# AutoComment
Comment Generation in Java Code

Navin Raj Prabhu, TU Delft
Jonathan Katzy, TU Delft
Rafail Skoulos, TU Delft
Thomas Pfann, TU Delft

October, 2019

## Abstract
*Commenting large code databases is crucial for code comprehension and efficient maintenance of a code base. Therefore, automatic code generation would be incredibly beneficial for both programmer and future maintainer of the code. In this paper, we propose a comment generator model using new state of the art techniques developed in the previous years, based on code2seq[1] for comment generation in Java code. With the DeepCom[2] as the baseline, the paper focuses on replicating the code2seq model with added capabilities such as, predicting natural language (Method-1) and modified ASTs (Method-2). The results shows us that, Method-2, is capable of understanding the syntactic and semantic meaning of java codes to generate comments automatically, but suffers from the incapability to generate longer and complete comments, hence leading to a poor BLEU-4 score when compared to the baseline. The source code and the dataset of the project can be found in -* `https://github.com/LRNavin/AutoComments`*.*

**Keywords: comment generation, deep learning, machine learning, LSTM, Code2Seq, Software Development, Java**

## 1 Introduction
For large software systems, the maintenance phase tends to have a comparatively longer duration than all the previous life-cycle phases taken together [3]. A large part of the maintenance cycle is software comprehension. So much so that around 59% of all time spent during the software development life cycle is spent on understanding of the code base. This due to time constraints and tight scheduling of projects which results into mismatching of code comments, completely omitting comments, or not updating comments of patched code in the database [2]. This has resulted in much research being conducted on automatic generation of comments given non-commented source code. Some of the first research on this subject done by Sridhara et al. [4] [5] [6] focuses on automatic comment creation for certain statements and methods with their parameters respectively. This really brought about the use of automatic learning to aid programmers with automatic comment suggestion.

The standard encoder decoder framework has proven to be highly effective in comment generation, as shown in [1]. The fully connected layer between encoder and decoder is represented by Long Short-Term Memory (LSTM) which are a form of Recurrent Neural Network (RNN). Due to LSTMs innate feedback connection, it can in addition to processing singular data points such as images, process time-series data such as speech and video. This feedback feature makes it particularly powerful within the field of automatic comment generation. Feedback allows an LSTM to cope with the lag between different important and characteristic events within code.

First we will first give an overview of the current state of the art in section 2. Then we propose a novel design for a comment generating model, explaining the architecture and specific design choices in section 3. We describe the experimental setup, results and discussion in sections 4, 5, and 6 respectively, and conclude our findings in section 7.

## 2 Background theory
Automatic comment generation for code has been around for years. Different techniques have been applied, ranging from using clone detection to copy comments [7] to applying deep learning techniques to predict generate new comments for uncommented code [2].

A related field of research is code summarization. Code summarization tries to summarize code snippets using as few words as possible. A lot of progress has been made in this area by using Bidirectional LSTMs such as in code2seq [1], where the combination of a bidirectional LSTM in an encoder -decoder architecture and attention mechanism is used to point out the most important nodes in an abstract syntax tree (AST). These points are then used to summarize the function. This results in a short summary of what the code is doing. These summaries are noticeably different from good commenting of course.

Traditionally, an encoder-decoder architecture uses 2 similar models. The first to encode the input sequence into a encoded state. This state is then passed to the decoder network that has been trained to decode its input into a required target. Often this technique is used in neural machine translation (NMT) as it allows the learning of translations between different types of sequences.

Previous works have shown that automatic generation of code comments is a feasible area for the application of deep learning models [2] [8]. These papers use syntactical information represented in an AST to train a sequence2sequence model (encoder-decoder architecture). This model is then used to predict the comment that would describe a piece of code from a collection of comments. To compare how well these models works the BLEU score is used. This score is a means of measuring machine translation accuracy [9]. The models in the papers [2] [8] achieved a BLEU-4 score of 38% and 39% respectively. Because the use of neural networks in software engineering is a rapidly progressing field, the authors of the paper (code2vec) upon which deepcomm is based have released a new paper boasting higher accuracy (code2seq). To improve upon the deepcomm paper we suggest a similar architecture, while using code2seq.

