer has the output dimensionally equal to the vocabulary size and returns a probability for each word in the vocabulary signifying how likely it is to be the next one in the sequence.

A regular language model learns to make predictions based on the prefix sequence, however a comment, in particular a docstring, will be surrounded by relevant context information – like the method signature and body – that can be used to improve the predictions. To take advantage of this additional information, we built the Context Model, described next.

2) Context Model: this is a multi-input model that generates predictions based on the sequence of words and additional context information. We trained two variants, and use either the method signature or all the identifiers from the full body as context information. In both cases, the context sequence is passed through an *embedding layer* to obtain the vector representation for each word. These are then sent to a *global average pooling layer* to compute the average embeddings of the context information. The input sequence for which we are generating suggestions is passed through an *embedding layer* and the vector representation for each single word is concatenated with the average embeddings of the context information. The rest of the model follows the Sequence Model architecture and is included in Figure 3.

Traditionally, context information is added to a language model through a sequence-to-sequence framework [46], which allows expressing a task as a mapping between a sequence of n items to a sequence of m items [15]. Goldberg [15] suggests that this is not always the best approach and other architectures are easier to learn or better suited for specific tasks. For this reason, we adopted an architecture for the Context Model that is inspired by the one used to generate completion suggestions in email texts [9]. In future work, we plan to further investigate this design decision and how it compares to alternatives.

IV. Methodology

In this section, we describe how we collected our dataset, the procedure for preprocessing and extracting training instances, and provide details about the trained models.

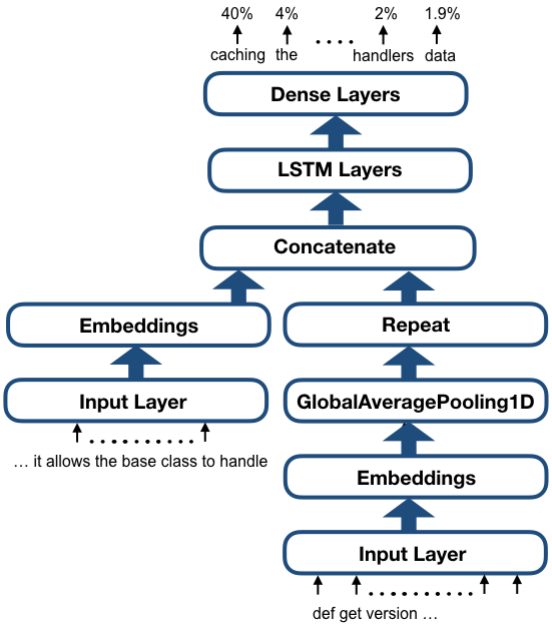

Fig. 3. Context Model

A. Collection of the Dataset

For the creation of the dataset we first crawled the source code of popular Python projects from GitHub in November 2018. To ensure a large enough collection of diverse and realistic projects, we selected the top-1000 most starred projects. The stars metric has been used by previous work [38], [16] to select open source projects, however it is likely not enough to identify mature projects. For this reason, we employ the additional filtering steps outlined bellow to improve our dataset. We extract the docstring comments from all the functions and class methods using the ast module [4] and noticed a large variety regarding the number of docstrings. While a few projects feature several tens of thousands of docstrings, the majority of projects define substantially fewer, with many projects in the lower hundreds and less. To not bias our dataset to the dominant projects and eliminate the lower quality ones, we keep a maximum of $1,000$ comments per project and filter out projects with less then 100 comments resulting in a remaining set of 397 projects. We eliminate duplicate comments inside, but also across projects, as recommended in Allamanis et al. [7]. Finally, we split the dataset into five folds, with each fold containing 80% of the data for training, 10% for validation and another 10% for testing. We have chosen a five-fold split as opposed to ten-fold split because of limited computational resources. The obtained sets and their corresponding averaged sizes are described in Table I.

B. Generation of Training Instances

The training instances are extracted after applying a preprocessing step on the documentation strings and the cor-

TABLE I
DATASET DETAILS: FUNCTION/METHOD TO DOCSTRINGS

Data	Nr. of docstrings - function/method pairs
Training Data	132,839
Validation Data	15,982
Testing Data	16,116

TABLE II
PREPROCESSING EXAMPLE.

