Journal of Computer Technology and Software

ISSN:2998-2383

Vol. 2, No. 3, 2023

Advancing Source Code Summaries via Abstract Syntax Trees and Neural Machine Translation

Rowan Sinclair

Department of Computer Science, University of California, Irvine, USA

rowan.sinclair93@uci.edu

Abstract: Source code summarization generates brief natural language descriptions of code, aiding developer documentation. Traditional neural models often overlook the hierarchical structure of code, leading to inaccuracies. This study introduces the ast-attendgru model, utilizing Abstract Syntax Trees (AST) to capture structural features. Evaluated on Java methods using the BLEU metric, our model outperforms traditional methods and an NLP baseline, especially in cases with minimal documentation. Incorporating AST significantly improves the coherence and accuracy of code summaries.

Keywords: Deep Learning; Source Code Comment Generation; Abstract Syntax Tree.

1. Introduction

Source code summarization is the task of writing brief natural language descriptions of code [1, 2, 3, 4]. These descriptions have long been the backbone of developer documentation such as JavaDocs [5]. Summaries like “uploads log files to the backup server” or “formats decimal values as scientific notation” can give programmers a clear picture of what code does, saving them time from comprehending the details of that code.

Trace its technological development, at first, the dominant strategy was based on sentence templates and heuristics derived from empirical studies [6-10]. Starting around 2016, data-driven strategies based on neural networks came to the forefront, leveraging gains from both the AI/NLP and mining software repositories research communities [11-14]. As far as we know, the existing deep learning based comment generation approaches mainly utilize the seq2seq model in which the program code is encoded into hidden space first and then decode it to produce the target comment. However, these kind of approaches have the following drawbacks: **(1)** they mainly take the source code as plain text and ignore the hierarchical structure of the source code; **(2)** most of the approaches only consider simple features, such as,tokens, which overlooking the hidden information that can help grab the relationships between source code and comments; **(3)** they typically train the decoder to produce the code annotation by calculating and maximizing the odds based on the subsequent natural language words, however in fact, they mainly produce the code annotation from scratch. Therefore, these drawbacks result in inferior comment generation accuracy and inconsistent of the generated comment.

To solve the limitations described above, this work proposes to use abstract syntax trees for source code summarization. The AST of source code provides additional structure features that are lost when you flatten source code to a sequence. These additional structure features allow the model to learn code representations even if the programmer provided features (e.g.identifiers) are obfuscated. Our solution is inspired by recent advances in neural machine translation, as well as an approach called SBT by Hu et al. [15]. We evaluate our approach using the automated metric BLEU and compare it to other relevant models

2. Approach

Our proposed neural model assumes a typical NMT architecture in which the model is asked to predict one word at a time. The input to the model is the code and AST, along with the preceding words in the summary. The model output is the next token in the summary sequence.

2.1. Model Overview

Our model is essentially an attentional encoder-decoder system, except with two encoders: one for code/text data and one for AST data. In the spirit of maintaining simplicity where possible, we used embedding and recurrent layers of equal size for the encoders. We concatenate the output of attention mechanisms for each encoder as depicted as following:

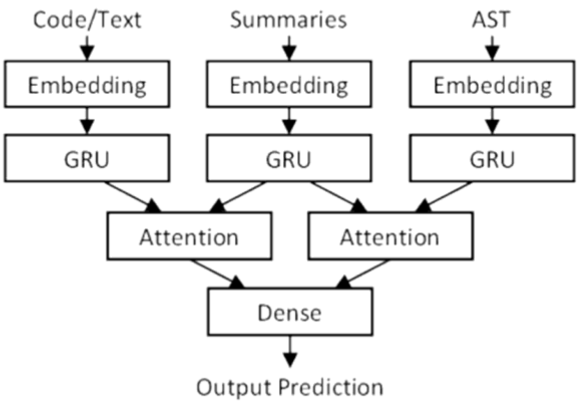


Figure 1. Model architecture

Precedent for combining different data sources comes heavily from image captioning [16] (e.g., merging convolution image output with a list of tags). One aim of this work is to demonstrate how a similar concept is beneficial for code summarization, in contrast to the usual seq2seq application to SE data in which all information is put into one sequence. We also hope to sow fertile ground for several areas of future work in creating unique processing techniques for each data type treating software's text and structure differently has a long tradition [17].