## 2.1 Out of Vocabulary problem

In NMT the Out of Vocabulary problem is well known, especially if made up words are used, such as in variable naming in source code. When doing language-to-language translations where most words have been defined and trained upon by the model this is no problem. This problem is very prevalent within comment generation though because a (Bi)LSTM in a sequence2sequence model can only predict words it has been trained on. In code many programmers have their own style of naming variables and it can differ majorly even within one project [2]. This presents a mayor challenge for good comment generation. In this paper, we tackle this probelm using the AST generation technique explained by Zhou et al. [10]. To be more specific, during AST extraction, instead of using each variable (leaf of the AST) with a generic name (e.g Var1) and its type only, we also added the actual name of the variable.

## 2.2 Encoder-decoder architecture.

The most common architecture in NMT is the encoder decoder [11] [1] [12]. The idea behind this architecture is that the encoder network (usually an LSTM, RNN or GRU) receives a sequence of inputs represented as $(x_1, x_2, ..., x_n)$ which is mapped to a hidden state using a sequence of continuous representations represented as $(z_1, z_2, ..., z_n)$. These hidden states are fed back into a decoder LSTM/RNN/GRU that takes one of the hidden states per time-step and predicts the most likely encoding, and feeds this encoding combined with the next hidden state to the next timestep. This assures that the prediction depends on previous output, as well as the input sequence. Figure 1 shows a graphical representation of the above mentioned process.

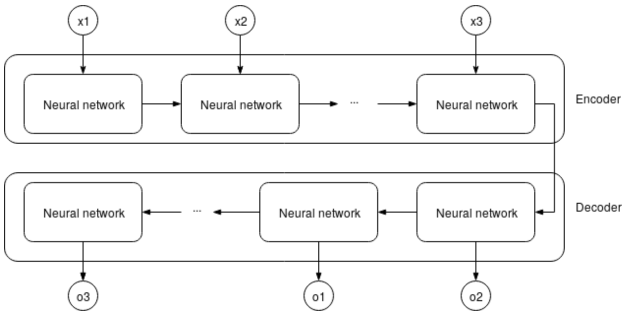

Figure 1: Generic Encoder-Decoder architecture, Neural network can be replaced by any form of Recurrent Neural Network

## 2.3 Bi-directional Long Short-Term Memory

A known problem with the encoder-decoder network is their inability to ’remember’ information in long input sequences [12]. To solve this problem different methods have been proposed. One such method is the attention vector. An attention vector is an extra learned vector which points the decoder network towards important parts of the input sequence [12]. Similarly, a coverage vector can be used to make sure that the decoder does not repeat itself often [13]. This repetition can be due to the aforementioned attention vector [14]. Finally a BiLSTM model can also be used.

To explain Bi-directional LSTMs, it is important to know how a normal LSTM works. An LSTM is a network that uses a time dependent hidden state ($c_t$) that keeps track of what the network has seen in the past. The network can choose to add knowledge into the hidden state and choose to forget knowledge. Below is shown that $h_{t-1}$ is an input. This assures that the future sequence is influenced by previous important features. Figure 2 gives a graphical representation of the described LSTM, where small circles denote element wise operations and big circles are layers [15].

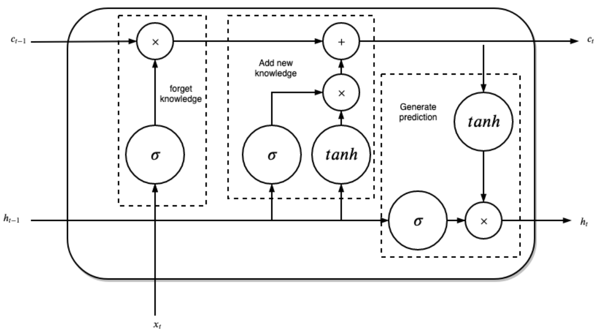

Figure 2: Example LSTM, operation groups are overlaid for clarity.

The value returned at timestep $t$ as the prediction is $h_t$. In order to overcome problems when a sequence starts and stops, special characters are used. These are often denoted as $< sos >$ and $< eos >$ for start and end of sequence respectively.

Figure 2 shows that the output on the right feeds back into the cell of the left. Most publications shown this by repeating the same module. The weights associated with the LSTM are saved only once however [15].

