Content Aware Source Code Change Description Generation

Pablo Loyola[1*], **Edison Marrese-Taylor**[2*], **Jorge A. Balazs**[2]
Yutaka Matsuo[2] and **Fumiko Satoh**[1]
IBM Research, Tokyo, Japan[1]
{e57095, sfumiko}@jp.ibm.com
Graduate School of Engineering, The University of Tokyo, Japan[2]
{emarrese, jorge, matsuo}@weblab.t-utokyo.ac.jp
[*]Authors contributed equally to this work.

Abstract

We propose to study the generation of descriptions from source code changes by integrating the messages included on code commits and the intra-code documentation inside the source in the form of docstrings. Our hypothesis is that although both types of descriptions are not directly aligned in semantic terms —one explaining a change and the other the actual functionality of the code being modified— there could be certain common ground that is useful for the generation. To this end, we propose an architecture that uses the source code-docstring relationship to guide the description generation. We discuss the results of the approach comparing against a baseline based on a sequence-to-sequence model, using standard automatic natural language generation metrics as well as with a human study, thus offering a comprehensive view of the feasibility of the approach.

1 Introduction

Transferring the semantics from source code to natural language and vice-versa is at the core of several machine learning endeavors, as it could enable a direct communication between man and machine, improving the level of interpretability and comprehension between each other and easing their collaboration.

In that sense, source code can be conceived as an actual medium of communication from two perspectives, which have been explored separately in both computational linguistics and software engineering communities (Allamanis et al., 2017).

In the first place, from a *developer-program* perspective, source code encodes, in a set of human readable instructions, the requirements a developer commands a program to satisfy. This view has been operationalized as a machine translation problem, trying to learn efficient transitions between the dependencies that words and source code tokens exhibit. With this, recent approaches have been able to summarize source code snippets (Allamanis et al., 2016) or even synthesize natural language instructions into actual commands (Oda et al., 2015; Yin and Neubig, 2017).

In the second place, from a *developer-developer* perspective, the collaborative nature of software development has transformed source code into a common ground for human interaction. In that sense, every new code contribution takes into account the previous modifications, allowing developers to communicate indirectly. One of these applications is the generation of descriptions for source code changes (Loyola et al., 2017), which uses the information contained in a code commit – the *diff* representing the changed code and the message the developer provides at submission time – to train an encoder-decoder architecture. This problem has the particularity of containing certain elements of summarization, as most salient characteristics of the code change need to be extracted, and translation, as it is required to generate a natural language description from a code change.

In this work, we consider the generation of descriptions for source changes as a testing task to explore if the perspectives presented above can be integrated into a single learning architecture. That is, we want to learn to generate descriptions from changes exploiting the information in the source code commits, but incorporating the program functionality expressed through the *docstrings* contained within the source code.

Our hypothesis is that, while both perspectives point at different semantic directions, there should be a certain degree of dependency, since in order to perform a change on the code the developer first needs to understand its functionality. Moreover, we consider that integrating these two perspectives could contribute to alleviate the issues that current approaches for generating descriptions from source code change present, such as the hallucination in the generation, where generated descriptions are syntactically correct but that do not keep any semantic relationship with the actual code change, and also the inability of the model to produce descriptions with a relevant amount of detail.

We propose an approach that, given a code change, compresses the information associated to the docstrings within the file being modified and uses it as an additional context when selecting the next word from the output vocabulary. We also reported an exploratory approach that generates a mask to be used at decoding time that considers the inter-perspective distances based on a bilingual embedding.

In addition to integrating change descriptions and source code documentation, we also explore how to represent the code change itself. Previous work on description generation has relied on the output from the *diff* command, which provides a distinction between the portions of the source code that were added and removed. Such data source has been treated just as a sequence of source code tokens, such as in the case of Loyola et al. (2017). In contrast, we explore an architectural variation where we use two encoders to obtain a more expressive signal from the source code perspective, which can lead to a better natural language generation.

We constructed a dataset by merging both change history and docstring data from several real world open source projects to evaluate our approach. We reported the results on standard translation-based metrics as well through a user study using a crowd-sourcing, to get a more qualitative estimation of the performance of the model.

Our results show that, on average, incorporating a signal from the content of the source code file has a positive impact on the performance of the model. We consider these results could open the door to further research that considers the generation of descriptions from software artifacts from a more systemic perspective. The source code and data for this approach is available at: `https://github.com/epochx/py-commitgen`.