2.2. Model Details

To encourage reproducibility and for clarity, we explain our model as a walkthrough of our actual **Keras** implementation.

```
txt_input =  Input(shape=(self.txtlen,))
com_input = Input(shape=(self.comlen,))
ast_input =   Input(shape=(self.astlen,))
```

First, above, are three input layers corresponding to the code/text sequence, the comment sequence, and the flattened **AST** sequence. We chose the sequence lengths as a balance between model size and coverage of the dataset. The sequence sizes of 100 for code/text and AST, and 13 words for comment, each cover at least 80% of the training set. Shorter sequences are padded with zeros, and longer sequences are truncated.

```
ee=Embedding(output_dim=self.embdims,
        input_dim=self.txtvocabsize)(txt_input)
se=Embedding(output_dim=self.embdims,
        input_dim=self.astvocabsize)(ast_input)
```

We start with a fairly common encoding structure, including embedding layers for each of our encoded input types (code/text and AST). The embedding will output a shape of (batch size, txtvocabsize, embdims). What this means is that for every batch, each word in the sequence has one vector of length embdims. For example, (200, 100, 100) means that for each of 200 examples in a batch, there are 100 words and each word is represented by a 100 length embedding vector. We found two separate embeddings to have better performance than a unified embedding space.

```
ast_enc=CuDNNGRU(self.rnndims,return_state=T
               rue,return_sequences=False)
astout, sa = ast_enc(se)
```

Next is a GRU layer with rnndims units (we found 256 to provide good results without oversizing the model) to serve as the AST encoding. We used a CuDNNGRU to increase training speed, not for prediction performance. The return state flag is necessary so that we get the final hidden state of the AST encoder. The return sequences flag is necessary because we want the state at every cell instead just the final state. We need the state at every cell for the attention mechanism later.

```
txt_enc=CuDNNGRU(self.rnndims,return_state=True,
               return_sequences=True)
txtout, st = enc(ee, initial_state=sa)
```

The code/text encoder operates in nearly the same way as the AST encoder, except that we start the code/text GRU with the final state of the AST GRU. The effect is similar to if we had simply concatenated the inputs, except that 1) we keep separate embedding spaces, 2) we allow for attention to focus on each input differently rather than across input types, 3) we ensure that one input is not truncated by an excessively long sequence of the other input type, and 4) we "keep the door open" for further processing e.g. via convolution layers that would benefit one input type but not the other. As we show in our evaluation, this is an important point for future work.

Tensor ***txtout*** would normally have shape (***batch size, rnndims***), an rnndimslength vector representation of every input in the batch. However, since we have return sequences enabled, ***encout*** has the shape (***batch size, datvocabsize, rnndims***), which is the rnndims-length vector at every time-step. That is, the rnndims-length vector at every word in the sequence. So, we see the status of the output vector as it changes with each word in the sequence. We also have ***return state*** enabled, which just means that we get st, the ***rnndims*** vector from the last cell. This is a GRU, so this st is the same as the output vector, but we get it here anyway for convenience, to use as the initial state in the decoder.

```
ast_attn = dot([astout,encout], axes=[2, 2])
ast_attn = Activation(‘softmax’)(ast_attn) ast_context
= dot([ast_attn, txtout], axes=[2, 1])
```

We perform the same attention operations to the AST encoding as we do for the code/text encoding.

```
context = concatenate( [txt_context, ast_context,
               decout])
```

But, we still need to combine the code/text and AST context with the decoder sequence information. This is important because we send each word one at a time, as noted in the previous section. The model gets to look at the previous words in the sentence in addition to the words in the encoder sequences. It does not have the burden of predicting the entire output sequence all at once. Technically, what we have here are two context matrices with shape (**batch size, 13, 256**) and a decout with shape (**batch size, 13, 256**). The default axis is -1, which means the last part of the shape (the 256 one in this case). This creates a tensor of shape (**batch size, 13, 768**): one **768-length** vector for each of the 13 input elements instead of three 256-length vectors.

```
Out=TimeDistributed(Dense(self.rnndims,
        activation="relu"))(context)
```

We are nearing the point of predicting a next word. A TimeDistributed layer provides one dense layer per vector in the context matrix. The result is one rnndimslength vector for every element in the decoder sequence. For example, one 256-length vector for each of the 13 positions in the decoder sequence. Essentially, this creates one predictor for each of the 13 decoder positions.

```
out = Flatten()(out) out = Dense(self.comvocabsize,
          activation="softmax")(out)
```

However, we are trying to output a single word, the next word in the sequence. Ultimately we need a single output vector of length comsvocabsize. So we first flatten the (13, 256) matrix into a single (3328) vector, then we use a dense output layer of length comsvocabsize, and apply softmax.

```
Model     =     Model(inputs=[txt_input,com_input,
          ast_input], outputs=out)
```

The result is a model with code/text, AST, and comment sequence inputs, and a predicted next word in the comment sequence as output.

2.3. Hardware Details

The hardware on which we implemented, trained, and tested our model included one Xeon E5-1650v4 CPU, 64gb RAM, and two Quadro P5000 GPUs.It was necessary to train on GPUs with 16gb VRAM due to the large size of our model.