A biLSTM is an LSTM that is fed the input sequence from both directions. The double feeding of input is done because the LSTM long term memory is not as good as on would expect[16]. This technique makes it less likely to 'forget' information, as the information is fed from both sides of the input. The maximum distance the information can go is then halved [16]. Figure 3 shows a graphical representation of how the BiLSTM explained.

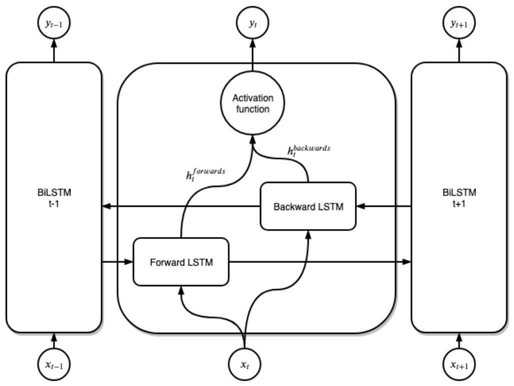

Figure 3: A generic BiLSTM that can be used for sequence2sequence predictions

## 3 Model design

To automatically generate comments, a sequence to sequence model will be used. This has been shown to work effectively for NMT[17], code summarization [1] and comment generation [2] [8]. The general architecture of the network will be discussed in this section; starting with an analysis of the dataset, then the processing of ASTs, followed by the encoding and decoding of input code.

### 3.1 Dataset

To compare the capabilities of the network with the current state of the art, the network is trained on the same dataset used by Hu et al in their deepcom network [2]. The dataset contains 588,108 code-comment pairs that can be used for training and evaluating, to keep it a fair test the same data split as in [2] will be used which is 80% train, 10% test and 10% validation. See table 1 for the particulars.

| # Methods | # All tokens | # All identifiers | #Unique tokens | #Unique identifiers |
|---|---|---|---|---|
| 588,108 | 44,378,497 | 13,779,297 | 794,711 | 794,621 |

Table 1: Statistics for codesnipets in DeepComm dataset

The data set is generated from 9,714 Java projects that have been scraped from Github [2].

Looking at the distribution of the comments (figure 4 we can see that it is extremely heavily biased towards short comments, however there are some comments that are more than 1500 words long. In order to get a better idea of how the data looks we also plot the distribution of only the comments less than 40 words in length in Figure 5.

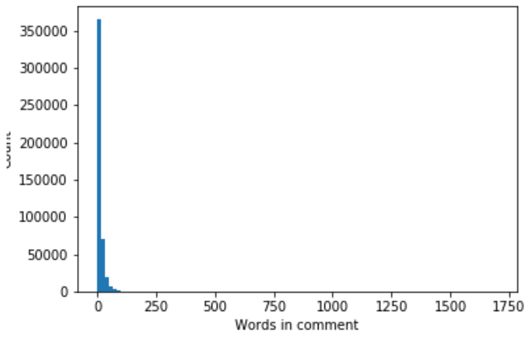

Figure 4: Full distribution of comment lengths.

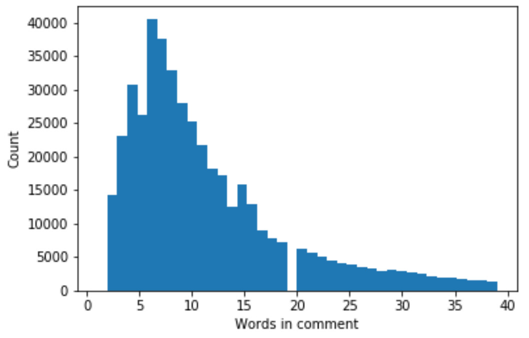

Figure 5: Distribution of comment lengths below 40 words in length

After analysing the distribution we address the Out of Vocabulary problem (OOV). As mentioned before the OOV problem is the inability of an LSTM to predict words it has never seen before. This is because in code variable names are made up and concatenation of different words in camel case or snake case do not matter.

Code summarization solves this by ignoring variable names. If you want to know whether you are dealing with a sorting algorithm, it doesn't matter what the variables are called. In Comment generation, the name of important variables can often be found in the comment. To make comments more natural the choice was made to replace the variables that appear in both the method and the comment by "$VARi$" where $i \in \mathbb{Z}$ and $i$ is the index of the variable. The original variables are saved to a dictionary for every method. After comment prediction using the placeholder variables, these can be swapped out for the corresponding variables in the dictionary.