Preprocessed Docstring	Preprocessed Context
<start> fetch the status of submitted athena query returns none or one of valid query states <param> <ident1> id of submitted athena query <type> <ident1> str <return> str <end>	check query status self query execution id response self conn get query execution query execution id query execution id state state response exception ex self log error ex state

responding method bodies. The preprocessed results for the example in Figure 1 are included in Table II. Next, we describe the steps for preprocessing a docstring. First, special tokens from the formatting style guide (e.g. :param, :return) are replaced with corresponding generic tokens (e.g. <param>, <return>). Numbers are substituted with a generic token (the most common integers from -1 to 6 are replaced with <nr-1> to <nr6>, and the rest with <nr>). The occurrences of formal parameter names from the method signature are substituted with numbered tokens (e.g. the first formal parameter is replaced with <ident0>). We remove lines corresponding to doctests [2], which describe Python interactive sessions and contain snippets of code and the expected results, as they are not common enough in our dataset and the model would get confused otherwise. Punctuation signs are removed and all characters are lowercased. We do not replace occurrences of identifier names that are not formal parameters, these are preserved and are not split based on either camel case (e.g. *createModel*) or snake case (e.g. *create_model*) conventions. The reason for doing this, is that it would be very difficult to correctly and reliably identify them and the model should still be able to predict identifiers that occur frequently enough in the training set. Finally, we add the <start> and <end> tokens to the beginning and end of the docstring. For preprocessing the method body, we first split it into tokens using the Python tokenize module [5] and we keep all tokens that are identified as names. These are then split into words using the camel case and snake case conventions and all characters are lowercased and we filter out all Python reserved keywords. For a neural model to learn meaningful representations for words, it needs enough training instances where a particular word occurs, therefore to eliminate noise and obtain a reasonable sized vocabulary, all words that appear less than three times in the training set are replaced with the UNK token, both for the documentations strings and the methods.

Given the preprocessed docstrings we extract sequences of length five accompanied by the corresponding context (signature or full method body). The first four words represent the prefix sequence and the fifth one is the target word which a model should predict. The average number of sequences obtained for the different datasets and folds are included in Table III. We wanted to generated suggestions already at the beginning of a docstring, therefore we used a short length of five words, although longer sequences would likely benefit the model. Docstrings with less than five words are padded. The collected datasets [12] and the source code [13] are available online.

TABLE III
DATASET DETAILS: NUMBER OF INSTANCES

Dataset	Instances
Training Data	4,100,228
Validation Data	495,393
Test Data	499,067

C. Training Details

We trained multiple neural network models to generate completion suggestions on a cluster with dedicated GPUs. Each training was repeated three times with different seeds for each of the TRAINING DATA folds, and was then evaluated on the corresponding TEST DATA fold by computing the cross-entropy loss and accuracy. All reported values are obtained by averaging the results from these multiple runs, unless indicated otherwise. We compute the top-3, top-5 and top-10 accuracies to understand the practical usefulness of the generated suggestions. All models are implemented using the Keras and Tensorflow frameworks.

We report and discuss the results in Section V for the three main models: a Sequential and two Context Models. The first Context Model uses the method signature as context and the second one the full body. The configuration for the Sequential Model includes an embedding size of 150, a first dense layer of size 100 with an Elu activation function [14] and the last dense layer has a size equal to the vocabulary and uses the softmax activation function. The first Context Model has the same configuration for all the layers, with an additional LSTM layer of size 200 with dropout and recurrent dropout equal to 0.2. The length of the context information is set to 10 tokens for the method/function signature and to 100 tokens for the full body to cover at least 90% of the data. For the second Context Model we increased the size of the embedding layer to 512, we experimented with smaller sizes but we noticed that increasing the embedding size leads to higher accuracies. The reason for this is likely caused by the loss of information when we average over the embeddings of the context, this loss is more pronounced when using the full method body as the length of the context is larger. Therefore to preserve enough information for the model, it is helpful to increase the embedding size. All models are trained using stochastic gradient descent and the Adam optimizer with a learning rate of $3\mathrm{e}{-4}$ and the categorical cross-entropy loss function for 20 epochs. The configurations of the models we described

above were obtained using hyperparameter optimization on the VALIDATION DATA.

V. EXPERIMENTS

This section describes our research questions, their motivation and the results we obtained.

A. Are Docstrings Natural?

Python docstrings contain a combination of natural language text and source code, software developers mention identifier names and include snippets of code. Moreover, docstrings typically adhere to one of several formatting style guides, therefore, follow an inherent structure and include specific keywords and syntax elements. As a result, docstrings although similar, are different to natural language text and source code.

Natural language is, by definition, natural and previous work has shown that software is natural as well [22]. However, it is not clear if docstrings follow the same regularities and can be successfully learned using neural language models. To verify this assumption we formulate the first research question:

RQ1: Are docstring comments natural? Do they show similar repetitiveness and predictability characteristics as natural language and source code?

