Chaos to Clarity with Semantic Inferencing for Python Source Code Snippets

Aviel Stein
College of Computing and Informatics
Drexel University
Philadelphia, USA
ajs568@drexel.edu

Spiros Mancoridis
College of Computing and Informatics
Drexel University
Philadelphia, USA
spiros@drexel.edu

***Abstract*—In this work we explore using semantic clues from source code to generate meaningful descriptions of source code snippets. Our approach uses novel applications of natural language processing (NLP) tools to improve Neural Machine Translation (NMT) model's performance in the context of source code. Our method is simpler than comparable models in literature and scores a higher BLEU score. We trained our model on 20,000 code-comment pairs from the Source Code Analysis Dataset (SCAD). We use a custom-made Python source code tokenizer. The architecture of our learning model consists of attention, Recurrent Neural Networks (RNN's), Long Short-Term neural networks (LSTM's), and Auto-encoders. To evaluate the efficacy of our model we measure the performance on a hold-out (test) dataset across several established translation metrics, which compare the proximity of original comments to the generated ones. Our methods out perform prior work and show a consistent distribution of results across these metrics with a mean Bilingual Evaluation Understudy (BLEU) score of 0.47 with simple NLP inspired methods.**

***Index Terms*—natural language, structured data, deep learning, semantic inference, source code analysis**

I. Introduction and Background

Software source code can be complex and, therefore, difficult to comprehend, even for experienced programmers. Developers spend a majority amount of time updating and maintaining code, a key part of which is understanding the software's purpose via comments [1]. Comments are one of the essential tools available to the software engineering. Unfortunately, comments are often absent or outdated. Deep Learning and probabilistic methods have been developed in an attempt to comment code automatically. The focus of these models has transitioned from the more difficult problem of translating source code into natural language to instead, the more tractable problem of generating useful abstract representations of the source code. The state-of-the-art approach introduced in [2] uses a Recurrent Neural Network (RNN) with an Attention cell within a Long Short-Term Memory module (LSTM) that feeds the representation of code snippets into downstream nodes of the network.

A. Neural Language Processing Techniques

NMT relies heavily on deep learning to achieve good results. In particular, it uses sequence-to-sequence (Seq2Seq) models such as RNNs and LSTM architectures. Additionally, it can include other layers such as attention, encoders-decoders, and embeddings.

Both the input and output text is encoded to make it easily readable by a machine. In the context of our work, the encoding of data refers to mapping the token strings to integer values. This numerical encoding of the source language becomes the output of the first segment of a Seq2Seq model, often referred to by the encoder. The decoder, second segment of a Seq2Seq model, takes the encoding as input, and outputs the target language in encoded form, which is then converted to human readable text as the final step in the process.

The next significant component of NMT is the use of word embeddings, often referred to the process of eliciting similar network outputs for similar words/phrases. This is accomplished by configuring an embedding layer within the machine learning model that represents tokens as dense, fixed-length vectors. This setup, as opposed to something like a one-hot encoding, results in similar outputs for synonymous tokens [3].

B. Deep Neural Architectures

Multi-Layer Perceptrons (MLP's) are simpler neural network models that are referred to as Feedforward Neural Networks or Fully-Connected Networks (FCN). These simple models are insufficient for processing sequential data. On the other hand, RNN's are a category of neural network that accounts for temporal information. There are several ways of implementing RNN's, including LSTM's, bidirectional RNN's, and hierarchical RNN's. Essentially, an RNN feeds its output to itself at each subsequent time step forming a loop and passing down information as needed. At each time step, the input from the previous time step is encoded and used in conjunction with the current input [4]. RNNs suffer from the vanishing gradient problem, which occurs when RNNs have a very small gradient due to activation functions squashing a large input space into a smaller space between 0 and 1. Moreover, in RNNs the derivative becomes smaller and the gradients approach 0. In some cases this may completely halt the learning of the network as the gradients become too small and, thus, prevent the weights from being updated.

1) Sequence-to-Sequence Learning: Sequence-to-Sequence (Seq2Seq) models are a special class of RNNs used to solve

complex language problems. Seq2Seq models are used to convert sequences of one type to sequences of another type. For example, translation of English sentences to German sentences is a possible Seq2Seq task. As detailed above in the Encoding/Embedding discussion, the typical setup for this type of translation uses an encoder that takes the source sequence as input and outputs an encoding, then a decoder takes the encoding and outputs a sequence in the target language. The encoder and decoder are built using a similar architecture to one another, and are typically based on either an RNN or an LSTM.