### 3.2 AST

Luckily, working with code does have some advantages. Compared to more traditional natural language tasks, code contains a clear semantic relationship between the different lines of the code. This relationship can be used when creating an AST which is a graphical representation of the code, each node is a statement and the connections are the relations between the nodes. Figure 6 shows an example AST that is generated from function 3.2, as can be see from the leaves, the names of the variables have been replaced by VAR0 and VAR1, after the comment is generated they can be re-substituted for the correct names.

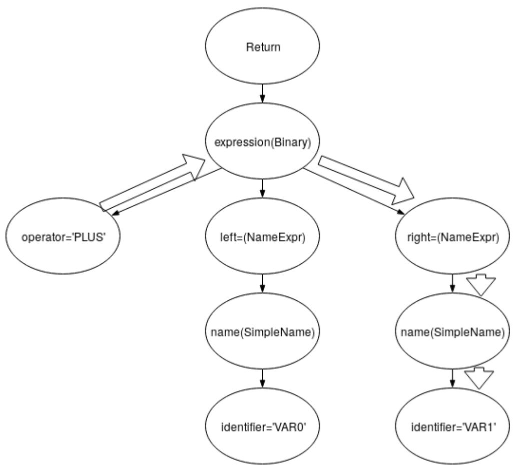

Figure 6: Example AST of Function 3.2, the example path has been superimposed with thick arrows.

```java
public static void add(int VAR0, int VAR1) {
    return VAR0 + VAR1;
}
```

Listing 1: Java example

After the AST generation, this AST is encoded as suggested in [1] by randomly choosing up to $k$ leaf pairs that will represent a path.

An example given the AST in Figure 6, a valid pair would be $\{"Operator =' PLUS'", "identifier = VAR1"\}$. The path belonging to this pair is

$$\{"Operator'PLUS'" \rightarrow "expression(BinaryExpr)" \rightarrow "right(NameExpr)" \rightarrow "name(SimpleName)" \rightarrow "identifier =' VAR1'"\} \quad (1)$$

The arrows in Figure 6 show this path. In general, each code snippet has a set of paths $\mathbb{V} = \{\{v_1, V_2, ..., v_l\}, \{v_1^2, v_2^2, ..., v_l^2\}, \{v_1^k, v_2^k, ..., v_l^k\}\}$ that is randomly sampled before each training session. This enables the model to learn that small differences in code like the use of a while loop or a for loop are not as important. It is re-sampled every time before training in order to ensure that the training is fair[1].

## 3.3 Encoding

Our model is trying to solve a sequence to sequence problem. The architecture used is a encoder decoder structure. To decode to a sequence, first the input sequence needs to be encoded. In this model a Bi-LSTM is being used for both encoding and decoding. Whilst encoding it takes as input sequence a set of paths through the AST [1]. Terminal nodes are split into subtokens which sum up to be tokens, using a learned embedding in the network.

Once this is done there are 3 main elements. From the start node of the path and the end node of the path there are 2 embedded tokens. The path through the AST is also encoded into an embedding. These are combined by concatenating all 3 parts and feeding them in to a fully connected layer which calculates the output of the encoder [1]. A graphical representation of this can be seen in 7.

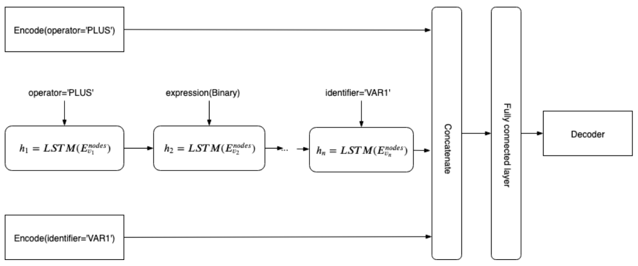
Encode(operator='PLUS')
operator='PLUS'
expression(Binary)
identifier='VAR1'
$h_1 = LSTM(E_{v_1}^{nodes})$
$h_2 = LSTM(E_{v_2}^{nodes})$
$h_n = LSTM(E_{v_n}^{nodes})$
Encode(identifier='VAR1')
Concatenate
Fully connected layer
Decoder

Figure 7: Graphic representation of Encoder, $Encode(x) = \sum_{s \in x} E_s^{\text{subtokens}}$

As can be seen from the described architecture, there is a hard limit to the length of the paths in an AST, which will indirectly limit the length of the functions that can be processed.