2 Related Work

The emergence of unifying paradigms that explicitly relate programming and natural languages in distributional terms (Hindle et al., 2012) and the availability of large corpus mainly from open source software opened the door for the use of language modeling for several tasks (Raychev et al., 2015). Examples of this are approaches for learning program representations (Mou et al., 2016), bug localization (Huo et al., 2016), API suggestion (Gu et al., 2016) and code completion (Raychev et al., 2014).

Source code summarization has received special attention, ranging from the use of information retrieval techniques to the addition of physiological features such as eye tracking (Rodeghero et al., 2014). In recent years several representation learning approaches have been proposed, such as (Allamanis et al., 2016), where the authors employ a convolutional architecture embedded inside an attention mechanism to learn an efficient mapping between source code tokens and natural language keywords. More recently, Iyer et al. (2016) proposed a encoder-decoder model that learns to summarize from Stackoverflow data, which contains snippets of code along with descriptions.

Both approaches share the use of attention mechanisms (Bahdanau et al., 2014) to overcome the natural disparity between the modalities when finding relevant token alignments. Although we also use an attention mechanism, we differ from them in the sense we are targeting the changes in the code rather than the description of a file.

In terms of specifically working on code change summarization, Cortés-Coy et al. (2014); Linares-Vásquez et al. (2015) propose a method based on a set of rules that considers the type and impact of the changes, and Buse and Weimer (2010) combines summarization with symbolic execution. The use of representation learning based models has been also explored recently, such as the work of Loyola et al. (2017) and Jiang et al. (2017). Both approaches make use of an encoder-decoder architecture, which receives code change, in the form of a *diff* output and the associated message submitted by the contributor.

In terms of ad-hoc datasets, we can mention Zhong et al. (2017) for questions, SQL queries, Oda et al. (2015) for pseudo code in Python, and more recently Barone and Sennrich (2017) for code-docstrings from Python projects on GitHub.

3 Proposed Approach

Our starting point is the code commit, understood as a pair conformed by (i) the differences in the source code obtained as the output of the *diff*[1] command and (ii) the associated message the committer provided to explain the action.

Therefore, for a given software project we can formalize our available data as the set of its T versions $v_1, \ldots, v_T$. Commits are well-defined for every pair of consecutive project versions $\Delta_{t-1}^{t}(v) \rightarrow Commit_t$, so we end up with a total of T commits, each associated to a project version. With this, we model each commit as a tuple (C_t, N_t), where C_t is a representation of the code changes associated to v in time t, and N_t is a representation of its corresponding natural language (NL) accompanying message. Concretely, C_t corresponds to the set of code tokens associated to the commit that was applied to a certain file F_{C_t}, based on the *atomicity* assumption. In principle, we do not assume this set of source code tokens is ordered in a sequential fashion, allowing us to also represent it as a bag of tokens.

Let $\mathcal{C}$ be the set of code changes and $\mathcal{N}$ be the set of all descriptions in NL. We consider a training corpus with T code snippets and message pairs (C_t, N_t), $1 \leq t \leq T$, $C_t \in \mathcal{C}$, $N_t \in \mathcal{N}$. Then, for a given code snippet $C_k \in \mathcal{C}$, our goal is to train a model to produce the most likely description $N^{\star}$.

Following Loyola et al. (2017), we start building our models upon a vanilla encoder-decoder model that at training time receives *(diff, message)* pairs. We use an attention-augmented architecture (Luong et al., 2015) with a bi-directional LSTM as encoder. Let $X_t = x_1, \ldots x_n$ be the embedded input code sequence $C_t = c_1, \ldots, c_n$ as extracted from the *diff*. After feeding these through our encoder, we have a set of vectors $H = h_1, \ldots h_n$ that represent the input. This is later given to the decoder, in our case also an LSTM, such that the probability of a description is modeled as the product of the conditional next-word probabilities, $p(n_i|n_1, \ldots, n_{i-1}) \propto W_c[s_i; a_i]$, where $N_t = n_1, \ldots, n_m$ corresponds to the message tokens, $\propto$ denotes a softmax operation, s_i represents the decoder hidden state and a_i is the contribution from the attention model on the input. W_c is a trainable combination matrix. The decoder repeats the recurrence until a fixed number of words or the special *EOS* token is generated.

[1] http://man7.org/linux/man-pages/man1/diff.1.html

The attention contribution a_i is defined as $a_i = \sum_{j=1}^{n} \alpha_{i,j} \cdot h_j$, where h_j is a hidden state associated to the input and $\alpha_{i,j}$ is a score obtained using the general attention scheme (Luong et al., 2015), $\alpha_{i,j} = \frac{\exp(h_i^{\top} W_a s_i)}{\sum_{j=1}^{n} \exp(h_j^{\top} W_a s_i)}$, where W_a is a trainable scoring matrix.