Seq2Seq models are traditionally composed of an encoder-decoder architecture where the encoder maps the input sequence to a context vector of fixed length and the decoder generates the transformed output given this vector. As a result of the fixed-length vector, this model suffers from the inability of the system to remember longer sequences. Often, the beginning of the sequence is forgotten once the entire sequence is processed. Seq2Seq models discard all the intermediate states of the encoder and only use its final states to initialize the decoder. This tends to work well for short sequences, but as the length of the sequence increases, the compressed vector becomes a bottleneck and struggles to encode all the data of a long sequence into the vector.

2) Attention Module: Attention [5] was introduced to utilize all the states in order to construct the context vector and not to discard any of the intermediate encoder states. Specifically, attention uses a sub-network to determine which hidden encoder states to attend to and by how much. This sub-network consists of a fully-connected dense layer followed by a softmax activation function. Here, the probabilities generated represent the attention weights that aid the decoder in choosing how much attention to focus on a specific set of tokens in the input sequence. With an attention module included in a NLP task, it is apparent which tokens in the output sequence depend on specific tokens in the input sequence and can normally be represented in terms of a heat map, deemed an attention map.

C. Evaluation of NMT Methods

Machine translation dates back to the 17th century [6] but it was not until the 2010's that modern Deep Learning (DL) methods started to be applied to translation [4], [7] now known as Neural Machine Translation. Google Translate relies on artificial neural networks, typically ones with RNN's and LSTM layers [8]. For long sequences, attention mechanisms help improve accuracy. There are many considerations one must take into account when attempting automating translation. For instance, the lexical, syntactic, and semantic information between the source and target language must be maintained. Generally, translations are measured by their fluency (correct syntax and spelling) and adequacy (semantics); however, this evaluation is challenging because there are no single correct answers, and even human evaluators often disagree.

Evaluation metrics can be used to quantify the quality of translations. An example of a widely-used translation metric is Bilingual Evaluation Understudy (BLEU), which considers the N-gram overlap between machine translation output and reference translations. Another translation metric is Metric for Evaluation of Translation with Explicit ORdering (METEOR), which gives partial credit for matching stems, synonyms, and the use of paraphrases. These metrics are nor perfect however. Specifically, these metrics treat all words as equally relevant; thus, the scores are not always informative, and even human translators score low.

II. Methods

A. Dataset

In our work, we used the Source Code Analysis Dataset (SCAD) [9], which is a collection of code-comment pairs mined from reputable GitHub repositories, to train and validate our Neural Machine Translation (NMT). The original SCAD contains Java, Python, C++, and C code, and we focus exclusively on Python, but other languages could be incorporated into future work. Only Github repositories with 10 or more stars and a redistributable license were considered. This dataset has the advantage of containing blocks of code of various granularity (classes, functions, methods, and variables) as well as being open-source and easily accessible.

B. Preprocessing

We choose to only consider Python because it is a versatile language and extremely popular. We filtered the code snippets for sequences with a maximum length of 100 characters between 4 and 20 lines of code. We used Google's sentencepiece [10] to tokenize the code and limit the vocabulary to fewer than 10,000 words. In this step, we generated dictionaries that map each token to a unique id as well as reverse dictionaries that map each id back to its corresponding word. The senticepeice algorithm is a subword tokenizer that build on a compression algorithm for the statistically most useful tokens for the dataset and assures that any plain text we see can be tokenized, which helps with the vast vocabulary that comes with the neologism found in code. Once we have tokenized the code, we prefixed a start token (`<start>`) and post-fixed an end token (`<end>`) so the model can identify the length of the sequence. Lastly, the input sequences were padded to a maximum length (100 characters) so that every sequence is consistent and there are no variable-length inputs. When generalizing to code outside the dataset, we can segment programs or for longer blocks of code when necessary [11].