## 3.4 Decoding

After passing through the encoders fully connected layer, the embeddings areare passed to another BiLSTM. This BiLSTM transforms the hidden state it receives into a highest probability embedding. This encoded embedding is then decoded to the corresponding human readable word.

To assist the BiLSTM generating a sequence, an attention mechanism is added over the output of the encoder. The attention vector helps the decoder to 'Attend' to important information. This vector is an extra weight vector applied to the output[12].

## 4 Experimental setup

In this section we show the setup of our project. We describe the data used and the network configuration for each of the 2 experiments we conducted. Then the evaluation metric utilized for our experiments and the reason why this is the proper quality measure for our case.

We run the experiments in two folds. For the first one (*Method-1*), we used the architecture of the Code2Seq paper to train a model with the dataset used in the DeepCom paper, while for the second one (*Method-2*) we used the same architecture enhanced with the inclusion of changed ASTs. This task challenging by the process of generating comments instead of the predicting function names is difficult. This difficulty arises from the fact that comments are usually longer than function names. Also, we are generating natural language instead of a sequence of descriptive words. The results of both methods are then compared with the results of the DeepCom[2] as the baseline. For each of these 2 experiments, we calculated the BLEU score as a measure for the quality of the obtained results.

## 4.1 Experiments

For our first experiment (*Method-1*), we modified the Java AST extractor used in Code2Seq. Instead of using the name of the examined function of the Java snippet, we used the comment for this function as target for our model to generate a comment for the snippet instead of a name. Regarding the model hyperparameters, we slightly modified the ones proposed by the Code2Seq paper for the Code Captioning task. We optimized the cross-entropy loss with a Nesterov momentum of 0.95. The learning rate was 0.01 and decayed by 0.05 every epoch. We applied dropout of 0.3 and a recurrent dropout of 0.5 on the LSTM encoding the AST paths. The embeddings size was 128, the same as the batch size. The encoder LSTM had 256 units while the decoder one had 640 units. The reason for the large amount of units in the decoder LSTM is that the target of our network (comments) is long enough. Finally, 200 paths were chosen from each of the extracted ASTs. We trained the model for 100 epochs, while we applied early stopping if no improvement was noticed for 10 epochs. The trainable parameters for this model are 38469376.

When it comes to our second experiment (*Method-2*), we repeated the first experiment with modified ASTs. Instead of using each variable (leaf of the AST) with a generic name (e.g Var1) and its type only, we also added the actual name of the variable [10]. The trainable parameters for this model were 34230144.

## 4.2 Evaluation

The main evaluation metrics for the comments generated by our models was the Bilingual evaluation understudy (BLEU-4) [9]. BLEU measures the quality of a translation (in our case from source code to comments) with a corpora of possible translations. It is widely-used to measure the accuracy of Neural Machine Translation (NMT) [18]. BLEU measures how close the output of the machine is to the one of a human expert. The closer they are, the higher the BLEU score will be [19]. The BLEU score ranges between 0 and 1, with 1 being perfect natural language. BLEU has shown a high correlation with human quality judgement [20] and is one of the most popular metrics to measure the the quality of machine translated text.

The BLEU score for each of the examined snippets is calculated and their average value is computed as our final evaluation result. BLEU calculates the n-gram similarity and uses this to come up with a score according to the equation:

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Where in our case $BP = 1$ as the reference corpus is only 1 comment per code snippet. $w_n$ is a weight vector that sums to 1, $p_n$ is the modified n-gram precision, and N is the max length of n-grams being used in the comparison[9], which in our case is N=4.

We also measured precision, recall, and F1 score as used in [21] and [22] over all the sub-tokens of the target sequence, case insensitive.