2.4. Corpus Preparation

We prepared a large corpus of Java methods from the Sourcerer repository provided by Lopes et al. [18]. The repository contains over 51 million Java methods from over 50000 projects. We considered updating the repository with new downloads from GitHub, but we found that the Sourcerer dataset was quite thorough, leading to a large amount of overlap with newer projects that could not be eliminated (due to name changes, code cloning, etc.). This overlap could lead to major validity problems for our experiments (e.g., if testing samples were inadvertently placed in the training set). We decided to use the Sourcerer projects exclusively.

We split the code and comments on camel case and underscore, removed non-alpha characters, and set to lower case. We did not perform stemming. We then split the dataset by project into training, validation, and test sets. By "by project" we mean that we randomly divided the projects into the three groups: 90% of projects into training, 5% into validation, and 5% into testing.

To obtain the **AST**s, we first used srcml [19] to extract an **XML** representation of each method. Then we built a tool to convert the **XML** representation into the flattened SBT representation, to generate SBT-formatted output described by Hu et al. [15] Finally, we created our own modification of **SBT** in which all the code structure remained intact, but in which we replaced all words (except official Java API class names) in the code to a special token. We call this **SBT-AO** for **SBT AST** only. We use this modification to simulate the case when only an **AST** can be extracted. From this corpus of Java methods, we create two datasets:

• The **standard dataset** contains three elements for each Java method: 1) the pre-processed Java source code for the method, 2) the pre-processed comment, and 3) the SBT-AO representation of the Java code.

• The **challenge dataset** contains two elements for each method: 1) the preprocessed comment, and 2) the SBT-AO representation of the Java code.

Technically, we also have a third dataset containing the default SBT representation (with code words) and the pre-processed comment, which we use for experiments to compare our approach to the baselines. However, the standard and challenge datasets are our focus in this paper, intended to compare the case when internal documentation is available, and the much more difficult case with only an **AST**.

3. Evaluation

This section covers our evaluation, comparing our approach to baselines over the standard and challenge datasets.

3.1. Research Questions

This section covers our evaluation, comparing our approach to baselines over the standard and challenge datasets.

Our research objective is to determine the performance difference between our approach and competitive baseline approaches in two situations that we explore through these Research Questions (RQs):

RQ1. What is the difference in performance between our approach and competitive approaches in the "standard" situation, assuming internal documentation?

RQ2. What is performance of our approach in the "challenge" situation, assuming an **AST** only?

Essentially, existing applications of NMT for the problem of code summarization almost entirely rely on the programmer writing meaningful internal documentation such as identifier names. As we will show, this assumption makes the problem "easy" for seq2seq NMT models, since many methods have internal documentation that is very similar to the summary comment (a phenomenon also observed by Tan et al. [20] and Louis et al. [21]). We ask **RQ1** in order to study the performance of our approach under this assumption.

In contrast, we ask **RQ2** because the assumption of internal documentation is often not valid. Very often, only the bytecode is available, or programmers neglect to write good internal documentation, or code has even been obfuscated deliberately. In these cases, it is usually still possible to extract an **AST** for a method, even if it contains no meaningful words. In principle, the structure of a program is all that is necessary to understand it, since ultimately that is what defines the behavior of the program. In practice, it is very difficult to connect structure directly to high-level concepts described in summaries. We seek to quantify a baseline performance level with our approach (since, to our knowledge, no published approach functions in this situation).

3.2. Baselines

To answer **RQ1** (the standard experiment), we compare our approach to three baselines. One baseline (which we call **attendgru**) is a generic attentional encoder-decoder model, to represent an application of a strong off-the-shelf approach from the **NLP** research area. Note that there are a huge variety of **NMT** systems described in the **NLP** literature, but that a vast majority have an attentional encoder-decoder model at their heart to maintain an "apples to apples" comparison, the baseline is identical to the "code/text" encoder in our approach (the decoder is identical as well). In essence, the baseline is the same as our proposed approach, except without the **AST** encoder and associated concatenation. While we could have chosen any number of approaches from **NLP** literature, it is very difficult to say up front which will perform best for code summarization, and we needed to ensure minimal differences to maximize validity of our results. If, for example, we had used an architecture with an **LSTM** instead of a **GRU** in the encoder, we would have no way of knowing if the difference between our approach and the baseline were due to the AST information we added, or due to using an **LSTM** instead of a **GRU.**