Good comments should convey the meaning of the code, be easy to read, and ideally have a consistent style [12]. Therefore we filter the code-comment pairs to try to assure that our data is correct, fluent, and consistent. For correctness we want to assume cohesion between the semantics of the code and comment so checked that the pairs share at least 3 tokens. Unlike in natural language where each language uses different words, many of the tokens in code are used as, or as part of, variable and function names. We found comments conveyed the meaning of the code more regularly when they shared some tokens with tokens in the code and were more focused on the function of the code than the codes function in its context. This

makes it more suitable for direct translation of code snippets without context. Additionally, some comments were overly technical and did not convey an abstract meaning of the code in a uniform way. To focus on code-comment pairs that are easily to read checked to make sure there are use of stop words in the comment. This is also a novel use of stop words, which are words that generally do not themselves contain significant semantic information but connect fragments, such as and, or, but, the, etc. This is a modified variant from traditional NLP techniques which often avoid using stop words because they crowd the important words. However, we found that by making sure some stop words were included (at least 3) we were able to get more sensible comments that were easier to understand. On the last point of having a consistent style, we do not impose any kind of restraints. However, something like that could be done post processing on the text, such as with domain adaptive text style transfer [13]. A model is only as good as the data is it trained on, so this prepossessing is one of the key contributions of the work because it constrains the data to high quality comments.

C. Neural Machine Translation

The model is implemented as an encoder-decoder architecture that follows the inherent attributes of the Seq2Seq model [7]. We utilized the Python programming language and PyTorch library to construct our project. In this model, the input is fed to a GRU encoder model that outputs shape $(\text{batch\_size}, \text{max\_length}, \text{hidden\_size})$ and generates a hidden state in the shape of $(\text{batch\_size}, \text{hidden\_units})$. We define the hyperparameters of the batch size as 64, embedding dimension as 256, and the number of hidden units as 1024. We arrived at using these values for the hyperparameters of the GRU after experimentation and found that these values provided efficient and accurate results. We used the adaptive momentum optimizer (Adam), used a teacher forcing ratio of 0.5, hidden dimension size of 1000, batch size of 5, learning rate of 0.001, and 1 percent dropout. The use of the Adam optimizer resulted in a faster convergence rate compared to traditional Stochastic Gradient Descent (SGD) and used the NLP technique of teach forcing to propagate the ground truth as input in further time steps in case the prediction from the previous time step was incorrect. To address the overfitting encountered during the experimentation of the model while tuning the hyperparameters, we included dropout to the GRU. The addition of probabilistically dropping connections from the GRU is a common practice to reduce overfitting.

The Seq2Seq model implemented uses an attention mechanism applied on top of the GRU encoder model. The weights of the attention mechanism are computed by a softmax function applied to the axis on the max_length or the length of the input. The score used to compute how much weight a given hidden state is given in the attention is measured by the Bahdanau's additive style in scoring as detailed in [14]. Moreover, the score is represented as the sum of the encoder output and hidden state each wrapped in their own dense fully connected (FC) layer and the sum of these FC outputs wrapped in a $\tanh$ activation function, then fed into another FC layer. The context vector is computed as the dot product of the attention weights and the output of the encoder, $\bar{\mathbf{h}}_s$. The final result of the attention is the composition of the context vector, $\mathbf{c}_t$, and the hidden state, $\mathbf{h}_t$, wrapped in an activation function.

The first part of the Seq2Seq model is the encoder, which takes as input the context vector from the Bahdanau attention module as well as the output of the input passed to an embedding layer (see Figure 1). The embedding used on the input maps the input from the size of the vocabulary (10,000) to the size of the embedding dimension (256) by transforming the positive integers into dense vectors of a fixed size. These two inputs: (1) embedding output and (2) context vector, are concatenated and the merged vector is then fed into a GRU. The main component of the encoder is the GRU with weights initialized according to the Glorot uniform distribution with a smaller standard deviation to avoid vanishing gradients [15]. The decoder follows a similar architecture and begins with the attention module computing the context vector and the attention weights (see Figure 2). Again, input is passed through an embedding layer and then concatenated with the context vector passed through the GRU with the same initialization technique. The main difference between the encoder and decoder is that the output of the GRU in the decoder is flattened and passed through an additional FC layer to the dimension of the vocabulary size with a linear activation function. This additional layer acts as the final downstream task of mapping the output decoding to an English sentence representing a comment that is easily readable by a human.