During training, the decoder iterates until the end-of-sentence token is reached. For generation, we approximate $N^{\star}$ by performing a beam search on the space of all possible summaries using the model output, with a beam size of 10 and a maximum message length equal to the maximum length of the inputs of the dataset.

This model considers a direct transition between *diffs* and messages extracted from source code commits. However, programs usually provide an additional relationship between source code and natural language, in the form of intra-code documentation, commonly known as docstrings.

This documentation appears in multiple locations inside a source code file, usually aligned with a specific line or block, explaining its functionality. The information contained in a *code, docstring* pair is intrinsically local, i.e. the docstring is used as an additional source to support the understanding of a portion of a program beyond the solely internalization of the available source code. Listing 1 presents an example of a real docstring associated to a class from the Pytorch library[2]. In this case we can see that the docstring provides an overall description of the functionality of the class and a summary of the required parameters.

```
class LambdaLR(_LRScheduler):
    """Sets the learning rate of each parameter group to the
      initial lr
    times a given function. When last_epoch=-1, sets initial
      lr as lr.
    Args:
        optimizer (Optimizer): Wrapped optimizer.
        lr_lambda (function or list): A function which
      computes a multiplicative
            factor given an integer parameter epoch, or a
      list of such
            functions, one for each group in optimizer.
      param_groups.
        last_epoch (int): The index of last epoch. Default:
      -1.
    ...
    """
    def __init__(self, optimizer, lr_lambda, last_epoch=-1):
    ...
```

Listing 1: Example of a docstring from a Pytorch module.

If we take a look at the changes committed to this specific class, we can find that most of the

[2] https://github.com/pytorch/pytorch/blob/master/torch/optim/lr_scheduler.py

commit messages associated keep certain relationship with the docstring. For example, a commit[3] from August 8th, 2018 states:

```
    Changed serialization
mechanism of LambdaLR
scheduler
```

Therefore, we are in the presence of two sets of pairs that provide information about the characteristics of a program from two different perspectives. A *(diff, message)* pair set that allow us to understand *why* and *how* changes are conducted over a given file, and a *(code, docstring)* that allow us to understand *what* is the functionality of such file.

Our goal is then to integrate both sources, i.e., to study how the local source code - natural language feature representations learned from the *(code, docstring)* pair can be used to support the generation of natural language descriptions from code changes. Our hypothesis is that while the *(diff, message)* and *(code, docstring)* pairs associated to a file are not pointing at the same semantic direction, they should share certain representational components, as both are centered on the information contained on the file: one trying to explain the code itself *(code, docstring)* and the other trying to explain changes on such code *(diff, message)*.

3.1 Content-aware encoder

We noted that the comments contained within a source file are related to the local functionality of its adjacent source code lines or blocks. In contrast, the message associated to a commit is related to the actual action carried out on the given file. Such message, in theory, is indirectly associated to the functionality of the code, i.e. the code was modified in a given way because its previous functional state led triggered in a developer the need to change it.

Motivated by this idea, we propose an augmented encoder that allows us to capture these relations. Again, let $H = h_1, \dots h_n$ be the result of embedding and processing the input content extracted from C_t. We extract the code and associated docstring of the total r lines of file F_{C_t}. With this, we model each code and docstring line as a sequence of tokens $L_k^c = x_1^c, \dots, x_p^c$ and $L_k^d = x_1^d, \dots, x_q^d$, of length p and q respectively. We use BiLSTMs to encode both sequences independently, as follows.

$$\overrightarrow{h}_i^c = \overrightarrow{\text{LSTM}}(x_i^c, \overrightarrow{h}_{i-1}^c) \quad (1)$$

$$\overleftarrow{h}_i^c = \overleftarrow{\text{LSTM}}(x_i^c, \overleftarrow{h}_{i+1}^c) \quad (2)$$

$$\overrightarrow{h}_i^d = \overrightarrow{\text{LSTM}}(x_i^d, \overrightarrow{h}_{i+1}^d) \quad (3)$$

$$\overleftarrow{h}_i^d = \overleftarrow{\text{LSTM}}(x_i^d, \overleftarrow{h}_{i+1}^d) \quad (4)$$

As Figure 1 shows, we concatenate the last hidden state corresponding to each code and docstring vector to obtain a representation for each line $h^k = [\overrightarrow{h}_p^c; \overleftarrow{h}_p^c; \overrightarrow{h}_q^d; \overleftarrow{h}_q^d]$, with $k = 1, \dots, r$.