# 5 Results

The results of both the methods - Method-1, which uses the code2seq based natural language prediction and Method-2, which used the Method-1 along with modified ASTs (solving out of vocabulary problem). The results will be discussed in terms of the BLEU-4 score, precision, recall and the actual comments generated during testing. For comparing the predicted comments, we used the same java codes as presented in the DeepCom paper [2].

| Approaches | BLEU-4 score |
|---|---|
| DeepCom | 38.17 |
| Method-1 | 6.08 |
| Method-2 | 10.02 |

Table 2: Evaluation results on Java Methods

| Approaches | Precision | Recall | F1 |
|---|---|---|---|
| Method-1 | 36.26 | 21.56 | 27.04 |
| Method-2 | 46.94 | 27.44 | 34.63 |

Table 3: Performance on Java code

Table 2, presents the BLEU-4 scores achieved by the baseline and our Methods-1,2. Table 3 presents the precision, recall and f1 scores of our methods. Finally, table 4 presents the comments generated by our models with respect to the java codes and results presented in DeepCom [2].

Table 2 shows that both our methods fails to improve over the baseline - Deepcom. The poor performance can be due to the imbalanced distribution of target comment lengths in the dataset (as seen in Fig-4, 5) and to the fact that the code2seq based learning architecture (adopted by the methods) was built to predict function names and not natural language. This tends to predict comparatively shorter comments. The evaluation metric (BLEU-4), is 4-gram based comparison technique, and severely punishes our shorter comments. It is also important to note that the Method-2 out performs the Method-1 confirming the effectiveness of solving the problem of *Out of Vocabulary* in ASTs. Similar results are seen in the Table-3 too, where the Method-2 out performs the Method-1. Hence, an important takeaway is that ASTs generally hold the problem of *Out of Vocabulary* affecting the training process, hence, a suitable AST generation technique (e.g. [10] - the one used by Method-2) should be used to avoid such problem.

| Case ID | *Java Method* | *Human-written comments* | *DeepCom comments* | *Method - 1* | *Method - 2* |
|---|---|---|---|---|---|
| 1 | `public static byte[] bitmapToByte(Bitmap b){`<br>`    ByteArrayOutputStream o = new ByteArrayOutputStream();`<br>`    b.compress(Bitmap.CompressFormat.PNG,100,o);`<br>`    return o.toByteArray();`<br>`}` | Convert Bitmap to byte array | Convert Bitmap to byte array | Convert a bitmap to a byte. | Serialize an array of png into. |
| 2 | `private static void addDefaultProfile(App app ,Simple source){`<br>`    if(!source.containsProperty("spring.profiles.active")`<br>`        && !System.getenv().containsKey("ACTIVE")){`<br>`        app.setAdditionalProfiles(Constants.DEVELOPMENT);`<br>`        }`<br>`}` | If no profile has been configured, set by default the "dev" profile. | If no profile has been configured, set by default the "dev" profile. | Checks if the var is active. | If not active, set additional profiles. |
| 3 | `public FactoryConfigurationError(Exception e){`<br>`    super(e.toString());`<br>`    this.exception=e;`<br>`}` | Create a new FactoryConfigurationError with a given Exception base cause of the error. | Create a new &lt;UNK&gt; with a given Exception base cause of the error. | -NA- | -NA- |
| 4 | `protected void createItemsLayout(){`<br>`    if (mItemsLayout == null){`<br>`        mItemsLayout=new LinearLayout(getContext());`<br>`        mItemsLayout.setOrientation(LinearLayout.VERTICAL);`<br>`    }`<br>`}` | Creates item layout if necessary | Creates item layouts if any parameters | Creates the layouts layout. | Creates item layouts if necessary. |
| 5 | `public static void sort(Comparable[] a){`<br>`    int n=a.length;`<br>`    for (int i=1; i < n; i++){`<br>`        Comparable v=a[i];`<br>`        int lo=0, hi=i;`<br>`        while (lo < hi){ ... }`<br>`        ...`<br>`        }`<br>`        assert isSorted(a);`<br>`}` | Rearranges the array in ascending order, using the natural order. | Sorts the array in ascending order, using the natural order. | Sorts the var with the given. | Sort a set of entries based. |
| 6 | `public boolean isEmpty(){`<br>`    return root == null;`<br>`}` | Is this symbol table empty? | Returns true if the symbol is empty. | Returns true if the tree is. | Returns true if the symbol empty. |
| 7 | `public boolean contains(int key){`<br>`    return rank(key) != -1;`<br>`}` | Is the key in this set of integers? | Checks whether the given object is contained within the given set. | Returns true if the key is. | Returns true if this symbol contains. |
| 8 | `public void tag(String inputFileName,String outputFileName){`<br>`    List sentences=jsc.textFile(inputFileName).collect();`<br>`    tag(sentences,outputFileName);`<br>`}` | Tags a text file, each sentence in a line and writes the result to an output file with a desired output format. | Replaces the message with a given tag. | This method is called when executing. | This method is used to process. |
| 9 | `public void unlisten(String pattern){`<br>`    UtilListener listener=listeners.get(pattern);`<br>`    if(listener!=null){`<br>`        listener.destroy();`<br>`        listeners.remove(pattern);`<br>`        }else{`<br>`            client.onError(Topic.RECORD, Event.NOT_LISTENING,pattern);`<br>`        }`<br>`}` | Removes a listener that was previously registered with listenFor - Subscriptions. | It can be called when the product only or refresh has ended. | Removes a var from the topic. | Removes the message from the listener. |