A second baseline is the **SBT** approach presented by Hu et al. [15]. This approach was presented at **ICPC'18**, and represents the latest publication about source code summarization in a software engineering venue. That paper used an LSTM-based encoder-decoder architecture based on a popular guide for building **seq2seq NMT** systems, but used their SBT representation of code instead of the source code only. For our baseline, we use their SBT representation, but use the same GRU-based encoder-decoder from our NLP baseline, also to ensure an "apples to apples" comparison. Since the model architecture is the same, we can safely attribute performance differences to the input format (e.g., SBT vs. code-only). A third baseline is **codenn,** presented by Iyer et al. [38]. Given the complexity of the approach, we used their publicly-available implementation. The original paper describes only applications to **SQL** and **C#**, but we noticed that their **C#** parser extracted common code features that are also available in **Java**. We made small modifications to the **C#** parser so that it would function equivalently for **Java**. We call our approach **ast-attendgru** in our experiments. We used a greedy search algorithm for inference for all approaches, rather than beam search, to minimize the number of experimental variables and computation cost.

3.3. Methodology

Our methodology to answer both RQs is identical, and follows best practice established throughout the literature on

NMT (see Section 2): for **RQ1**, we train our approach and each baseline with the training set from the standard dataset for a total of 10 epochs. Then, for each approach, we computed performance metrics for the model after each epoch against the validation set. (In all cases, validation performance began to degrade after five or six epochs.) Next we chose the model after the epoch with the highest validation performance, and computed performance metrics for this model against the testing set. These testing results are the results we report in this paper. Our methodology to answer **RQ2** differs only in that we trained and tested using the challenge dataset.

We report the performance metric BLEU [21], also in keeping with standard practice in NMT. BLEU is a measure of the text similarity between predicted summaries and reference summaries. We report a composite **BLEU** score in addition to BLEU1 through **BLEU4** (BLEUn is a measure of the similarity of n-length subsequences, versus entire summary sentences).

3.4. Threats to Validity

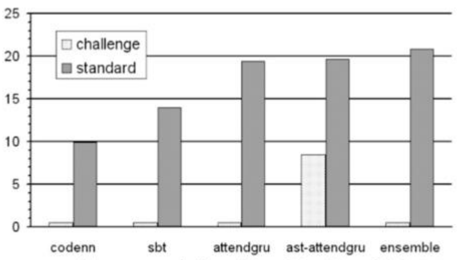
The primary threats to validity to this evaluation include: 1) Our dataset. We use a very large dataset with millions of Java methods in order to maximize the generalizability of our results, but the possibility remains that we would obtain different results with a different dataset. And, 2) we did not perform cross-validation. We attempt to mitigate this risk by using random samples to split the **training/validation/testing** sets, a different split could result in different performance. This risk is common among NMT experiments due to very high training computation costs (4+ hours per epoch).

4. Results

This section discusses our evaluation results and observations. After answering our research questions, we explore examples to give an insight into how the network functions and why it works. Note that we use these observations to build an ensemble method at the end of this paper.

4.1. RQ1: Standard Experiment

We found in the standard experiment that ast-attendgru and attendgru obtain roughly equal performance in terms of **BLEU** score, but provide orthogonal results, as we will explain in this section and the example in next section.



model	B	B1	B2	B3	B4	dataset
attendgru	19.4	39.0	22.0	14.8	11.3	standard
sbt	14.0	31.8	16.0	10.1	7.5	
codenn	9.95	21.2	9.7	7.6	6.3	
ast-attendgru (Our method)	19.6	39.3	22.2	14.9	11.4	
ast-attendgru	9.47	25.7	11.0	6.1	4.7	challenge

Figure 2. Below are BLEU1-4 scores and the composite BLEU score for each approach and dataset.

In terms of **BLEU** score, **ast-attendgru** and **attendgru** are roughly equal in performance: 19.6 BLEU vs 19.4 **BLEU**. **SBT** is lower, at about 14 BLEU, and codenn is about 10 BLEU. Figure 2 includes a table with the full **BLEU** results for each result (and additional data in our online appendix). For **SBT**, the results conflicted with our expectations based on the presenting paper [15], in which SBT outperformed a standard seq2seq model like **attendgru**. We see two possible explanations: First, even though our **seq2seq** baseline implementation represents a standard approach, there are a few architectural differences from the paper by Hu et al. [15], such as different embedding vector sizes. While we did not observe major changes in the results from these architectural differences in our own pilot studies, it is possible that “one’s mileage may vary” depending on the dataset. Second, the previous study did not split by project, so methods in the same project will be in the training and test set. The very high reported **BLEU** scores in [36] could be explained by overloaded methods with very similar structure – **SBT** would detect a function in the test set with a very similar **AST** to an overloaded method in the same project in the training set. The improvement by all approaches over **codenn** matches expectations from previous experiments.