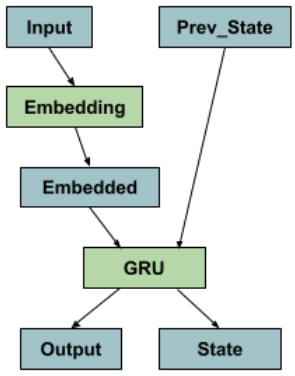

Fig. 1. Architecture for the Encoder Model.

The training of this model is traditional in terms of Seq2Seq learning where the input is passed through the encoder to generate its output, as well as the hidden state of the encoder. These two outputs, encoder output and encoder hidden state, are passed to the decoder along with the decoder input, which is represented as the start token. The decoder then outputs the predictions as well as the hidden state of the decoder. For each prediction, the hidden states and the prediction are fed back into the model to compute the loss of that given sample. As an alternative to back-propagation through time,

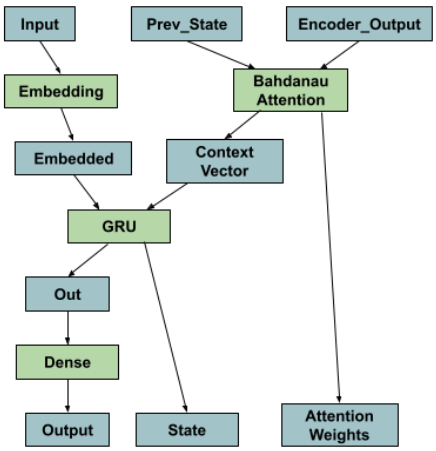

Fig. 2. Architecture for the Decoder Model.

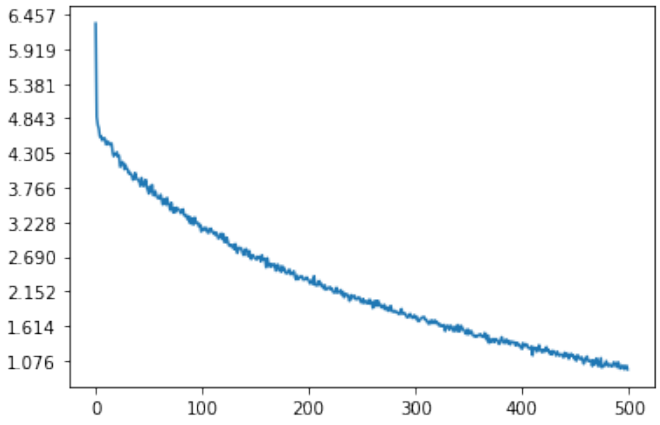

Fig. 3. Plot of the loss incurred by the model across epochs.

we use teacher forcing to train the RNN. This technique uses actual or expected output from the training data at the current time step as the input in the next time step rather than the output generated by the network. This method converges more quickly, and without teacher forcing, the hidden states of the model could be updated by a sequence of wrong predictions and cause the errors to accumulate when the model is struggling to learn. Finally, the gradients are computed with respect to the sparse categorical cross-entropy loss function and the model is optimized using Adam gradient descent.

The last important component of our architecture is the translation or the evaluation of the embedded output. The evaluation function is similar to the process followed during training, but excludes the use of teacher forcing and instead the input to the decoder is the prediction at the previous time step, along with its hidden state and the output of the encoder. The output of the evaluation halts its prediction once the model predicts the end token (`<end>`). Also, to note, the attention weights are stored at each time step to generate a map of the attention for each code snippet and translation pair.

III. Results

A. Neural Machine Translation

The model is quite computationally expensive and both the English and Python code languages have a vocabulary of 10,000 tokens, and there are 20,000 samples in the training set and 100 in the validation set, and 1,000 in the test set and ran it for 35,000 epochs. The categorical cross-entropy loss fell from 6.41 to a final value of 0.94.