To answer this question, we take inspiration from Hindle et al. [22], who trained n-gram models on English text from the Brown and Gutenberg corpora and on Java source code and compared the obtained cross-entropy values. We train the Sequential Model on the same natural language corpus as [22] and on our docstring TRAINING DATA. In both cases we perform five fold cross-validation and repeat each training three times with different seeds. We report the average cross-entropy and accuracy for both models in Table IV, which also includes the number of training instances and the vocabulary sizes for comparison. The natural language corpus is preprocessed the same way as the docstrings.

The obtained results are higher for docstrings than for English text. This is similar to the conclusion of Hindle et al. [22], although their results are obtained on Java source code without comments. We can also conjecture that source code comments are similar to other types of technical English text, like StackOverflow posts, therefore they might show similar repetitiveness characteristics as found by Rahman et. al [41]. Software, whether source code or comments, seems to be more regular than natural language text. We can conclude that: *Python docstrings are characterized by a high level of predictability that can be captured and exploited using neural language models.*

B. Can Additional Context Improve Predictions?

A language model learns to predict the next word given a prefix sequence, regardless of the type of the model (n-gram or neural based). However, when composing a docstring, it is likely that the developer has written at least part of the code. In particular, in Python, documentation strings follow the method signature, therefore we assume that the developer has included the signature before documenting the method.

TABLE IV
COMPARISON BETWEEN LANGUAGE MODELS TRAINED ON NATURAL LANGUAGE AND PYTHON DOCSTRING SEQUENCES.

Dataset	Size		Cross-Entropy	Accuracy
	Instances	Vocabulary		
Brown & Gutenberg	2,630,560	28,459	6.3362	0.1586
Python Docstrings	4,100,228	30,456	4.6873	0.2483

TABLE V
TOP-K ACCURACY VALUES FOR LANGUAGE MODELS TRAINED ON PYTHON DOCSTRINGS SEQUENCES.

Model	Vocab	Top-1	Top-3	Top-5	Top-10
Sequential Model	30,456	0.248	0.376	0.437	0.521
Context Model (signature)	33,842	0.261	0.395	0.458	0.545
Context Model (full body)	51,822	0.275	0.407	0.469	0.552

On the other hand, a developer might prefer to comment a method after finishing its implementation, which can contain important context information. As a result of these use cases, we consider both the signature and the full body as important context information that a neural language model could leverage to improve its completion suggestions. The next research question investigates this intuition:

RQ2: Will context information improve the predictions of a neural language model trained on Python docstrings?

To answer this question we train and evaluate three models. As a baseline, we train a Sequential Model on the TRAININIG DATA that only receives the prefix sequence as input. To cover both aforementioned cases, we train two Context Models, the first one uses only the signature as context information and the second one the full method body. We report the average top-k accuracies for all models in Table V. From these, we conclude that *including context information does, indeed, improve the prediction accuracy of the model, and using more information –the full method body– does lead to better results.*

The reported values are computed taking into account all predictions provided by the model. But a neural language model returns a probability for each word in the vocabulary, representing the likelihood that this word follows the prefix sequence. The confidence of the model, the probability of the predicted words, will vary from case to case. A developer will be annoyed by frequent bad suggestions, hence, to be useful in practice, we would prefer to have a model with high precision, even if the recall is lower and suggestions are less frequent. To investigate how a confidence filter would affect the accuracy of the model, we formulate the following sub-question to our initial research question:

RQ2.1: What is the accuracy of the model if we only consider predictions that pass a specified probability threshold? What is the frequency of suggestions given such a threshold?

To answer this question, we use the Context Model with the full method body and compute the top-k accuracies with probability thresholds in the range $[0.1, 1.0]$. We plot the results