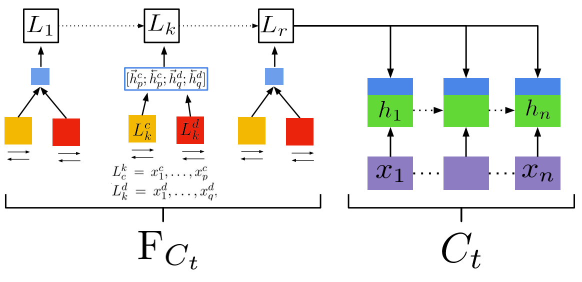
Finally, we use a standard LSTM to model the dependency across the r code/docstring line vectors and take the last hidden state as a means of aggregating and representing the content of both the code and docstring in F_{C_t}. This summarizing vector is concatenated to each h_i coming from the *diff*-level representation. The decoding phase works in a way analogous to the vanilla encoder-decoder model.

3.2 Content-aware decoder

During our feasibility study, we empirically observed that there was a significant overlap between the source code vocabularies coming from the *diffs* and from the code extracted from the files, which in some cases reaches up to 90%.

Our intuition based on such observation is that we can consider both source code vocabularies as a single vocabulary, which is used in two different contexts. In other words, a defined set of source code tokens is conforming a bridge between the messages from the code changes and the docstrings.

To exploit such bridge we explored incorporating the information contained in the *(code, doc-*



L_1
L_k
L_r
$[\overrightarrow{h}_p^c; \overleftarrow{h}_p^c; \overrightarrow{h}_q^d; \overleftarrow{h}_q^d]$
L_k^c
L_k^d
$L_c^k = x_1^c, \dots, x_p^c$
$L_k^d = x_1^d, \dots, x_q^d$
h_1
h_n
x_1
x_n
F_{C_t}
C_t

Figure 1: Diagram of our proposed content-aware encoder. It can be seen how the regular encoder hidden states $h_1, \dots, h_n$ (in green) are augmented using the representation extracted from the content and docstring in F_{C_t} (in blue).

[3] https://bit.ly/2zx4041

string) pairs into the code change description generation by building a mask to guide the decoder. This mask is built upon co-occurrence patterns between message and docstring words, which are used to re-weight the scores the decoder is generating on the output vocabulary. Concretely, during inference our goal was to re-weight the probabilities that are passed to the beam search module.

As stated before, for each project, we have a set of code changes represented as pairs (C_t, N_t), for $C_t \in \mathcal{C}$ the set of *diff* outputs, $N_t \in \mathcal{N}$ the set of messages submitted at commit time. On the other hand, for each commit-derived modified file associated to F_{C_t} $t = 1, \ldots, T$ we have a set of pairs *(code, docstring)*. We train bilingual word embeddings between the set of source code content and their associated docstring lines. The intuition behind this is that the docstring basically explains — or translates to natural language— the functionality of a source code block. While this intuition is arguably not entirely true from a machine translation perspective, at least it allows us to obtain a shared feature space between code and docstring tokens. We adapted the approach by Artetxe et al. (2017), calling the output embedding *C-DC*.

In the second place, we construct another embedding space to combine the set of messages from the commits, and the docstrings, such that this embedding only contains natural language tokens. In this case we do not expect a high overlap between the vocabularies of each set, as they are pointing at different semantic directions —their intent is different. To build this embedding space, we use a standard word2vec (Mikolov et al., 2013) implementation, and call the resulting embedding *M-DC*.

Using the above embedding spaces as mappings, the approach works as follows. Given an input sequence $C_t = c_1, \ldots, c_n$ from a *diff*, each of its tokens c_i used to query the *C-DC* embedding to obtain a set of k neighboring NL tokens K_t. For each NL token in K_t, we obtain its vector representation in *M-DC* and identify a medoid, med_{K_t}. We consider this medoid not only as the representative of K_t, but also indirectly of the associated source code token c_i. We then use this medoid vector and compute its distance to all the elements in the output message vocabulary OV present in *M-DC*, $d_{t,i} = dist(med_{K_t}, i)$ for $i \in OV$. We repeat this process for all the source code tokens from the input *diff* sequence obtaining a distance matrix of $n \times |OV|$.