B. Translation metrics

We used automated translation metrics common in NLP. Namely, we used BLEU [16], CHRF [17], GLUE [18], METEOR [19], and RIBES [20]. These methods are language independent and correlate with human evaluation. The Bilingual Evaluation Understudy score (BLEU), defined in Equation **??**, was first introduced in 2002 and was one of the first metrics to exhibit high correlation with human translations. The BLEU equation where the N-gram precision p_n using with N-grams up to length N, positive weights W_n for each N-gram length, and a brevity penalty BP for sentences that are too short. We also use the character N-gram F1 score (CHRF) which analyzes a sentence on the character level instead of the word level. The Google-BLEU (GLEU) score is a variation of the BLEU score which calculates the recall as the ratio of the number of matching N-grams in the reference translation to the total number of N-grams in the proposed translation, which works better on the sentence level. The Rank-based Intuitive Bilingual Evaluation Score (RIBES) tackles the problem of word ordering, rewarding sentence which maintain the correct word ordering which is important for causal semantics. METEOR improves on the BLEU metric by stemming words and considering synonyms in WordNet [21].

1) Automated Evaluation: Measuring the efficacy of NMT models can be difficult due to the subjectivity of evaluating language tasks. We choose to use automatic methods over human evaluation since it is the common practice in the literature and human evaluation can take a long time. Our choice of BLEU, CHRF, GLEU, and RIBES gives us a wide range of perspectives about the outcome of our work. In our analysis we initially obtained a BLEU score of .29 and, subsequently, after applying some custom filters increased the score to .49, which is considered a high quality translation.

Shared tokens	% Qualified	Average BLEU
0	100	.28
1	93.3	.30
2	70.6	.35
3	37.6	.47
4	11.4	.57

TABLE I

FILTERING FOR SHARED TOKENS BETWEEN CODE AND GENERATED COMMENT CAN HELP IMPROVE CONFIDENCE OF A GOOD BLEU SCORE BUT LIMITS NUMBER OF GENERATED COMMENTS. WHILE A LOW BLEU SCORE DOES NOT GUARANTEE A GOOD TRANSLATION, IT DOES HELP IMPROVE CONFIDENCE THAT THE COMMENT IS PRODUCING RELEVANT MATERIAL. FILTERING FOR 3 SHARED WORDS IS A GOOD COMPROMISE.

Filtering for some number of shared words can be helpful to assure that the comment is relevant. We recommend

using a filter of 3 shared words because it provides reliable translation according to the BLEU score and are focused on relevant portions of the code. The result of the RIBES score of 0.41 emphasises causal consistency, suggesting that our comments were attentive to the order they were generated. The distributions for CHRF both look to be fairly normally distributed with a mean score of 0.42 for the at character scale. BLEU and GLEU have the highest mean scores at 0.47 and 0.50 respectively which are at the token scale. We see similar distributions for very good and very bad results with most distributed favorably well given the difficulty of the code-comment pairs. Good natural language translations often score between 0.4 and 0.7 even with human made translations since there are many ways to translate a sentence. This is also true of code, which can be described in many ways resulting in direct translations to be very rare. Additionally code—as well as the comments describing them—tend to be neologistic, *i.e.,* the names of variables, functions, classes, and libraries are frequently made up on the spot and not in any common lexicon. This could have significant effects on the NMT metric scores.

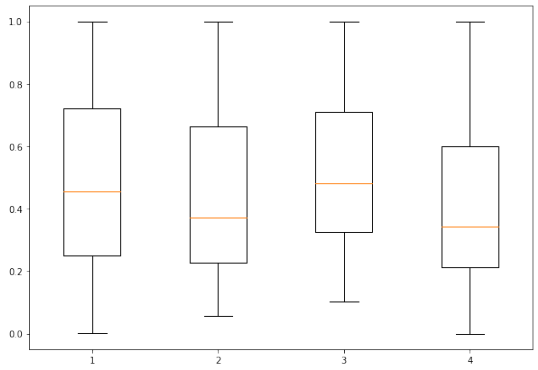

Fig. 4. These are the results from filtering with at least 3 shared words. 1, 2, 3, and 4 represent the distributions of BLEU, CHRF, GLEU, and RIBES scores. A comparison of translation metrics show a distribution and generally positive results.

2) Human Level Evaluation: Due to the extensive length of our training and testing samples, it would take too long to examine each output, we detail a few samples and provide some analysis. We noticed that sometimes the generated comments are more succinct or do not go into as much detail as the original comment and focus more on the core function. Occasionally, our technique would produce output that was very close to that in the test dataset, but our technique would choose different words such as "get" in place of "return". We also noticed that some translations with a very low BLEU score still described the function of the code fairly well.