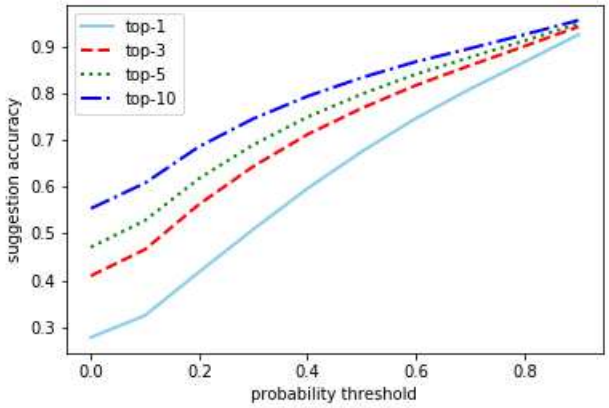

Fig. 4. Effect of probability threshold on suggestion accuracy

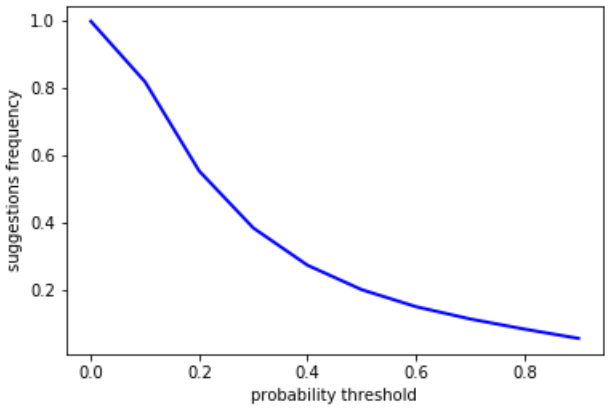

Fig. 5. Effect of probability threshold on suggestion frequency

computed for a model trained on a single fold in Figure 4. The x-axis includes the probability threshold, while the y-axis includes the top-k accuracies. To visualize how a higher probability threshold affects the frequency of predictions, we plot in Figure 5 on the x-axis the probability threshold and on the y-axis the percent of cases for which the model returns predictions with a probability at least as high as the threshold.

We can observe from the plot in Figure 4 that *the accuracy increases as we consider more and more confident predictions*, although this leads to less frequent suggestions. By setting a probability threshold of 0.4, we are able to suggest completions with a top-1 accuracy of more than 50%, and a top-3 accuracy of around 70%. But this means we generate a prediction in only a third of cases.

C. Cross-project Performance

Neural language models are able to learn and exploit the predictability and statistical patterns from our training corpus of Python docstrings. However, it is not clear if this predictability is limited to documentation strings in general, or to specific projects. How would our models fare if we apply them on completely new projects that were not part of the TRAINING DATA? To investigate this issue we formulate our final research question:

RQ3: How well can neural language models, trained on Python docstrings, generate completion suggestions for projects not seen during training?

To answer this research question we queried GitHub for a second time for the top-1000 most started Python projects in April 2019, then selected all the projects that were not part of our original TRAINING DATA with at least 100 method docstrings. We randomly sampled ten projects from this set, extracted the method docstrings and corresponding bodies, preprocessed them and obtained sequences of length five, including the prefix and target word pairs as described in Section IV-B. Finally, we compute the top-k accuracies of the Context Model with the full method body on this data set, separately for each project. The top-k accuracies for a single project are averaged using the models trained on the five folds and with the three repetitions. The results are included in Figure 6 using boxplots to understand the variability and dispersion of the data. On the x-axis we indicate the type of the accuracy, while the y-axis includes the accuracy values.

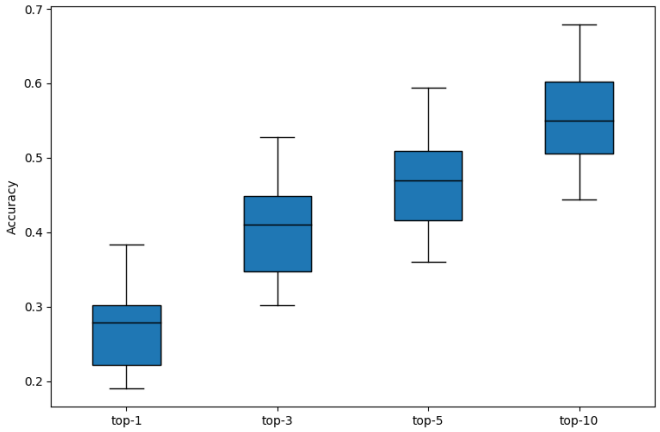

Fig. 6. Top-k accuracy for unseen projects.

From the plot we notice the obtained values are quite similar to the ones reported in Table V for the Context Model. For some projects the model is even able to make better predictions than in the original evaluation. We investigate the generalizability of the model on a small set of ten projects, but these preliminary results indicate that *the Context Model with full method body can be used successfully on new projects that are not part of the* TRAININIG DATA.

VI. DISCUSSION AND THREATS TO VALIDITY

In this section we perform a preliminary qualitative study by manually analysing multiple input sequences, the corresponding target words, and the predictions of the language models. We also discuss our findings, the challenges we faced, and where we see potential for improvement.