Finally, we compute the column-wise average distance, obtaining a vector d_s of size $1 \times |OV|$, which represents a compressed association between the input source code sequence C_t and the output vocabulary. This resulting vector is used during inference via a convex combination with the softmax vector l_{dec} output by the decoder $l_{dec} = \alpha * l_{dec} + (1 - \alpha) * d_{C_t}$. The modified vector is passed to the beam search, which selects the next tokens in a regular fashion. It should be noted that as the embedding training operations can be performed off-line, the inference overhead added by our mask is negligible, so there is almost no impact in terms of inference time.

3.3 Structure-aware encoder

Finally, we also explore a different take on the encoding phase. We note that the *diff* associated to a change has an inherent structure that allows us to distinguish the lines that were *added* and/or *removed*. In Loyola et al. (2017), the authors ignored such distinction and simply generated a single sequence by concatenating both added and removed parts. While this approach appears as a simple solution, we consider it limits the expressiveness of this input and introduce issues related to i) loss of the alignment between added and removed code within the source code file, and ii) source code token redundancy.

In order to overcome such issues, we propose to consider the *diff* explicitly as two inputs, one for the added tokens, and one for the removed tokens. We hypothesize that the quality and richness of the encoded input could be improved by comparing these two inputs in order to identify elements that may play a key semantic role in understanding the *diff*, both in terms of added or removed code chunks.

Concretely, let $X_t^+ = x_1^+, \ldots, x_n^+$ and $X_t^- = x_1^-, \ldots, x_m^-$ be the embedded sequences of the concatenated added and removed code lines, as extracted from the *diff* associated to example C_t. We use a single embedding matrix E and the same bidirectional LSTM to encode both sequences.

$$\vec{h}_i^+ = \overrightarrow{\text{LSTM}}(x_i^+, \vec{h}_{i-1}^+) \quad (5)$$

$$\overleftarrow{h}_i^+ = \overleftarrow{\text{LSTM}}(x_i^+, \overleftarrow{h}_{i+1}^+) \quad (6)$$

$$\vec{h}_i^- = \overrightarrow{\text{LSTM}}(x_i^-, \vec{h}_{i-1}^-) \quad (7)$$

$$\overleftarrow{h}_i^- = \overleftarrow{\text{LSTM}}(x_i^-, \overleftarrow{h}_{i+1}^-) \quad (8)$$

We later apply two matching strategies over the resulting vector sequences, which are based on the multi-perspective matching operation by Wang et al. (2017). The operation is built upon a cosine matching function f_m, which is used to compare two vectors, as follows.

$$m = f_m(v_1, v_2; W) \tag{9}$$

where v_1 and v_2 are two d-dimensional vectors, $W \in \Re^{l \times d}$ is a trainable parameter with the shape $l \times d$, l is the number of perspectives, and the returned value m is a l-dimensional vector $m = [m_1, ..., m_k, ..., m_l]$. Each element $m_k \in m$ is a matching value from the k-th perspective, and it is calculated by the cosine similarity between two weighted vectors

$$m_k = cosine(W_k \circ v_1, W_k \circ v_2) \tag{10}$$

where $\circ$ is the element-wise multiplication, and W_k is the k-th row of W, which controls the k-th perspective and assigns different weights to different dimensions of the d-dimensional space.

Full-Matching: In this strategy, each forward (or backward) vector $\vec{h}_i^+$ (or $\overleftarrow{h}_i^+$) is compared with the last time step of the forward (or backward) representation of the other sequence $\vec{h}_m^-$ (or $\overleftarrow{h}_m^-$).

$$\begin{aligned} \vec{m}_i^{full} &= f_m(\vec{h}_i^+, \vec{h}_m^-; W^1) \\ \overleftarrow{m}_i^{full} &= f_m(\overleftarrow{h}_i^+, \overleftarrow{h}_m^-; W^2) \end{aligned} \tag{11}$$

Maxpooling-Matching: In this strategy, each forward (or backward) vector $\vec{h}_i^+$ (or $\overleftarrow{h}_i^+$) is compared with every forward (or backward) vector of the other sequence $\vec{h}_j^-$ (or $\overleftarrow{h}_j^-$) for $j \in 1 \dots m$, and only the maximum value of each dimension is retained.

$$\begin{aligned} \vec{m}_i^{max} &= \max_{j \in (1 \dots m)} f_m(\vec{h}_i^+, \vec{h}_j^-; W^3) \\ \overleftarrow{m}_i^{max} &= \max_{j \in (1 \dots m)} f_m(\overleftarrow{h}_i^+, \overleftarrow{h}_j^-; W^4) \end{aligned} \tag{12}$$

where $\max_{j \in (1 \dots m)}$ is element-wise maximum.