Table 4: Comment Generation Results

Post the preliminary analysis of quantitative results as above, we wanted to analyse the comments generated by our models in a qualitative manner. Such a qualitative analysis is critical, specially for the task at hand (Comment generation), as a qualitative analysis of the comments may explain whether the model has learnt the *syntactic and semantic meaning* from the code to generate comments. For this task, we used the same java codes as used by our baseline paper [2]. This will help us derive unbiased insights in to the comment generation capability of out models (Method-1,2). Table-4, presents these results. From the Table, we see that while compared to the *Human-written* and *DeepCom* comments, the comments of *Method-1,2* are quite shorter, nearly half of its size. As discussed earlier, this could be the reason for the poor BLEU score. Nevertheless, by looking deeper, we see that almost all the comments generated by Method-1,2 has infact learnt the true syntactic and semantic meaning of the java code. In all cases the comments generated by our methods captures the crux of the java code, verifiable using the *Human-written* and *DeepCom* comments. For Example, In Case-1 and 5, we see that our best model (*Method-2*), understands the main concept of the code, i.e. the concept of *image processing* and *sorting* respectively. But it fails to finish the comment and leaves it abruptly, due to the method's inability to predict longer natural language comments. Nevertheless, In some cases like Case-2, 3, 6, 7, it performs perfectly and very similar to the human written comment, sometimes better than the baseline DeepCom itself (i.e in Case 4). Hence, The main conclusion we can draw from these results is that, our best model - *Method-2*, is capable of understanding the *syntactic and semantic meaning* of java codes to generate comments automatically, but suffers from the incapability to generate longer and complete comments.

## 6 Discussion

In the first experiment we conducted we were not able to build a model that can generate proper comments. The generated comments were much shorter than the original ones while they were less descriptive. However, after applying the preprocessing step, in which we included the variable name in the ASTs, we observed a satisfactory increase in the BLEU-4 score achieved. This suggests an increased quality of generated comments. The using of more specifically linked variables in both comment and code resulted in making the comments more accurate.

By visually inspecting the comments generated from our second experiment, it can observe that the generated comments are quite close to the ones written by human. Comments generated by our model tended to be short, which led to a low BLEU-4 score. This issue is caused by the fact that Code2Seq model was built for another purpose, which differs significantly from the comment generation process. The Code2Seq model was built for generating function names which are shorter and not so structured as a comment. Better results may have been achieved if we had the time and the computational resources to experiment with some of the model hyperparameters (e.g. increasing the decoder size). Despite the improvement achieved by the inclusion of the variable name in the ASTs, we believe that a lot of work need to be done upon building a model which can work properly in a real case scenario.

Looking back at the dataset that was used, it may have been beneficial to filter out some of the methods with shorter comments. This because the distribution of comment length is heavily biased towards short comments. It may also have been helpful to filter out methods in general with comments that have a length of less than about 4 words, assuming that for a clear and descriptive comments slightly longer comments are preferred. Re-balancing the dataset so each length has an equal representation would also have been a good idea. This would either require a special form of data augmentation, over-sampling, or under-sampling.