TABLE VI
EXAMPLE INPUT SEQUENCES WITH CORRECT PREDICTIONS (THE GREY PART OF THE DOCSTRINGS IS PROVIDED AS CONTEXT INFORMATION TO THE READER, BUT IS NOT PART OF THE INPUT, THE EXPECTED TARGET WORD IS HIGHLIGHTED IN BOLD.)

Nr.	Partial Docstring	Partial Context	Input Sequence	Top-10 Predictions
1	service to respond before raising a :class: ‘geopy.exc.GeocoderTimedOut‘ **«exception»**.	geocode self query exactly one timeout default sentinel params self format string query self api key params self api key	<class> geopy exc geocodertimedout	**«exception»**, exc, error, exceptions, message, item, if, valueerror, empty, instance
2	Create, populate and return the VersioneerConfig() **«object»**.	get config cfg versioneer config cfg vcs cfg style cfg tag prefix cfg parentdir prefix cfg versionfile source cfg verbose	and return the versioneerconfig	**«object»**, instance, class, objects, string, dict, code, type, array, attribute
3	calculate quantiles of Monte **«Carlo»** results	quantiles self idx frac self mcres ndim idx mcres self mcres idx value error mcres self mcres self frac frac	calculate quantiles of monte	**«carlo»**, updates, operations, testing, check, operation, checks, statistics, tests, records
4	Useful to bypass very weak and bespoke web application **«firewalls»** that filter	tamper payload kwargs payload payload find payload find index payload find depth comma end i xrange index len len payload	and bespoke web application	**«firewalls»**, with, you, uses, using, application, n_samples, can, that, data
5	Tiny setting with a recurrent next-frame **«model»**.	registry register hparams rlmb tiny recurrent hparams rlmb ppo tiny hparams epochs hparams generative model hparams generative model params hparams	with a recurrent next-frame	**«model»**, dataset, models, layer, point, object, context, instance, field, state

TABLE VII
EXAMPLE QUERIES WITH INCORRECT PREDICTIONS

Nr.	Partial Docstring	Partial Context	Input Sequence	Top-10 Predictions
1	statsmodels.data.utils.Dataset instance. This **«objects»** has attributes:	get rdataset dataname package cache data base url package docs base url package cache get cache cache data cache get	utils dataset instance this	is, method, will, function, allows, can, parameter, returns, includes, uses
2	decorator that will run a test with some mongoworker threads in **«flight»**	worker threads n threads dbname n jobs sys maxsize timeout UNK ii threading thread target worker thread fn args ii	some UNK threads in	the, parallel, a, order, this, UNK, <ident0>, an, case, that
3	If a topic is sharded by account_id, the router can send us to the Faust worker responsible for any **«account»**.	cached property router self router t self conf router self	worker responsible for any	handler, requests, logging, request, workflow, exceptions, wsgi, handlers, resources, traceback
4	The message and thread /search endpoints, and the /send endpoint directly **«interact»** with the remote server.	app before request before remote request request endpoint request method g namespace request environ g namespace account provider valid account	the send endpoint directly	to <end>, from, into, and, with, using, or, in, as
5	Generate a random title for name. **«You»** can generate random prefix or suffix for name using this method.	title self gender optional gender title type optional title type str gender key self validate enum gender gender title key	random title for name	<end>, <ident1>, or, <param>, and, of, <ident2>, UNK, in, <ident0>

TABLE VIII
EXAMPLE QUERIES WITH CORRECT CONTEXT MODEL AND INCORRECT SEQUENCE MODEL PREDICTIONS

Nr.	Partial Docstring	Partial Context	Input Sequence	Top-10 Predictions
1	verify if element is present using a custom wait **«time»**	test element present by css using a custom wait time self self browser find by css click self self browser	using a custom wait	**«time»**, call, for, value, method, page, <end>, parameter, line, control
2	return: Repo instance initialized from the **«repository»** at our submodule path	module self module checkout abspath self abspath repo git repo module checkout abspath repo self repo repo invalid	instance initialized from the	**«repository»**, <ident1>, given, <ident0>, UNK, root, current, <ident2>, url, specified
3	given number of digits (when “fix_len==True“). :param digits: maximum **«number»** of digits	random number self digits fix len digits digits self random digit fix len self generator random randint pow digits pow	true <param> <ident1> maximum	**«number»**, value, <ident0>, length, precision, <ident1>, decimal, random, string, allowed
4	Any additional keyword arguments will be passed to Elasticsearch.**«indices»**.shard_stores unchanged.	shard stores self kwargs self connection indices shard stores index self name kwargs	be passed to elasticsearch	**«indices»**, elasticsearch, arg, index, cluster, indexes, <ident1>, UNK, with, operations
5	Whether score_func takes a continuous decision certainty. This only **«works»** for binary classification.	make scorer name score func optimum greater better needs proba needs threshold	decision UNK this only	**«works»**, applies, makes, happens, used, supports, affects, is, uses, accepts

