applied sciences

MDPI

Article

Bi-LSTM-Based Neural Source Code Summarization

Sarah Aljumah and Lamia Berriche *

College of Computer & Information Sciences, Prince Sultan University, Riyadh 12435, Saudi Arabia
* Correspondence: lberriche@psu.edu.sa

Featured Application: Code comment generation.

Abstract: Code summarization is a task that is often employed by software developers for fixing code or reusing code. Software documentation is essential when it comes to software maintenance. The highest cost in software development goes to maintenance because of the difficulty of code modification. To help in reducing the cost and time spent on software development and maintenance, we introduce an automated comment summarization and commenting technique using state-of-the-art techniques in summarization. We use deep neural networks, specifically bidirectional long short-term memory (Bi-LSTM), combined with an attention model to enhance performance. In this study, we propose two different scenarios: one that uses the code text and the structure of the code represented in an abstract syntax tree (AST) and another that uses only code text. We propose two encoder-based models for the first scenario that encodes the code text and the AST independently. Previous works have used different techniques in deep neural networks to generate comments. This study's proposed methodologies scored higher than previous works based on the gated recurrent unit encoder. We conducted our experiment on a dataset of 2.1 million pairs of Java methods and comments. Additionally, we showed that the code structure is beneficial for methods' signatures featuring unclear words.

Keywords: software engineering; neural network; code summarization; software development; software maintenance; deep learning; big data

Citation: Aljumah, S.; Berriche, L. Bi-LSTM-Based Neural Source Code Summarization. *Appl. Sci.* **2022**, *12*, 12587. https://doi.org/10.3390/app122412587

Academic Editors: Robertas Damaševičius, Sanjay Misra and Bharti Suri

Received: 2 November 2022
Accepted: 6 December 2022
Published: 8 December 2022

1. Introduction

Lack of documentation causes cost growth and an extension in a project's schedule. Manual documentation takes extra effort, and it is hard to maintain, causing frustration for developers when making new changes to their code. The automatic generation of comments saves time for developers, and it is effective in terms of simulating human comments. Long short-term memory (LSTM) has shown its effectiveness in text summarization and translation.

In software development and maintenance, software details, such as dependencies, internal structures, integrations, and configurations, must be properly documented in the same source code files. Code comments are essential to software and can help in decreasing the time of software development, better understanding, easy code alteration, better bug detection, and, most importantly, can allow for software reuse. Code comments are one of the most important artifacts to understand for maintenance. A study in [1] showed that artifacts such as literature and architectural models are not as important as code comments. An earlier study has shown that, on average, developers spend 60% of their time on program understanding [2]. A commented code is easier to understand than an uncommented code. Therefore, developers save much time by reading the code description to find key information needed for a code change or maintenance. The absence of code comments can greatly decrease software quality and maintainability.

However, because of the dynamic nature of software projects and project management's tight schedules, source code documentation is usually ignored. Code documentation

is time-consuming and difficult to write effectively. To address this issue, many studies have introduced the idea of automated source code documentation [1–7]. Automatic source code documentation can significantly improve software development in terms of speed, quality, easy code handovers, and code reusability.

An important component of automated documentation is source code summarization. The source code summarizer generates natural language sentences that explain the functionality of the source code. Summaries such as "updates the log files in the server" or "generates a serial number from a random function and timestamp" can give a clear picture of how the code works and save the developer from tracing 20+ lines of code.

Source code summarization is an expanding research area. Developers are often blamed and noted for neglecting the manual effort of code commenting [4,8]. Recent studies on code summarization have proved that high-quality, consistent comments can be generated automatically with deep neural networks that are trained on a large corpus [2,5]. These approaches lack the usage of context and domain knowledge, and they only rely on syntax or language structure [9,10]. Certain approaches, such as [11,12], retrieve the code comments from a similar code. Some approaches, such as that discussed in [13], investigated the use of the two-input information, lexical and syntactic, with a gated recurrent unit (GRU) encoder–decoder. In [14], authors combine both retrieval-based and bidirectional GRU with two inputs. The Bi-LSTM showed better performance than a simple LSTM in Python [15,16].

Our proposed solution aims to study the impact of code's structural features when a Bi-LSTM attentional encoder–decoder model, with two encoders, is applied to the Java method's comment generation. We propose two methods based on neural networks for generating summaries for source code. The two methods use different types of information to compare which information can enhance the performance of the prediction.

In Section 2, we will show the general framework of deep neural network-based code comment generators. In Section 3, we will present the related works. Then, in Section 4, we will describe our methodology. Afterward, in Section 5, we will present our implementation details and discuss our results. Finally, in Section 6, we will summarize our work and give some recommendations for future works in the conclusion.

2. General Framework

2.1. Preprocessing

2.1.1. Cleaning

To ensure the accuracy of the model, we perform data cleaning. In this step, both the comments and the code snippets are cleaned. This is performed by removing the following:

- Comment symbols such as '//';
- Annotations in the code or comment, such as @Transactional, @Service, etc.;
- Keywords from the code snippets that are not helping the meaning, such as public, abstract, final, try, catch, static, etc.

2.1.2. Word Embedding

Word embedding is a technique for representing a document's vocabulary. This technique captures the semantic and syntactic similarities and relations between words and the context of a document. Word embedding represents a word as a vector. Word2Vec is a learning technique that takes a text corpus as input and produces an output of vectors that represent each word in terms of the probability of its appearance.

There are two techniques in Word2Vec: a continuous bag of words (CBOW) and skip-gram. Skip-gram loops through the words in a sentence and uses the current word to predict the next word. However, in CBOW, the surrounding words are used to predict a word in the middle.