These matching strategies are applied for both added and removed sequences. After, we concatenate each of the sequences of matching vectors and utilize another BiLSTM model to obtain a context-aware version each sequence. These two vector sequences are concatenated in the time dimension and provided to the decoder as context for the attention layer. Finally, to initialize the decoder, we concatenate the vectors from the last time-step of the BiLSTM models and aggregate them using an affine transformation. The decoder works analogously to the vanilla encoder-decoder case.

4 Empirical Study

Data: We consider real world open source Python projects. For our experiments using the content-augmented encoder, we resorted to the code-docstring-corpus (Barone and Sennrich, 2017). This dataset is a diverse parallel corpus of a hundred thousand Python functions with their docstrings, generated by scraping open source repositories on GitHub. In order to make our experiments comparable across settings, we only worked with projects that were also present in this dataset. We sorted the projects in the code-docstring-corpus according to their total number of commits in GitHub and chose the ones that contained at least 10,000 commits aiming at diversity in terms of topics. Specifically, in this paper we work with *Theano*, *astropy*, *nova*, *scikit-learn*, *mne-python*, *flocker* and *matplotlib*.

Following Loyola et al. (2017), we obtained all the *diff* files and the metadata associated to each commit, for a given project using the GitHub API. We also recovered information such as the author and message of each commit. The commit messages were processed using a modified version of the Penn Treebank tokenizer (Marcus et al., 1993). Besides using the rules defined by the original script, we replaced commit SHAs, commit author names and file names with generic tokens. In order to do so, we first collect the set of commit SHAs, committer names and project file names using the downloaded metadata for each commit. Each one of these lists is then matched against the words in the text to produce the output. Finally, we also removed certain repetitive patterns from the messages, such as the phrase *merge pull request*, keeping the rest of the content of each sequence, if any. Messages that solely contained these sequences were discarded as they provide no useful semantic information about the nature of the content of the commit. On the other hand, to obtain a representation of the source code content of each commit, we parsed the *diff* files and used a lexer (Brandl, 2016) to tokenize their contents in a per-line fashion. We ignored docstrings and code comment tokens, as well as tokens contained in literal strings. For our structure-aware encoder, when parsing the *diff* file we separately extract the added and removed lines, while the rest of the pre-processing remains the same.

We discarded commits that modify more than a single file, thus we consider only *atomic* commits. To combine the commit data with the code-docstring-corpus, we found the set of files modified by the commits and discarded all the ones that modify a file not present in the examples from the code-docstring-dataset. With this list, we extract all the source code and docstring lines from the corpus in a per-line fashion. In this manner, we create a mapping that allows us to recover, for each commit in our examples, the content and docstring lines of the file that commit modifies. Table 1 summarizes the size of our raw and pre-processed datasets.

Project	Total	Atomic	Content	Structure
Theano	22,995	15,814	7,708	15,210
astropy	19,599	12,195	4,708	11,896
nova	13,400	18,110	4,617	17,412
scikit-learn	15,575	12,885	3,965	12,482
mne-python	12,761	6,762	4,083	6,531
flocker	16,027	11,702	4,707	10,821
matplotlib	20,001	14,284	5,840	13,836

Table 1: Number of commits available on each dataset subset. Both the **Content** and **Structure** subsets are obtained using the **Atomic** subset.

Evaluation: As stated in the previous section, the problem of generating descriptions from source code changes does not yet have a formal way of evaluation. As the problem has certain elements from both translation and summarization, in principle metrics such as BLEU (Papineni et al., 2002) seem to appear as feasible alternatives for evaluation in our case. BLEU is based on n-gram overlap between the gold standard and the generated sequences. Smoothing techniques are also applied to deal with cases in which certain generated n-grams are not found on the gold standard. In particular, for this work we use BLEU-4 and for smoothing we simply add $\epsilon = 0.01$ to 0 counts.

On the other hand, we find METEOR (Lavie and Agarwal, 2007), a metric based on the alignment between hypothesis-reference pairs, which in turn is based on n-gram matching. Specifically, METEOR computes the alignment by comparing exact token matches, stemmed tokens and paraphrase matches. In addition to that, it also finds semantically similar tokens between hypotheses and references by using Word-Net synonyms. To obtain the final alignment, different overlap counts are combined using several free parameters that are tuned to emulate various human judgment tasks. Although this gives METEOR some extra flexibility, it makes it context dependent, specifically in terms of language. In our case, we work with the latest version available (1.5) with the model pre-trained for English, using the included scripts to tokenize and normalize punctuation.