A. Qualitative Analysis of the Results

It is generally difficult to understand the predictions of a neural network due to the large number of parameters in the order of millions or more. Nevertheless, by manually analyzing the returned predictions for a limited number of examples, we can gain an intuition of what the model is doing. To achieve this, we first focused on the best possible case: the model is correct and very confident. To find such examples, we first saved the predictions of the Context Model with full method body for all the training instances in the TEST DATA. Then we selected all the correct top-1 predictions and sorted them in decreasing order of the probability assigned by the model to the target word. We then randomly sampled 200 examples from the first $1,000$ instances and analyzed them manually; we include five such examples in Table VI. For these cases, the model assigns very high probabilities, close to 1.0, so we expected that these results will mainly include words that are very common in the training corpus. We noticed that, indeed, this is true and that the model is very confident when predicting common tokens like the string "com", "org", "edu" at the end of a web address, "<end>", which represents the end of a docstring, or several stop words. Nevertheless, it is also able to return more meaningful tokens like the ones included in Table VI. For the first example, the model correctly predicts the word "exception", interestingly several words from the top-10 results would be valid completions ("exc", "error", "exceptions"). For the second example the model recognizes that a constructor call is followed by the token "object". The word "Monte" is usually followed by "Carlo", this is the name of a known algorithm, and probably the TRAINING DATA contained enough occurrences of these tokens to enable the model to output a correct prediction. For the fourth case, the model returns the word "firewalls" as following the sequence "web application". It is interesting that the model correctly learned this association, although the target word is not included in the context information. In the last example, the model predicts that the word "model" comes after the sequence "recurrent next-frame", here it seems that the model is able to leverage the context information to make a correct prediction, since the target word is also contained in the method body.

After examining the positive examples, we analyzed the cases for which the model assigns very low probabilities to the target words. We selected all instances with incorrect top-1 predictions, sorted them in increasing order of the probability allocated to the target word and then randomly sampled 200 such cases from the first 1000 results. Several examples from this set are included in Table VII. The target word for the first row is likely a typo, several of the top-10 predictions are valid completions of the input sequence (e.g. "method", "function", "parameter"). For the following example, the model is not able to correctly output the word "flight", it is likely that associating the word "threads" with this word is not very common in our TRAINING DATA, therefore the model is not able to learn this. However, some of the tokens from the top-10 predictions would also make reasonable completions. For the third example, the target word "account" is not part of the top-10 results, though the input sequence can be completed with several of the predicted tokens, likely the model does not have enough information in the input to return a correct prediction. The input for the fourth example should be completed with the verb "interact", but the model is not able to correctly recognize this. Interestingly, most predictions in the top-10 results do not even include verbs, probably the input sequence is not informative enough. The last example is quite difficult, since the target completion requires to start a new sentence with the pronoun "you" and the prefix sequence is not relevant for generating the suggestion. From the analyzed examples, we noticed that there are instances that are quite difficult to complete, others are unusual thus too rare in the TRAININING DATA for the model to learn them effectively however, often, even if the target word is not present in the top-10 results, several of the suggested tokens represent semantically and syntactically correct completions.

From the examples we analyzed so far it is not obvious if the input context is helpful. To understand in which situations the model can use the context information to improve its predictions, we analyzed a random sample of 200 instances from the TEST DATA, in which the Sequential Model does not return the target word in the top-10 results and the Context Model with the full method body predicts the correct word. Several of these examples are presented in Table VIII, we do not include a column with the predictions of the Sequential Model for reasons of space. For the first example, the model correctly predicts the word "time" as a completion suggestion, which is also part of the context information. In the second example, the model is able to predict the word "repository", although the context information contains the abbreviation "repo". Likely the model learned that these two words are very similar, even if the word "repo" is not included in the top-10 results. In the next example, the model correctly returns the word "number" as the first result, but the next prediction ("value") would also be a good completion. Interestingly, for the fourth case, the model is able to return both plural forms of the word "index", as "indices" and "indexes" in the top-10 results, while the word "indices" is also part of the context information. The model learned that these two words are highly related. We also encountered situations, in which it is not obvious how the input context is helping the Context Model make a correct prediction. One such example is included in the last row of Table VIII. The model is able to correctly predict the word "works", although it contains an UNK token in the input and the target word does not appear in the context information.

