Where should I comment my code? A dataset and model for predicting locations that need comments

Annie Louis
alouis@inf.ed.ac.uk
University of Edinburgh*

Santanu Kumar Dash
s.dash@surrey.ac.uk
University of Surrey

Earl T. Barr
e.barr@ucl.ac.uk
University College London

Michael D. Ernst
mernst@cs.washington.edu
University of Washington

Charles Sutton
charlessutton@google.com
Google Research

ABSTRACT

Programmers should write code comments, but not on every line of code. We have created a machine learning model that suggests locations where a programmer should write a code comment. We trained it on existing commented code to learn locations that are chosen by developers. Once trained, the model can predict locations in new code. Our models achieved precision of 74% and recall of 13% in identifying comment-worthy locations. This first success opens the door to future work, both in the new *where-to-comment* problem and in guiding comment generation. Our code and data is available at http://groups.inf.ed.ac.uk/cup/comment-locator/.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; • **Computing methodologies** → *Neural networks*; *Natural language processing*.

KEYWORDS

NLP, natural language processing, comments

ACM Reference Format:
Annie Louis, Santanu Kumar Dash, Earl T. Barr, Michael D. Ernst, and Charles Sutton. 2020. Where should I comment my code? A dataset and model for predicting locations that need comments. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381736

1 INTRODUCTION

Code comments are essential. With too *few* comments, software is difficult to maintain and change. Too *many* comments are also harmful: they may not contribute to understanding the code, they can reduce readability, they are more likely to fall out of date with the code, and creating them wastes the most valuable resource, which is developer time. Thus, it is important to write comments in the most useful locations.

Although a common suggestion is to write comments at the same time as the code, many programs lack sufficient comments. Therefore, developers frequently return to code to add comments: when preparing for a code review, when coming up to speed on a codebase, when answering a colleague's questions, when performing refactoring, etc. Currently, developers must use their judgment in deciding where to write explanatory code comments.

We propose a new research problem: identifying *where to write comments*. Such a facility could guide developers to write comments in the most useful locations, without wasting time elsewhere. Our tool does not suggest comment text, but is a first step towards the larger research goal of automatically writing comments.

Our machine learning approach identifies comment locations from simple cues present within the source code, and is trained on the comment locations from existing code. So it is important that the code in the model's training set be high quality and well commented. For this reason, we trained our model on a corpus of commented C code from the Android code base, so that the training set is a model of good practice.

When applied to uncommented code, the model suggests locations where comments could be written. Our best approach is based on neural networks, specifically a simple extension to the standard recurrent neural network, which we call a *hierarchical sequence model*, that computes a representation of a snippet of contiguous lines of code, rather than representing each line of code in isolation. Our results show that even a technique that shallowly captures the content of code reaches a precision of 74% and recall of 13% in identifying comment-worthy locations. A good comment-suggestion tool must ensure precise suggestions and guard against too many false positives, so our initial results are in the right direction.

In summary, we make three contributions:

- We propose the where-to-comment task: a useful problem that has not been studied before.
- We present a corpus of C code where comment locations are identified.
- We developed a machine learning solution. Our results are promising, but they also point out the immense potential for further work to improve techniques for solving the where-to-comment problem.

2 RELATED WORK

We have proposed a new task: where to write comments. There is no previous work on this topic. There is, however, a growing body

* now works at Google.

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05.
https://doi.org/10.1145/3377816.3381736

of work on generating comments. Pioneering work started with rules or templates [2], before work moved to comparing n-gram language models to LDA models for generating class comments [11]. Representative recent work translates code to comments via deep learning for method header comments [6]. This generation task must produce comments at particular locations, which means it must implicitly solve our where-to-comment task. Because the where-to-comment problem is hard, comment generation work has, to date, either commented its input code snippet as a whole [6] or it has produced a comment at every instance of a specific kind of comment location, like statement, class, or method headers. Most work falls into the latter category, e.g., [11]. Learning where to comment is an essential first step to generation of high-quality comments.

There is also work on translating comments into assertions (comments to code) [1, 3, 5, 10] and/or detecting inconsistency between comments and code [12–14]. Such research assumes that comments already exist. Our work can be seen as a degenerate case of code-comment inconsistency where the inconsistency is between the code and lack of a comment.