The **codenn** approach was intended as a versatile technique for both code search and summarization, and was a relatively early attempt at applying **NMT** to the code summarization problem. In addition, it was designed for **C#** and **SQL** datasets; we adapted it to Java as described in the previous section.

A key observation of the standard experiment is that **ast-attendgru** and **attendgru** provide orthogonal predictions – there is a set of methods in which one performs better, and a different set in which the other has higher performance. While ast-attendgru is slightly ahead of **attendgru**, we do not view a 0.2 BLEU difference a major improvement in and of itself. Normally we would expect an approach to outperform a different approach by some margin across a majority of the examples (i.e., non-orthogonal performance), and this is indeed what we observe when comparing **ast-attendgru** to **SBT**, as shown on the left below (around 60k methods in which ast-attendgru performed better, vs. 20k for **SBT**):

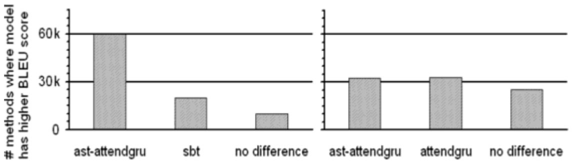


Figure 3. Comparison of BLEU score overlap between models

But what we observe for **ast-attendgru** and **attendgru** is that there are two sets of roughly 33k methods in the 91k test set in which one or another approach has higher performance (above, right). In other words, among the predictions in which there was a difference between the approaches, **ast-attendgru** and gives better predictions (in terms of BLEU score) for about half, while **attendgru** performs better on about half.

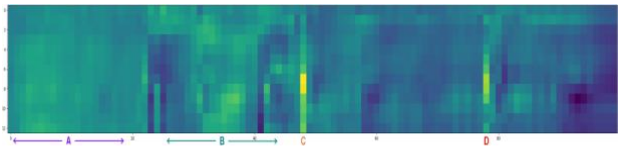


Figure 4. Heatmap of the attention layer in ast-attendgru for the AST input for Example 1. The x-axis is the summary input and the y-axis is the AST (SBT-AO) input. High activation (more yellow) indicates more attention paid to e.g., position 48 of the AST input.

4.2. RQ2: Challenge Experiment

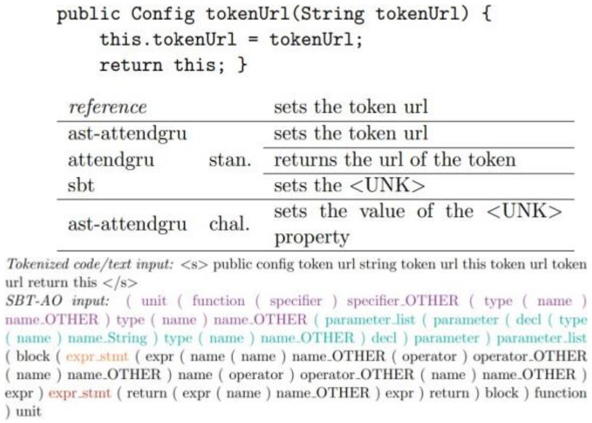
We obtain a **BLEU** score of about 9.5 for **ast-attendgru** in the challenge experiment. Note that the only difference between the standard and challenge experiments is that we trained and tested using the AST only, in the form of the **SBT-AO** representation fed to ast-attendgru. Technically, there are other configurations that would produce the same result, such as using **SBT-AO** as input to **attendgru** instead of the source code. Any of these configurations would meet our objective with this experiment of establishing performance for the scenario when only an **AST** is available.

4.2.1. Explanation and Example

Merely reporting **BLEU** scores leaves an open question as to what the scores mean in practice. Consider these two examples from the standard and challenge experiments (method IDs align with our downloadable dataset for reproducibility). We chose the following examples for illustrative purposes, and as an aid for explanation. While relatively short, we feel that these methods provide a useful insight into how the models operate. For a more in depth analysis, a human evaluation would be required, which is beyond the scope of this paper.

Example 1 is one of the cases where **ast-attendgru** succeeds when **attendgru** fails. To understand why, recall that, in our model as with a majority of **NMT** systems, the system predicts a sentence one word at a time. For each word, the model receives information about the method (the code/text plus the AST for models that use it), along with each word that has been predicted so far. So to predict "token", **ast-attendgru** would receive the code/text, the **AS**T, and the phrase "sets the".In contrast, **attendgru** only receives the code/text and "sets the". To predict the first word, "sets", **attendgru** only knows that it is the start of the sentence (indicated by a start-of-sentence token), and the code/text input. To help make the prediction **attendgru** is equipped with an attention layer learned during training to attend to certain parts of the input. That layer is depicted in Figure 4.4(a). Note that there is high activation (bright yellow) in position (14,1), indicating significant attention paid to location 14 in the code/text input: this is the word return. What has happened is that, during training, the model saw many examples of getter methods that were only a few lines and ended with a return.