In summary, we noticed by manually analyzing multiple predictions of the model on instances of the TEST DATA, that the Context Model is able to return meaningful completion suggestions with high confidence, the mistakes are often caused by unusual target words or ones that are rare in our TRAINING DATA, and, finally, the context information can help the model return better predictions than the Sequence

Model. Future work is necessary to understand how it is able to do that in non-obvious cases.

B. Implications of our Findings

The results of our experiments confirm that we are able to generate meaningful completion suggestion for Python documentation strings. To integrate such a model into an IDE, we would likely need to obtain higher accuracy values, nevertheless, our first results are promising. We believe that documenting code involves more than just summarizing it, and only the developer is able to make the right decision about what to include in a comment and what not. Therefore by accurately generating completion suggestions, we can reduce the time and effort required to write a comment while letting the developer take these decisions. Additionally, we confirmed that using context information (the signature or the full method body) improves the accuracy of the suggestions. However, it will not always be the case that the full method body is available before it is documented. Therefore, we plan to analyze in future work how the performance of the model changes with amount of available context information. We currently only consider the cases which include the signature or the full method body.

Our approach of averaging the embeddings of the context information, compresses and as a result losses part of the information, this has a stronger effect for longer context inputs. We partially mitigated this problem by increasing the embedding size, nevertheless using a different architecture for conditioning predictions on context information might lead to better results. Therefore, we plan as future work to implement and evaluate alternative model architectures. We currently treat the method body as a sequence of words, but source code has a well defined structure and using a representation that takes into account its abstract syntax tree might lead to better results. Investigating alternative ways for preprocessing the context information is a good opportunity for improving our results.

Machine learning models are only as good as the data used for training them, and neural language models are known to be data hungry. The current state of the art results for language models were obtained using a dataset of 40 GB of Internet text data [40]. Source code will never be as abundant as natural language text, additionally it is difficult to find good quality code that is well commented. We ensured the quality of our training data by only selecting projects with a high star rating on GitHub with enough docstring comments. Nonetheless, we could also observe the duplication effect that was described by Allamanis [7]. To mitigate this issue, we eliminated all cases of exact duplication. We will investigate near-duplication in future work, because this might still be a threat to the validity of our work. As mentioned before, Python projects are less abundant than English text, so improving our models will likely require more data. An interesting alternative might be using different data sources. Comments contain a combination of source code and natural language, therefore StackOverflow posts might be a viable option.

A current problem of word-based language models is how to treat out-of-vocabulary (OOV) words. Introducing new identifier names is very common in source code, consequently, code suffers from this problem at a higher rate than natural language text. We partially handle this issue by replacing formal parameter names with numbered symbols, but better solutions are advised. For example, using open-vocabulary models like a subword-units-based approach [27], in which each subword unit is a sequence of characters that occurs as a subsequence of some tokens in the data; the model then returns a sequence of subword units instead of full words [43]. Another way to handle OOV issues is using pointer networks [47] that employ a copy mechanism to either predict a token that appeared in a previous context, even if this was not present in the training data, or one that is part of the vocabulary. We will investigate the OVV problem in the future.

Python docstrings follow formatting guidelines and have a structure. We currently treat docstrings as a sequence of tokens, but we could take this structure into account to improve our models. For example, docstrings start with a summary and a detailed description, followed by an introduction of function arguments and return values. These two parts will likely use different kind of words and providing the location as information to the model has the potential to further improve the accuracy of completion suggestions. Docstrings contain identifier names, snippets of code, and even examples of interactive sessions called doctests [2]. We currently eliminate doctests from docstrings, but we might be able to offer completion suggestions for doctests in the future by using models trained specifically on them.

Finally, in this work we focused on generating completion suggestions for Python method and function docstrings. We still need to verify if our approach can be extend to other types of docstrings (e.g. class, variable). Additionally, developers can benefit from support when writing inline comments as well. In this case we need to investigate what kind of context information could be used. As future work, we plan on extending our approach to other types of comments.

VII. Summary

Good comments do not only describe the source code, but also provide additional information such as justifying a design or implementation decision. This task cannot be fully automated and the developer is needed to decide what kind of content to include. To improve the productivity and reduce the required time to write comments, we have introduced the novel idea to support comment writing through a semi-automated approach and our results show the feasibility of such a comment completion system. Based on neural language models trained on a large corpus of Python docstrings, we show that these comments are indeed natural and can be predicted. Our findings also confirm that context information (the method signature or full body) can further improve the prediction quality. These promising first results show the potential of a comment suggestion engine and open the door for future work in this area.