In contrast to our goal of suggesting where to add comments, CRAIC [7] is a system that identifies where to *remove* redundant comments.

3 DATASET OF COMMENTED CODE

To train and evaluate models for the where-to-comment problem, we curated a dataset of historical commenting behavior. Our dataset is based on the idea of *snippets*: code blocks delimited by empty lines. An example is shown in Listing 1. Using blank lines to create logical groupings is a standard coding practice recommended by the official Python style guide (PEP 8), Oracle's Java code conventions, Google's style guides (for C++ (which encourages a blank line before comments), Java, Python, and more), McConnell's *Code Complete* [8], etc. We propose predicting where-to-comment at the level of snippets, rather than individual lines of code. By doing so, we aim to make the prediction problem easier for an automatic method — because snippets provide more information to the method than single lines — while still providing a location that is specific enough to a developer.

We created a dataset of 41,506 snippets of C/C++ source code. Each snippet was delimited by blank lines in the original program, All comments have been removed. Each snippet is annotated by a Boolean label indicating whether a programmer wrote a comment within the snippet.

3.1 Source Code

Our dataset comes from 601 files from 9 C/C++ libraries in the native substrate of the Android Open Source Project (AOSP): boringssl, libjpeg-turbo, libmpeg2, libpcap, libpng, netcat, netperf, tcpdump, and zlib. These libraries' domains include cryptography, graphics, network, and codecs. This codebase is professionally developed, and we consider it to demonstrate good commenting practice.

A threat to validity is the possibility that the AOSP developers wrote too few or too many comments. Extra comments are a serious concern: they would reduce a trained model's precision and cause the model to make useless suggestions to programmers. Missing comments in AOSP would only reduce recall, but the model would still be useful to programmers.

Listing 1: Two snippets with comments. It is part of an implementation of Huffman coding. Taken from `jchuff.c` in the `libjpeg-turbo` project of the Android Open Source Project.

```
193
194  /* Find the input Huffman table */
195  if (tblno < 0 || tblno >= NUM_HUFF_TBLS)
196    ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, tblno);
197  htbl =
198    isDC ? cinfo->dc_huff_tbl_ptrs[tblno] :
           cinfo->ac_huff_tbl_ptrs[tblno];
199  if (htbl == NULL)
200    ERREXIT1(cinfo, JERR_NO_HUFF_TABLE, tblno);
201
202  /* Allocate a workspace if we haven't already done so. */
203  if (*pdtbl == NULL)
204    *pdtbl = (c_derived_tbl *)
205      (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo,
           JPOOL_IMAGE, SIZEOF(c_derived_tbl));
206  dtbl = *pdtbl;
```

3.2 Snippets and Comment Locations

To create our dataset, we divided the source code into snippets, labeled each snippet depending on whether it contains a comment, and removed all comments from the snippets.

Segmenting source code into snippets. A comment may explain multiple lines of code. Our model does not merely indicate a single line of code (LOC) as needing a comment. Rather, we segment the source code into snippets, and predict whether a comment is needed somewhere within each snippet.

A *snippet* is a contiguous sequence of non-blank lines in the source code. It is intended to capture a logical unit of related code, as indicated by the programmer who wrote blank lines between snippets. We hypothesize that these snippets are semantically or syntactically meaningful spans.

Our tool creates snippets as follows. (1) Remove blank lines between a comment and subsequent code (such as lines 193 and 201 in Listing 1). We assume that a comment is documenting the immediately following code except for those on the same line as a LOC. (2) Split the code into segments at blank lines. The resulting snippets have a mean length of around 6 LOC and median length of 4 LOC, but some have hundreds of LOC. (3) If a snippet is more than 30 lines long (only 2% of snippets), we split it into n separate segments, the first $n-1$ of which are exactly 30 lines long. This split is arbitrary, but it keeps suggestions to programmers focused.[1]

Labeling. We attach a *snippet-label* of 1 (*comment location*) to any snippet containing a comment. The label does not depend on whether the comment is on the same line as source code or on a line of its own. The label also does not depend on how many comments are contained within the snippet, only on whether any comment

[1]We also tried other ways of dealing with long segments—truncate to length 30 or remove long segments altogether from the training and test sets. We found these configurations to give similar results as the one with maximum length of 30.

data	total snippets	commented snippets anywhere	commented snippets at top
train	36,277	10,904 (30%)	7,330 (20%)
valid	2,436	614 (25%)	386 (16%)
test	2,793	711 (25%)	417 (15%)

Table 1: Dataset Statistics: the number of snippets that are commented and, how many have a comment at the top of the snippet.

is present. All other snippets are given a label of 0 (*not a comment location*).[2]

Our goal is to predict these binary labels on the snippets. Those predicted with a label of 1 can be presented to the developer as possible locations to be commented.