Example 1:

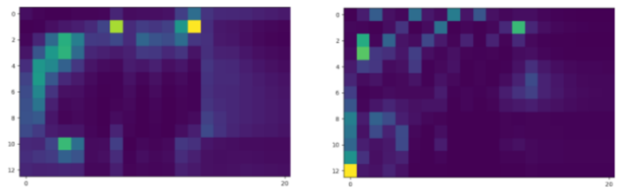


```
public Config tokenUrl(String tokenUrl) {
    this.tokenUrl = tokenUrl;
    return this; }
```

reference		sets the token url
ast-attendgru		sets the token url
attendgru	stan.	returns the url of the token
sbt		sets the <UNK>
ast-attendgru	chal.	sets the value of the <UNK> property

Tokenized code/text input: <s> public config token url string token url this token url token url return this </s>

SBT-AO input: (unit (function (specifier) specifier_OTHER (type (name) name_OTHER) type (name) name_OTHER (parameter_list (parameter (decl (type (name) name_String) type (name) name_OTHER) decl) parameter) parameter_list (block (expr_stmt (expr (name (name) name_OTHER (operator) operator_OTHER (name) name_OTHER) name (operator) operator_OTHER (name) name_OTHER) expr) expr_stmt (return (expr (name) name_OTHER) expr) return) block) function) unit



(a) (b)

Figure 5. Heatmaps of the attention layer in (a) attendgru and (b) ast-attendgru for the code/text input for Example 1. The x-axis is the 13 positions in the summary input. The y-axis is the 100 positions in the code input. Images are truncated to code input length

In many cases, the model could rely on very explicit method names, **attendgru** performed remarkably well in these cases, as the situation is quite like natural language – it learns to align words in the input vocabulary to words in the target vocabulary, and where they belong in a sentence. However, in cases such as Example 1 where the method name does not clearly state what the method should do (the name **tokenUrl** is not obviously a setter), **attendgru** struggles to choose the right words, even if, as in **Example 1**, it correctly identifies the subject of the action ("url of the token").

These situations are where the **AST** is beneficial. The code/text activation layer for **ast-attendgru** attends heavily to the start of sentence token (note column 0 in Figure 5(b)), which, since is the start of every sentence, probably acts like a "not sure" signal. But the model also has the **AST** input. Figure 5 shows the AST attention layer of **ast-attendgru** when trying to predict the first word. There are four areas of interest that help elucidate how the model processes the structure of the model, denoted A through D in the figure, and color-coded to the corresponding areas in the **AST** input. First, area A, is the portion of the method signature prior to the parameter. Recall that our **AST** representation is structure only, so almost all methods will start the same way. So as expected, the attention in area A is largely formless. The heatmap shows much more definition in area B. It is the parameter list, and the model has likely learned that short methods with parameter lists tend to be setters. The model activates very heavily at locations C and D, which are the start and end of the expr stmt AST node. A very common situation in the training set is that a short method with a parameter and an assignment is a setter. The model has learned this and chose "sets" as the first word.

All of the models with **AST** input correctly chose "sets". **SBT** found that the method is a setter, but could not determine what was being set – we attribute this behavior to the fact that the **SBT** representation blends the code/text and structural information into a single input, which creates a challenge for the model to learn orthogonal types of information in the same vector space (which work in other areas e.g. image captioning implies is not advisable [23]). While there is not space in this paper to explore fully, we note that even **ast-attendgru** during the challenge experiment correctly characterized the method as setting the value of a property, generating an unknown token when it could not determine which property. In fact, **ast-attendgru** correctly predicted the first word of the summary (which is usually a verb) 33% of the time during the challenge experiment, compared to 52% of the time in the standard experiment.

5. Ensemble Decoding and Future Work

As a hint toward future work, we test a combination of the attendgru and ast-attendgru models using ensemble decoding. The combination itself is straightforward: we compute an element-wise mean of the output vector of each model (the same trained models used in our evaluation). The training and test procedure does not change, except that during prediction, we use the maximum value of the combined output vector, rather than just one output vector from one model. This is the same ensemble decoding procedure implemented by **OpenNMT** [24], and is one of the most common of several options described by literature on multi-source NMT [25]. Since we are combining output vectors, the models "work together" during prediction of every word – it is not just choosing one model or another for the whole sentence. The idea is that one model may assign similar weights in the output vector to two or more words, in cases where it performs less well. And another model that performs better in that situation may assign more weight to a single word. In our system, the hope is that attendgru will contribute more when code/text words are clear, but ast-attendgru will contribute more when they are unclear.