IV. Discussion

This work draws on several active areas of research in NLP and their applications to code comprehension. Additionally, our classification results are less prone to language-specific abnormalities that make NMT metrics hard to compare across

```
--------
Code
 def now_reign_year():

    now_ = datetime.datetime.now()
    return now_.year - 2015


Original Comment:  return now year reign for king .
Generated Comment: return the current year in date .
BLEU: 0.38570315774665637

--------
Code
     def get_used_size_from_instance(self, instance):

        raise NotImplementedError()


Original Comment: method used in update infra instances sizes . return use
d size in bytes from instance .
Generated Comment: get a size of the instance .
BLEU: 0.07868235647726078

----------
Code
     def device_state_attributes(self):

        return self._state_attributes

Original Comment: return all the state attributes .
Generated Comment: return the state attributes of the device .
BLEU: 0.6354719482589886

----------
Code
     def legacy_event_type(self):

        return self.name_to_event_type_map[self.name]


Original Comment: return the legacy event type of the current event
Generated Comment: return the string name type event .
BLEU: 0.4147310221083959
----------
```

Fig. 5. Generated comments that show that the translation score may be low but still be a good description of the code. In these cases they tend to be more concise than the original. However there are times where it misses some of the point.

languages. Regarding the NMT model we used, we found that natural language translation (*i.e.,* French to English) is easier than formal language translation (*i.e.,* Python source code to English). We tried several simple Seq2Seq models that performed adequate for natural language but were not able to produce reliable results for comment generation from corresponding source code. We found that an attention mechanism is required for reliable results.

Below are some of the generated attention plots showing the relationship between code tokens and the corresponding predicted output English text. While these examples exhibit fairly accurate comment results, their attention maps are much sparser than one might expect. We did not find the attention maps to be always informative, it seemed like attention focused on unrelated parts of the code and not on the shared words as we saw in our experience with NLP.

V. Threats to Validity

We have identified three main threats to validity. The first possible threat is that we evaluate and compare our work only on Python code and the respective English comments. While the model is trained on Python syntax as a result of the datasets we chose, the model is not designed for Python specifically and we would require more evaluation needed when generalizing our work to other languages.

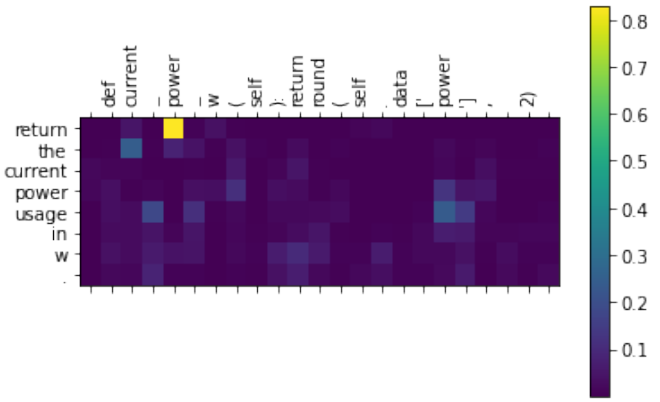

Fig. 6. Attention map for small function, the focus of the return is correlated with the power, but not between shared words.

Model	% Average BLEU
CoCoSUM [29]	.19
code2seq [22]	.19
NeuralCodeSum [22]	.22
codeBERT [30]	.24
Seq2seq [24]	.31
Full model (Python) [27]	.33
SG-Trans (Python) [28]	.33
TXT+AST+CFG+HAN+DRL [26]	.33
DeepCom [24]	.38
API+Code [23]	.37
TL-CodeSum(fine-tuned) [23]	.42
Full model (Java) [27]	.45
SG-Trans (Java) [28]	.46
Our Approach	.47

TABLE II
COMPARISON OF BLEU SCORES FROM THE LITERATURE, WITH VALUES CLOSER TO 1 INDICATING A GENERATED COMMENT BEING MORE SIMILAR TO THE ORIGINAL COMMENT. OUR GETS THE HIGHEST BLEU SCORE BUT DESPITE BEING ON PYTHON WHICH APPEARS TO BE TOUGHER THAN OTHER LANGUAGES, BUT BLEU SCORES CAN BE DECEIVING.