While our goal is to predict snippet labels, some of our models are trained with the additional signal of *LOC-labels*. We obtain these labels by dividing the file into single LOCs (snippets of size 1). The LOC is given a label of 1 if there is an end-of-line comment on the same line or a whole-line comment just before it. These labels are called LOC-labels.

Remove comments. Before using the snippets for training, we removed all comments and any blank lines that would be caused by the removal of a comment.

Dataset Statistics. We randomly divided our dataset into 501 files for training, 50 validation, and 50 test (Table 1).

Around 25–30% of the snippets are commented. About two thirds of these have a comment before the first line of the snippet. This high percentage of snippets beginning with comments indicates that many snippets may correspond to semantically meaningful segments of code.

4 TECHNIQUE

Our goal is to produce a binary label for each snippet indicating whether it should be commented or not. The comment might be needed on the snippet as a whole or on some LOC within it. We evaluated several models for this task.

4.1 LOC Model

The "LOC model" treats each line of code as independent from others. This model has no information about surrounding LOCs nor the boundaries of the snippet.

The model is trained on only the LOC labels, and predicts whether individual LOCs should be commented. To produce a snippet-label prediction, the predictions for the LOCs within a snippet are combined by logical OR.

The model is a multi-layer perceptron model that takes a LOC as input and produces a binary label at the output layer. The input LOC is represented by the average of embeddings for each token. These embeddings are obtained using word2vec [9] on about 10M lines of C code (different from our training/test data) In all the following models as well, the input LOC is represented in the same manner.

[2]We did not filter out any comments, but those such as TODOs occur rarely (less than 1%).

4.2 Sequence Model

The "sequence model" also uses per-line labels, but the label produced for a LOC is conditioned on the content of all the previous LOCs in the file.

Here a recurrent neural network (RNN), with LSTM cells, is run from the start of a code file to its end. Each step in the RNN consumes one LOC and produces a state which summarizes the LOCs encountered thus far in the sequence. This summary state obtained after each LOC is used to predict a binary label which indicates whether a comment appeared before (or on) this LOC. Similar to the LOC model, the LOC-labels are used for training.

After training, the per-LOC predictions are combined with the logical OR operation to produce the prediction for a snippet. We call this model "seq-OR".

4.3 Hierarchical Sequence Models

Here we have a RNN at the snippet level from the start of a file to its end. Each step of the network encodes one snippet and summarizes the snippets so far. Each snippet in turn is represented by another RNN which sequentially encodes the LOCs within that snippet. The states produced by this LOC-level RNN are combined by max pooling to produce the snippet representation. This model ("hier") is trained on snippet level labels.

A second model ("hier-multi-task") is similar to the previous except that it also takes advantage of LOC-labels. We add a fine-grained loss which characterizes how well the model can predict LOC level labels. The total loss is a weighted average of the snippet level and the fine-grained loss:

$$Model\_loss = \alpha \cdot snippet\_level\_loss + (1 - \alpha) \cdot LOC\_level\_loss$$

where $0 \leq \alpha \leq 1.0$.

We trained this model using a curriculum learning approach. We first train with fine-grained loss only. When the loss does not improve further on the validation set, we switch to the two-loss training. We tuned α on the validation data.

5 EVALUATION

We evaluated our models using precision and recall. Precision is the fraction of snippets predicted as comment locations that are truly commented. Recall is the fraction of truly commented snippets that the model predicts. Some of our models use LOC-labels, but only as learning signals. Our goal is prediction of snippet labels only.

Presenting low-confidence suggestions would waste the developer's time and lose trust. So while tuning the hyperparameters of our model on the validation set, we pick the highest-precision model, among those that have at least 20% recall. Obtaining high recall is not particularly important for this task: so long as the model predicts useful locations, it is beneficial even if it does not identify every location.