The ensemble decoding procedure improves performance to 20.9 **BLEU**, from 19.6 for **ast-attendgru** and 19.4 for **attendgru**. This is more than a full **BLEU** point improvement, which is quite significant for a relatively simple procedure. This result points us to future work including more advanced ensemble decoding (e.g. predicting when to use one model or another), optimizations to the network (e.g. dropout, parameter tuning), and, critically, using different data processing techniques on each type of input.

6. Related Work

Automatic comment generation approaches vary from manually-crafted templates [26,27, 28], IR [29,30, 31, 32] to neural models [33,34,35].

Comment generation based on manually-craft templates was one of the common methods for generating comments. Sridhara et al. [36] developed the **Software Word Usage Model (SWUM)** to capture the occurrences of terms in source code and their linguistic and structural relationships and then defined different templates for different semantic segments in source code to generate readable natural language. Moreno et al. [37] defined heuristic rules to select relevant information in the source code, and then divided the comments into four parts, and defined different text templates for each part to generate natural language descriptions. McBurney et al. [38] also used the SWUM model to extract the keywords in the Java method, employed the PageRank algorithm to select the important methods in the given method's context, and used a template-based text generation system to generate comments. These frameworks have achieved good results on Java classes and methods.

IR techniques have been widely used in comment generation task. Haiduc et al. [39] used two IR techniques, Vector Space Model and Latent Semantic Indexing, to retrieve relevant terms from a software corpus, and then organized these terms into comments. Eddy et al. [40] used hierarchical **PAM**, a probabilistic model that selected relevant terms from the corpus and included them to the comments. Unlike the first two research works, Wong et al. [43] proposed that code snippets and their descriptions on the **Q&A** sites can be used to generate comments for a piece of code. They used a token-based code clone detection tool SIM to detect similar code snippets and used their comments as target comments. Wong et al. [42] further thought that the resources of the Q&A sites were limited and proposed to use token-based code clone detection tools to retrieve similar code snippets from GitHub and leverage the information obtained from their comments to generate comments.

Recently many neural networks have been proposed for comment generation. With large-scale corpora for training, neural based approaches quickly became state-of-the-art models on this task. Iyer et al. [16] first introduced the seq2seq model from neural machine translation into comment generation, whose encoder is the token embedding and decoder is an LSTM. Their model outperforms traditional methods on C# and SQL summaries. Inspired by the difference between natural language and programming language, Hu et al. [15] proposed a neural model named **DeepCom** to capture the structural information of source code. They proposed a structure-based traversal method, using one LSTM to process the AST's traversal sequence, and the other **LSTM** to generate comments for Java methods. LeClair et al. [25] proposed a neural method to predict the comment by combining the sequence information and structure information of the source code with two GRU encoders. In addition, they reconstructed the benchmark dataset for this task, removed duplicate and auto-generated code in the dataset, and divided the dataset into training, validation, and test by project.

We proposes to use abstract syntax trees for source code summarization .Our solution is inspired by recent advances in neural machine translation, as well as an approach called **SBT** by Hu et al[15].

7. Conclusion

We have presented a neural model for generating natural language descriptions of subroutines. We implement our model and evaluate it over a large dataset of Java methods. We demonstrate that our model **ast-attendgru**, in terms of **BLEU** score, outperforms baselines from **SE** literature and is slightly ahead of a strong off the shelf approach from NLP literature. We also demonstrate that and ensemble of our approach and the off-the-shelf NLP approach outperforms all other tested configurations. We provide a walk-through example to provide insight into how the models work. We conclude that the default NMT system works well in situations where good internal documentation is provided, but less well when it is not provided, and that **ast-attendgru** assists in these cases. We demonstrate how **ast-attendgru** can produce coherent predictions even with zero internal documentation.

References

[1] N. KALCHBRENNER, L. ESPEHOLT, K. SIMONYAN, A. V. D. OORD, AND K. KAVUKCUOGLU, Neural machine

translation in linear time, arXiv preprint arXiv:1610.10099, (2016).

[2] A. GRAVES, Generating sequences with recurrent neural networks, arXiv preprint arXiv:1308.0850, (2013).

[3] A. LOUIS, S. K. DASH, E. T. BARR, AND C. SUTTON, Deep learning to detect redundant method comments, arXiv preprint arXiv:1806.04616, (2018).

[4] R. COLLOBERT AND J. WESTON, A unified architecture for natural language processing: Deep neural networks with multitask learning, in Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), 2008, pp. 160–167.

[5] 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 (1), 2016, pp. 2073–2083.

[6] M. MALHOTRA AND J. K. CHHABRA, Class level code summarization based on dependencies and micro patterns, in 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), IEEE, 2018, pp. 1011–1016.