Finally, we also considered MEANT (Lo and Wu, 2011). Our interest in this metric derives from the fact that it considers the verb as a key element when evaluating. More specifically, MEANT is based on semantic role labels and uses the Kuhn-Munkres algorithm to find matches in a bipartite graph built upon semantic frames. Thus, this metric aims at aligning the generated and gold standard sequences by finding semantically equivalent passages focusing on the main action in each passage. Compared to other metrics, the main drawback of standard MEANT is that it requires the inputs to have been annotated with their corresponding semantic role labels, while also requiring a notion of semantic distance to use for matching frames. To this end, we work with MEANT 2.0 (Lo, 2017), which is based on automatic SRL and word-embedding-based similarity for matching. For both requirements, we rely on SENNA (Collobert et al., 2011).

We trained our models using Adam (Kingma and Ba, 2014) with a learning rate of 0.001 with decay of 0.8 when there was no improvement in the validation loss. We used early stopping when the learning rate dropped below 10^{-4}. For evaluation on the test set, we used the three automatic metrics introduced before.

In addition, we conducted a crowd-sourced human evaluation. Concretely, we selected the best validation models on each case and relied on Amazon Mechanical Turk to evaluate the results on 50 randomly-chosen examples from the test sets. Each turker was presented with the gold standard and the generated message, and was asked to rate the level of correlation between them, from 1 (min) to 5 (max). We randomly swapped the order in which the messages appear, to avoid the turkers from easily locating each message. To ensure the quality of the evaluation, we filtered turkers using a quiz-based qualification in which users had to prove they had basic knowledge of Python and GitHub. In addition, each example was shown to 3 different turkers.

5 Results and Discussion

Table 2 summarizes our results in terms of all the evaluation metrics for both experiments, namely, i) the use of a content-aware encoder and ii) the use of the structure-aware encoder. In terms of notation, *Len* means the maximum length of the input sequence, *Use* refers to if content-aware and structure-aware was used (*No* means the standard baseline from Loyola et al. (2017)) . We see that in general, the usage of a context-aware encoder tends to increase performance, as the models with content perform better in 4 or 5 out of our 7 datasets, for each of the automatic evaluation metrics. These gains are also reflected in the average correlation scores from our human evaluation, where we can see that the content-aware models outperform the baseline in 4 datasets. In terms of sequence length, we observe that some content-aware models are able to outperform the baseline using shorter input-output pairs.

Regarding the usage of two linearized inputs, we see that the tendency is for the performance to decrease. This is evidenced in both automatic and human-based evaluation, where the majority of the structure-aware models perform worse than our baseline. We think this reinforces the urge for moving into a more ad-hoc representation in terms of structure, in which the code lines of the input *diff* are exploited thoroughly. Despite the fact that our current proposal goes in that direction, being designed to compare two sequential inputs, if these two inputs lack expressive power, still there is little the model can learn.

Comparing across models for a given automatic evaluation metric, we see a big difference in terms of their absolute values. In this sense, we note that MEANT offers scores that are arguably more lenient compared to BLEU and METEOR. We think the fact that these last two metrics are heavily based on n-gram overlap hinders their value. As MEANT essentially performs an action-based alignment between hypotheses and references, our intuition is that this could be a good direction in terms of evaluation, as the phenomenon to model is basically an action performed on a document, which is naturally articulated with a verb when summarizing the action performed (e.g. *Fix* a bug). Some empirical evidence about this was given by Jiang et al. (2017), who found that out of a sample of 1.6 M commit messages, roughly 47% of them begin with a verb and its direct object.

Regarding the mask-based approach, Table 3 presents some results associated to a initial exploratory study considering a subset of the projects. In this case, we can see that while the results are in the order of magnitude of the best results associated to the previous approach for content integration, there is still no clear pattern in terms of which metrics is more reliable as indicator of generative performance. In that sense, we consider it is critical to work towards obtaining an ad-hoc metric that is better aligned with the actual performance of the generation.

One known limitation of the current approach is that while the the data coming from the code changes in intrinsically time dependent, for the case of the code-docstring source, we are just using a static version, therefore we are not considering how docstring documentation could also change over time. While we were aware that such decision has direct implication on the vocabulary matching, it was a necessary simplification given the available dataset.

Additionally, given that the results associated to the use of two encoders did not produce a relevant improvement , we believe that even a two-encoder configuration does not produce a sufficiently expressive signal to be used by the decoder. That makes us conclude that treating a code change just a set of token sequences is not enough to obtain considerable increments and that it is necessary to obtain such input from a more flexible perspective, for example, by using an explicit dependency graph between changes, or even more complex constructs such as differences of execution traces or abstract syntax trees.