All our models use a vocabulary size of 5000 consisting of the most frequent tokens (appearing least 10 times on the training data). The hyperparameters of each model are tuned on the validation set. We tune the learning rate, dropout, number of hidden layers, and units. For multi-task learning, we also tune the α parameter.

Table 2 presents our results. The table also reports validation results for reference in future work and to demonstrate how we chose

Model	Validation		Test	
	Precision	Recall	Precision	Recall
LOC-OR	52	39	51	34
seq-OR	88	30	72	9
Hier	85	22	**74**	**13**
Hier-multi-task	78	27	61	17

Table 2: Performance of our models on comment prediction for snippets.

our best models during tuning. But it does not reflect our expected performance on new data. So we show results on a completely blind test set, meaning that we never perused or used these files until we performed all model development, tuning, and selection of the best settings.

Our experiments allow us to test a couple of hypotheses behind our choice of models. Firstly, we wanted to see if a continuous representation of a full snippet is better than having representations at LOC level. This question can be answered by comparing the "hier" models (which compose snippet representations from LOCs) against the "seq" and "LOC" models (these two do not have a notion of snippet during training). During validation, "seq" has best precision. But "hier" gives the top test performance.

Second, we introduced the "hier-multi-task" to understand if some training signals at the LOC level can still help training for snippet labels. The α parameters in the best models always turned out close to 1.0, indicating that the model ignores the LOC signals. Although the LOC signals could help to pretrain the models (find good weights which can be further optimized), the final weights do not seem to depend much on these LOC signals. Hence the results for "hier" and "hier-multi-task" are fairly similar during validation. On new data, the "hier" model is better.

Based on both accuracy and the model's generalization ability on unseen data, "hier" emerges as the best choice.

Note that we specifically chose models with high precision to avoid incorrect suggestions. Currently, these models have low recall. To improve upon our recall (or the number of suggestions to the developer), we could leverage other sources of information, such as a developer's surprise upon seeing certain constructs, or her background knowledge that indicates some parts as needing comment (or otherwise). Our initial efforts to incorporate background knowledge and 'surprise'-capturing features into the models were not successful. These are interesting directions for future work.

6 FUTURE WORK

Future work[3] could automatically segment the code into snippets rather than using our blank-line heuristic. Such a model would predict not only the comment location, but also which span of code should be commented at that location.

Just as fault localization is essential to automated program repair, predicting comment locations is essential to effective comment generation. Future work could evaluate ways to combine the techniques, as well as other modelling improvements to increase performance.

[3]We have released our dataset and code:http://groups.inf.ed.ac.uk/cup/comment-locator/

It would be interesting to evaluate whether code that needs comments is worse code, or code that requires refactoring. (An extracted method's name acts as a kind of comment [4].) Are our models also code smell detectors?

Acknowledgments This work was partly supported by the Engineering and Physical Sciences Research Council [grant number EP/P005314/1] and NSF grant CCF-1836813.

REFERENCES

[1] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. 2018. Translating code comments to procedure specifications. In *ISSTA*. 242–253.
[2] R. P. Buse and W. R. Weimer. 2010. Automatically documenting program changes. In *ASE*. 33âĂŞ42.
[3] M. D. Ernst. 2017. Natural language is a programming language: Applying natural language processing to software development. In *SNAPL*. 4:1–4:14.
[4] M. Fowler. 2000. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
[5] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *ISSTA*. 213–224.
[6] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. 2018. Deep code comment generation. In *ICPC*. 200–210.
[7] A. Louis, S. K. Dash, E. T. Barr, and C. Sutton. 2018. Deep Learning to Detect Redundant Method Comments. http://arxiv.org/abs/1806.04616.
[8] S. McConnell. 2004. *Code complete: A practical handbook of software construction* (2nd ed.). Microsoft Press, Redmond, WA, USA.
[9] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*. 3111–3119.
[10] M. Motwani and Y. Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *ICSE*. Montreal, Canada, 188–199.
[11] D. Movshovitz-Attias and W. W. Cohen. 2013. Natural language models for predicting programming comments. In *ACL*. 35–40.
[12] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. 2007. /*iComment: Bugs or Bad Comments?*/. In *SOSP*. 145–158.
[13] L. Tan, Y. Zhou, and Y. Padioleau. 2011. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE*. 11–20.
[14] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *ICST*. Montreal, Canada, 260–269.