[7] A. SEE, P. J. LIU, AND C. D. MANNING, Get to the point: Summarization with pointer-generator networks, in Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Vancouver, Canada, July 2017, Association for Computational Linguistics, pp. 1073–1083.

[8] H. SHI, H. ZHOU, J. CHEN, AND L. LI, On tree-based neural sentence modeling, arXiv preprint arXiv:1808.09644, (2018).

[9] 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 on association for computational linguistics, Association for Computational Linguistics, 2002, pp. 311–318.K.

[10] M. HAMMAD, A. ABULJADAYEL, AND M. KHALAF, Summarizing services of java packages, Lecture Notes on Software Engineering, 4 (2016), pp. 129–132.

[11] L. MORENO, J. APONTE, G. SRIDHARA, A. MARCUS, L. POLLOCK, AND K. VIJAYSHANKER, Automatic generation of natural language summaries for java classes, in 2013 21st International Conference on Program Comprehension (ICPC), IEEE, 2013, pp. 23–32.

[12] R. COLLOBERT AND J. WESTON, A unified architecture for natural language processing: Deep neural networks with multitask learning, in Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 200.

[13] N. KALCHBRENNER, L. ESPEHOLT, K. SIMONYAN, A. V. D. OORD, AND K. KAVUKCUOGLU, Neural machine translation in linear time, arXiv preprint arXiv:1610.10099, (2016).8), 2008, pp. 160–167.

[14] N. J. ABID, N. DRAGAN, M. L. COLLARD, AND J. I. MALETIC, Using stereotypes in the automatic generation of natural language summaries for c++ methods, in 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2015, pp. 561–565.

[15] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin. Deep code comment generation. In Proceedings of the 26th International Conference on Program Comprehension, pages 200–210. ACM, 2018.

[16] K. Chen, J. Wang, L.-C. Chen, H. Gao, W. Xu, and R. Nevatia. Abc-cnn: An attention based convolutional neural network for visual question answering. arXiv preprint arXiv:1511.05960, 2015.

[17] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In 25th IEEE/ACM International Conference on Software Engineering (ICSE'03), pages 125–137, Portland, OR, 2003.

[18] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010. URL http://www.ics. uci.edu/$\ sim $ lopes/ datasets/.

[19] M. L. Collard, M. J. Decker, and J. I. Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on, pages 173–184. IEEE, 2011.

[20] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 260–269, April 2012. doi: 10.1109/ICST.2012.106.

[21] A. Louis, S. K. Dash, E. T. Barr, and C. A. Sutton. Deep learning to detect redundant method comments. CoRR, abs/1806.04616, 2018.

[22] 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 on Association for Computational Linguistics, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics, Association for Computational Linguistics. doi: 10.3115/1073083.1073135.

[23] Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3156–3164, 2015.

[24] S.-A. Gr¨onroos. Opennmt ensemble decoding, 2019. URL https://github. com/OpenNMT/OpenNMT-py/pull/732.

[25] E. Garmash and C. Monz. Ensemble learning for multi-source neural machine translation. In Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers, pages 1409–1418, 2016.

[26] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. IEEE Trans. Software Eng. 42, 2 (2016), 103–119. https://doi. org/ 10.1109/TSE.2015.2465386.

[27] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013. 23–32. https://doi.org/10. 1109/ ICPC. 2013. 6613 830.

[28] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. VijayShanker. 2010. Towards automatically generating summary comments for Java methods. In ASE. ACM, 43–52.

[29] Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, and Jeffrey C. Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013. 13–22. https:// doi.org/10.1109/ICPC.2013. 6613829.

[30] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In 17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA. 35–44. https://doi.org/ 10.1109/ WCRE.2010.13.

[31] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016. 87–98. https:// doi. org/ 10.1145/2970276. 2970326.

[32] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015. 380–389. https://doi.org/ 10. 1109/ SANER.2015.7081848.

[33] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension.

[34] 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, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers. https://www.aclweb. org/anthology/P16-1195/.

[35] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. 795–806. https://doi.org/ 10. 1109/ICSE.2019.00087.

[36] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori L. Pollock, and K. VijayShanker. 2010. Towards automatically generating summary comments for Java methods. In ASE. ACM, 43–52.

[37] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013. 23–32. https://doi.org/10. 1109/ ICPC. 2013. 6613830.

[38] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. IEEE Trans. Software Eng. 42, 2 (2016), 103–119. https:// doi.org/ 10.1109/TSE.2015.2465386.

[39] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In 17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA. 35–44. https://doi.org/ 10. 1109/WCRE.2010.13.

[40] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Auto Comment: Mining question and answer sites for automatic comment generation. In 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. 55–67. https://doi.org/10.1109/ASE. 2013.6693113.