REFERENCES

[1] Docstrings in python. https://www.datacamp.com/community/tutorials/docstrings-python. Accessed: 2019-03-27.

[2] doctest - test interactive python examples. https://docs.python.org/2/library/doctest.html. Accessed: 09-05-2019.

[3] Pep 257 – docstring conventions. https://www.python.org/dev/peps/pep-0257/. Accessed: 2019-03-27.

[4] Python - abstract syntax trees. https://docs.python.org/3/library/ast.html. Accessed: 09-05-2019.

[5] Python - tokenize. https://docs.python.org/3/library/tokenize.html. Accessed: 09-05-2019.

[6] Writing system software: code comments. http://antirez.com/news/124. Accessed: 2019-03-06.

[7] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. *CoRR*, abs/1812.06469, 2018.

[8] Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[9] Mia Xu Chen, Benjamin N. Lee, Gagan Bansal, Yuan Cao, Shuyuan Zhang, Justin Lu, Jackie Tsay, Yinan Wang, Andrew M. Dai, Zhifeng Chen, Timothy Sohn, and Yonghui Wu. Gmail smart compose: Real-time assisted writing. *CoRR*, abs/1906.00080, 2019.

[10] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *EMNLP*, pages 1724–1734. ACL, 2014.

[11] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

[12] Adelina Ciurumelea. Data for "suggesting comment completions for python using neural language models". https://figshare.com/articles/Data_for_Suggesting_Comment_Completions_for_Python_Using_Neural_Language_Models_/11379120, 2019.

[13] Adelina Ciurumelea. Source code for "suggesting comment completions for python using neural language models". https://figshare.com/articles/Source_code_for_Suggesting_Comment_Completions_for_Python_Using_Neural_Language_Models_/11406765, 2019.

[14] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[15] Yoav Goldberg and Graeme Hirst. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017.

[16] Emitza Guzman, David Azócar, and Yang Li. Sentiment analysis of commit comments in github: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 352–355, New York, NY, USA, 2014. ACM.

[17] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*, pages 35–44, Oct 2010.

[18] D. Haouari, H. Sahraoui, and P. Langlais. How good is your comment? a study of comments in java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 137–146, Sep. 2011.

[19] Hao He. Understanding source code comments at large-scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 1217–1219, New York, NY, USA, 2019. ACM.

[20] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA, 2018. ACM.

[21] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. Will they like this?: Evaluating code contributions with language models. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 157–167, Piscataway, NJ, USA, 2015. IEEE Press.

[22] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[24] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 200–210, New York, NY, USA, 2018. ACM.

[25] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics.

[26] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[27] Rafael-Michael Karampatsis and Charles A. Sutton. Maybe deep neural networks are the best choice for modeling source code. *CoRR*, abs/1903.05734, 2019.

[28] Jack Lanchantin and Ji Gao. Exploring the naturalness of buggy code with recurrent neural networks. *CoRR*, abs/1803.08793, 2018.

[29] Yuding Liang and Kenny Q. Zhu. Automatic generation of text descriptive comments for code blocks. *CoRR*, abs/1808.06880, 2018.

[30] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[31] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[32] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In Takao Kobayashi, Keikichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH*, pages 1045–1048. ISCA, 2010.

[33] Tomas Mikolov, Stefan Kombrink, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *ICASSP*, pages 5528–5531. IEEE, 2011.

[34] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32, May 2013.

[35] Dana Movshovitz-Attias and William W. Cohen. Natural language models for predicting programming comments. In *ACL*, 2013.

[36] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584, Nov 2015.

[37] John Ousterhout. *A Philosophy of Software Design*. Yaknyam, 2018.

[38] Rohan Padhye, Senthil Mani, and Vibha Singhal Sinha. A study of external community contribution to open-source projects on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 332–335, New York, NY, USA, 2014. ACM.

[39] Luca Pascarella and Alberto Bacchelli. Classifying code comments in java open-source software systems. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 227–237, Piscataway, NJ, USA, 2017. IEEE Press.

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 2018.

[41] Musfiqur Rahman, Dharani Palani, and Peter C. Rigby. Natural software revisited. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 37–48, Piscataway, NJ, USA, 2019. IEEE Press.

[42] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM.

[43] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *CoRR*, abs/1508.07909, 2015.

[44] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[45] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 83–92, May 2013.

[46] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[47] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 2692–2700, Cambridge, MA, USA, 2015. MIT Press.