Another potential threat is that the code-comment pairs in the Source Code Analysis Dataset (SCAD) are collected from Github repositories developed by any user. Although the repositories are filtered to have at least a redistributable license and at least 10 stars, the comments may not be a factor when verifying the code through a license or tagging it with starts. It makes sense to train a model on real-world code; yet, Github repositories may receive stars for the novelty of the work and not the validity of the code or the helpfulness of the comments describing the code. As a result, a higher-quality dataset could be used in the future.

The automated evaluation metrics of the generated comments could pose as another potential threat to validity. BLEU, CHRF, GLEU, and RIBES scores can provide a good indication to how well our model performs compared to the reference text and other models, but there are scenarios where the model may generate valid comments that do not align to the ground truth (ie, the reference text). We chose the use of BLEU and other automated metrics to expedite our evaluation phase and allow for faster hyperparameter tuning and optimization in the modeling approaches we chose. The alternate method of evaluation could have been the use of human evaluators. Yet, with human evaluation, there exists the lack of reliability as another potential threat to validity. Furthermore, bias may exist in the scores assigned to the same sample by different evaluators.

VI. RELATED WORK

Summarization of code in the form of comments has become a very popular topic as of late, here is a survey of some of the methods built with similar goals [22]. Most methods use BLEU to evaluate the generated comments as discussed more in the evaluation section [12], [22]–[30]. There are a lot of ways to approach this problem depending on what results you want and data you use. Several researchers took advantage of the Abstract Syntax Tree (AST) which does appear to aid performance, and suggests a way to improve our work as well [12], [24]–[26], [31], [32]. Many languages have been tested with the application of ASTs, especially Java, but Python as well. While the work presented by Zhang et al. [33] is not directly applied to code-to-comment translation, their work shows that AST representation can be an effective method of representing code and can provide useful information in the structure for other NLP techniques to be used such as text generation and text classification. CoCoSum [29] leverages a Graph Neural Network (GNN) to learn not only on the code or the ASTs alone, but also includes data regarding the class info and the graph topology of UML in a multi-modal approach to summarize code. Similarly, [28] takes a multi-modal approach by jointly learning on the semantics of the code as well as structural properties of the code in their transformer model that accurately represents data flow and symbolic information.

Hu et al. [23] trained their model using data from API documentations, which is smart because there is likely a lot of overlap between what you would see in documentation and comments. However, the use of documentation can serve as a limiting factor such that this would generate documents on whole functions and not perform as well on inter-function comments that we see in our dataset, SCAD. Other work [25], [32] trains jointly on both code and the AST structure to generate its comments. Outside of transformers and other Seq2Seq modeling approaches, [12] attempts to summarize code building upon Deep Reinforcement Learning (DRL). The model architecture, HAN [26], uses DRL as well as multi-modal data to learn a code representation via program control flows as well as the hierarchical property of ASTs. While our approach does appear to perform better than other comparable state-of-the-art techniques looked at here, it doesn't mean that they are all comparable. The difference between languages and the complications with data quality can affect metrics like BLEU even when it performs well. That is why in this work we filter our dataset to use stop words and token overlap to clean the data and limit what is deemed acceptable.

To clarify, the novelty of our approach compared to the existing methods as mentioned before is the result of the pipeline of our unique preprocessing followed by the seq2seq model. Moreover, we use a simple seq2seq model that performs superiorly well on the preprocessed data that is generated from the use of sentencepiece tokenization, filtering stop and overlap words, and limiting the tokens per code trained. As a result, we show that the seq2seq GRU performs exceedingly well when we feed it data that is corresponding to "good comments": correct, fluent, and consistent.

VII. Conclusions and Future Work

A next step in our work could be to generate longer comments across large pieces of code; for example, in the form of a README file, that document entire code repositories (*i.e.,* several related source code programs). Another avenue to pursue is to exploit the structure of Abstract Syntax Trees (ASTs) as a representation of the code, rather than relying solely on sequential source code tokens.

Additionally, another direction to extend this work could be using the Seq2Seq and transformer models outside of translation, but to find security vulnerabilities in code. This could be done by using differences (diffs) in GitHub repositories, and using those diffs to learn before and after comparisons such that if you identify a vulnerability, you could suggest a solution.