Keeping the contribution of this paper in mind, any future works should work towards improving the models capability to generate longer and complete comments. While this paper concentrated more on the AST generation and encoder modules of the model, future works should deploy a more suitable decoder module for the target natural language comments.

## 7 Conclusion

Automatic comment generation for Java code can be very beneficial. Both for readability of old code, but also better comment suggestions as a programmer aid. With DeepCom as the baseline, this paper focused on replicating the code2seq model with added capabilities such as, predicting natural language (Method-1) and modified ASTs (Method-2). From our results we see that both our methods do not perform as good as the baseline in terms of the BLEU-4 scores. However, Method-2 was much better then our first method. With Method-2 as our best method, an important takeaway is that ASTs generally hold the problem of *Out of Vocabulary* affecting the training process and a suitable AST generation technique (e.g. [10] - the one used by Method-2) should be used to avoid such problem. Nevertheless, looking at the comments generated by our best performing model (Method-2), we see that Method-2 is capable of understanding the *syntactic and semantic meaning* of java codes to generate comments automatically, but suffers from the incapability to generate longer and complete comments.

We would like to thank our teachers Georgios Gousios and Maurício Aniche for their help and suggestions. We would also like to thank all our fellow classmates from the IN4334 class of 2019 for their interest in our presentations, though and interested questions, and their own presentations with good discussions in return.

## 8 Reflection

In retrospect we find that this project was a partial success. From the start we found that it was a very manageable amount of work with a clearly defined goal.

The research phase went well and we quickly had a clear goal of what we wanted to achieve and how we wanted to get there. We are happy how this worked out and feel that everyone in the group has a better understanding of how NMT works as well as a better understanding of the mechanisms used such as the BiLSTM, ASTs and attention mechanisms.

The area that we struggled the most with was the implementation. As we wanted to build upon other research we needed to adapt code that was released along with the papers. This sounded like a good idea at the start. However, after a a few weeks we were having problems with understanding the code written by the original researchers and we spent many hours going through the JavaExtractor code, which ironically was hard to understand as none of it was documented/had comments.

Once we did finally manage to change the JavaExtractor to replace the variables with VARi it was unfortunately rather late in the project from which our training time suffered as we could not do all the runs that we had wanted to. Looking back, it would have been better to start the changing of the JavaExtractor earlier, as we already knew we wanted to do this at the start of the research phase, but chose to focus on research first and only afterwards start with editing/implementing.

Nevertheless, we were able to train two different models (Method- 1,2) and contribute to the automatic generation comments to java codes. An important takeaway from the results is that ASTs generally hold the problem of variable names affecting the training process, hence, a suitable AST generation (e.g. [10] - the one used by Method-2) to avoid such problem. Hence, the main conclusion that we can draw regarding our best model - *Method-2*, is that, our model is capable of understanding the *syntactic and semantic meaning* of java codes regarding the automatic comment generation, but suffers from the incapability to generate longer comments and complete.

## References

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *arXiv preprint arXiv:1808.01400*, 2018.

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

[3] Krishan K Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No. 02CH37318)*, pages 235–241. IEEE, 2002.

[4] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[5] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 101–110. ACM, 2011.

[6] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 71–80. IEEE, 2011.

[7] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389. IEEE, 2015.

[8] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*, Jun 2019.

[9] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing 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.

[10] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156:328 – 340, 2019.

[11] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[13] Abigail See, Peter J Liu, and Christopher D Manning. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368*, 2017.

[14] Jingwen Wang, Wenhao Jiang, Lin Ma, Wei Liu, and Yong Xu. Bidirectional attentive fusion with context gating for dense video captioning. *CoRR*, abs/1804.00100, 2018.

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

[16] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991, 2015.

[17] An Nguyen Le, Ander Martinez, Akifumi Yoshimoto, and Yuji Matsumoto. Improving sequence to sequence neural machine translation by utilizing syntactic dependency information. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 21–29, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing.

[18] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*, 2017.

[19] Kishore Papineni Salim Roukos Todd Ward and John Henderson Florence Reeder. Corpus-based comprehensive and diagnostic mt evaluation: Initial arabic, chinese, french, and spanish results. 2002.

[20] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. Re-evaluation the role of bleu in machine translation research. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006.

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

[22] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.