6 Conclusion and Future Work

We studied how to model the generation of description from source code changes by integrating the intra-code documentation as a guiding element to improve the quality of the descriptions. While the results from the empirical study are not completely conclusive, we consider that adding this extra information on average contribute positively, measured in terms of standard NLP metrics as well as through a human study. For future work, we consider necessary to focus on the expressiveness of the feature representations learned from the encoder. In that sense, we will explore other ways to treat the source code change, such as exploiting their abstract syntax tree representation.

Dataset	Content-aware encoder						Structure-aware encoder					
	Len.	Use	MEANT	METEOR	BLEU	Human	Len.	Use	MEANT	METEOR	BLEU	Human
Theano	200	No	0.1683	0.0505	0.0081	2.2533	100	No	0.1450	0.0310	0.0077	2.1667
	300	Yes	0.1600	0.0360	0.0080	2.0667	300	Yes	0.0080	0.0061	0.0053	1.4533
astropy	200	No	0.1942	0.1074	0.0292	2.5067	200	No	0.2586	0.0738	0.0220	2.8400
	300	Yes	0.2170	0.1100	0.0300	2.4867	200	Yes	0.2697	0.0555	0.0167	2.7133
flocker	300	No	0.0320	0.0405	0.0131	1.9133	300	No	0.1608	0.0668	0.0143	2.2267
	100	Yes	0.1100	0.0540	0.0110	2.0467	100	Yes	0.1186	0.0375	0.0054	2.1267
matplotlib	300	No	0.1944	0.0523	0.0126	2.3267	100	No	0.1687	0.0559	0.0139	2.3867
	100	Yes	0.1240	0.0830	0.0220	2.4067	300	Yes	0.1357	0.0542	0.0174	2.0000
mne-python	200	No	0.0147	0.0099	0.0052	2.2733	200	No	0.0568	0.0265	0.0171	2.4200
	200	Yes	0.0200	0.0250	0.0170	1.7667	300	Yes	0.0587	0.0250	0.0230	2.3933
nova	300	No	0.2798	0.0259	0.0275	2.4900	200	No	0.3151	0.0372	0.0187	2.4467
	100	Yes	0.3350	0.0410	0.0240	2.8066	300	Yes	0.2976	0.0477	0.0236	2.7000
scikit-learn	200	No	0.0669	0.1327	0.0276	2.0600	300	No	0.0547	0.0577	0.0170	2.0300
	100	Yes	0.0590	0.1010	0.0220	2.2200	300	Yes	0.0586	0.0341	0.0113	2.1267

Table 2: Best results using our context and structure aware architectures.

Dataset	Best Value		
	METEOR	MEANT	BLEU
Theano	0.1953	0.2103	0.0112
astropy	0.1077	0.2308	0.0302
matplotlib	0.0950	0.2397	0.0289

Table 3: Results of our content-based masking technique.

References

Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2017. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2017. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 451–462.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

Georg Brandl. 2016. Pygments: Python syntax highlighter. `http://pygments.org`.

Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically documenting program changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 33–42, New York, NY, USA. ACM.

Ronan Collobert, Jason Weston, Lon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12:2493–2537.

Luis Fernando Cortés-Coy, Mario Linares Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *SCAM*, volume 14, pages 275–284.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 631–642, New York, NY, USA. ACM.

Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE.

Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning unified features from natural and programming languages for locating buggy source code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1606–1612. AAAI Press.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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. Association for Computational Linguistics.

Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146. IEEE Press.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*.

Alon Lavie and Abhaya Agarwal. 2007. Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, StatMT '07, pages 228–231, Stroudsburg, PA, USA. Association for Computational Linguistics.

Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 709–712. IEEE Press.

Chi-kiu Lo. 2017. MEANT 2.0: Accurate semantic MT evaluation for any output language. In *Proceedings of the Second Conference on Machine Translation*, pages 589–597.

Chi-kiu Lo and Dekai Wu. 2011. MEANT: An inexpensive, high-accuracy, semi-automatic metric for evaluating translation utility based on semantic roles. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 220–229, Portland, Oregon, USA. Association for Computational Linguistics.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–292. Association for Computational Linguistics.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proc. AAAI*, pages 1287–1293. AAAI Press.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 574–584. IEEE.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM.

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM.

Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 390–401. ACM.

Zhiguo Wang, Wael Hamza, and Radu Florian. 2017. Bilateral multi-perspective matching for natural language sentences. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 4144–4150. AAAI Press.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.