NMT models have been valuable for the study of natural language and text repositories. When applied to converting code to English text, we see the NMT model is automatically able to describe the function of the code. Even though the act of automatically creating comments is not the same as a direct translation between "natural" languages, we can still use it to annotate code with some assurance that the quality of the generated code is good enough to be useful to a software engineer. In our work, we obtain comparable results from traditional NLP translation metrics such as a .47 mean BLEU score after preprocessing and filtering for shared words. This high quality data allows us to use a simple seq2seq model to outperform prior work because they correspond to "good comments": correct, fluent, and consistent. This score is on par, or better, than natural language domains found in the literature. We found that NMT models take a longer time to train than NLP models and require attention modules in addition to merely using a simple Seq2Seq model.

Acknowledgment

This research was funded by the Auerbach Berger Chair in Cybersecurity held by Spiros Mancoridis.

References

[1] B. P. Lientz and E. B. Swanson, *Software maintenance management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[2] S. Iyer, I. Konstas, A. Cheung, and L. 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)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16-1195

[3] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[4] N. Kalchbrenner and P. Blunsom, "Recurrent convolutional neural networks for discourse compositionality," *arXiv preprint arXiv:1306.3584*, 2013.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[6] D. Stein, "Machine translation: Past, present and future," *Language technologies for a multilingual Europe*, vol. 4, no. 5, 2018.

[7] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *arXiv:1409.3215 [cs]*, Dec. 2014, arXiv: 1409.3215. [Online]. Available: http://arxiv.org/abs/1409.3215

[8] D. Castelvecchi, "Deep learning boosts google translate tool," *Nature News*, 2016.

[9] B. Gelman, B. Obayomi, J. Moore, and D. Slater, "Source code analysis dataset," *Data in brief*, vol. 27, p. 104712, 2019.

[10] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *arXiv preprint arXiv:1808.06226*, 2018.

[11] A. Stein, D. Schwartz, Y. Shi, and S. Mancoridis, "Linguistic approach to segmenting source code," in *IEEE International Conference on Semantic Computing (ICSC'22)*, 2022.

[12] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[13] D. Li, Y. Zhang, Z. Gan, Y. Cheng, C. Brockett, M.-T. Sun, and B. Dolan, "Domain adaptive text style transfer," *arXiv preprint arXiv:1908.09395*, 2019.

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

[15] D. Mishkin and J. Matas, "All you need is a good init," *arXiv preprint arXiv:1511.06422*, 2015.

[16] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://www.aclweb.org/anthology/P02-1040

[17] M. Popović, "chrF: character n-gram F-score for automatic MT evaluation," in *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Lisbon, Portugal: Association for Computational Linguistics, Sep. 2015, pp. 392–395. [Online]. Available: https://www.aclweb.org/anthology/W15-3049

[18] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[19] A. Lavie and A. Agarwal, "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*. Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 228–231. [Online]. Available: https://www.aclweb.org/anthology/W07-0734

[20] H. Isozaki, T. Hirao, K. Duh, K. Sudoh, and H. Tsukada, "Automatic evaluation of translation quality for distant language pairs," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 2010, pp. 944–952.

[21] C. Strapparava, A. Valitutti *et al.*, "Wordnet affect: an affective extension of wordnet." in *Lrec*, vol. 4. Citeseer, 2004, p. 40.

[22] J. Mahmud, F. Faisal, R. I. Arnob, A. Anastasopoulos, and K. Moran, "Code to comment translation: A comparative study on model effectiveness & errors," *arXiv preprint arXiv:2106.08415*, 2021.

[23] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," pp. 2269–2275, 7 2018. [Online]. Available: https://doi.org/10.24963/ijcai.2018/314

[24] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 200–20 010.

[25] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[26] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Transactions on software Engineering*, 2020.

[27] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[28] S. Gao, C. Gao, Y. He, J. Zeng, L. Y. Nie, and X. Xia, "Code structure guided transformer for source code summarization," *arXiv preprint arXiv:2104.09340*, 2021.

[29] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, "Cocosum: Contextual code summarization with multi-relational graph neural network," *arXiv preprint arXiv:2107.01933*, 2021.

[30] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[31] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[32] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," *arXiv preprint arXiv:2103.11318*, 2021.

[33] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.