2.2. Recurrent Neural Networks

A recurrent neural network (RNN) is a neural-based approach that is effective in processing sequential data. The RNN works recursively on the input and applies computations to all instances of the input. These are represented in a vector of tokens. The main point of the RNN is that it can memorize the result of the previous computation and apply it to the current computation. The RNN is a class of artificial neural networks in which connections between the nodes represent a directed graph such that information flows from a layer to a previous layer and allows the information to flow back into the previous parts. The RNN not only uses the input information but also whatever it is fed during the training of the network [17]. The RNN has problems such as the vanishing gradient, often called the long-term dependency problem; this problem means that the network is not able to hold information for a long time, and it rapidly loses information over time. To solve the vanishing gradient issue, two neural network approaches have been introduced: first, the gated recurrent unit (GRU), and second, LSTM. The two versions provide a memory cell to store information on previous inputs. The gates are used for controlling the flow in the network. These gates can determine what information is important and then store that information in the memory unit. The gates can pass the stored information and use it to make better predictions [18].

(1) Gated Recurrent Units

The GRU works like the RNN in terms of workflow but differs in the operations inside the GRU. The GRU has two gates: one is the reset gate, and the other is the update gate. The update gate decides if the current cell state should be updated with the current activation value or not. The reset gate decides if the previous cell state is important or should be removed [19].

(2) Long-Term Short-Term Memory

As we mentioned, in RNN, one of the challenges is its short-term memory; it is hard to carry the output from one step to the next steps for long sequences. To solve this issue, we use the LSTM, a kind of RNN that keeps in its memory the information for long periods. LSTM has three steps; first is the input gate, then the forget gate, and lastly, the output gate. The input gate decides what information is to be added to the cell state from the input, the forget gate decides which data to forget and which data to remember, and the output gate decides which parts in the current cell go to the output [20].

(3) Encoder–Decoder Architecture

Sequence-to-sequence (seq2seq) prediction problems often involve predicting the next value in a real-valued sequence or a label for an input. One of the challenges in sequence-to-sequence prediction problems is that the length of the input sequences and the output sequences may vary; this problem is called a many-to-many prediction problem. To solve the problem in seq2seq, the encoder–decoder architecture has been proposed. The architecture is composed of two components: the encoder, which is responsible for reading the input sequences and encoding them into a fixed length vector, and the decoder, for decoding the fixed length vector back to a variable length vector and then predicting the output sequence [21].

3. Related Works and Motivations

3.1. Related Works

In this section, we show some of the existing approaches in source code comment generation. In [22], the authors presented CODE-NN, which uses LSTM and neural attention to model the source code. They created a dataset of C# and SQL language codes from Stack Overflow. Their approach achieved a 20.5% bilingual evaluation understudy (BLUE-4) for C# and 18.4% for SQL.

Hu et al. [23] considered the code summarization problem as a machine translation problem. Their approach translated Java source code to comments written in natural

language. They proposed DeepCom, which is an attention-based LSTM encoder–decoder. Abstract syntax tree (AST) was used as an input to DeepCom after converting it to a formatted sequence using a structure-based traversal (SBT). Their solution scored 38.17% in terms of BLUE-4 on a dataset of 69,708 Java methods.

Wan et al. [15] worked on two problems: one was the code representation, and the other was exposure bias. In their methodology, they first encoded the structural and the sequential content of the source code using AST-based LSTM and hybrid attention for integration. Then they fed the code vector into deep reinforcement learning. Their model achieved 4.41% BLEU-4 with a Python dataset of 108,726 code snippets.

Hu et al. [24] proposed a model named TL-CodeSum that made use of API knowledge and Java methods. Their approach consisted of three main parts: first, the data processing, then the training, and finally, the online code summary generation. The model performed two tasks: first, the API sequence summarization, and second, the source code summarization task. The API summarization task built a mapping between API knowledge and the description of its functionality; then, this knowledge was applied to the code summarization task. Both tasks were built using deep neural networks; specifically, they used GRU for the encoder and decoder. They used 340,922 pairs (API sequence, summary) and 69,708 tuples (API sequence, code, summary). Their work showed an impressive result of 41.98% BLEU.

Chen et al. [25] presented a bi-variational autoencoder named BVAE for source code summarization. They introduced two instances of BVAE: one for code retrieval and the other for code summarization. Their framework allowed bidirectional mapping between source code and its natural language summary. They tested their model on C# and SQL. BVAE gave a 20.9% BLEU-4 in C# and 19.7% in SQL.

Shido et al. [26] proposed a framework consisting of three components: parsing code into ASTs, encoding the ASTs, and decoding the generated sequences with attention. First, they converted each code snippet into an AST using a standard AST parser. Each node in the tree was then embedded into a vector of a fixed length. The AST with labeled nodes was then encoded by a multi-way tree-LSTM. Then, finally, they decoded the encoded vectors into sentences using an LSTM attention-based decoder. They ran their experiments on a dataset of 588 108 Java code comment pairs. They achieved a 20.4% BLUE-4.

In [13], the authors proposed a model that took input code and AST separately, which allowed it to learn code structure independently of the text in code. They used the attentional encoder–decoder with two GRU encoders: one for text and the other one for AST. They evaluated their technique with a dataset that they created from 2.1 million Java methods. They compared a text-only input model to the text/AST input model. They found that the performances of the two models were close to each other: 19.6% BLEU for the AST-based model and 19.4% for the text-only-based model.

Ye et al. [16] presented an end-to-end model for code retrieval and summarization named CO3. CO3 used dual learning and multi-task learning. They worked on finding the correlation between code summarization in natural language and code written in a programming language. CO3 had an effective architecture that consisted of only two Bi-LSTM instances: one for code summarization and the other one for code generation. CO3 gave an 11.9% BLEU-4 in SQL and 8.5% in Python.

Zhou et al. [27] proposed a model that made use of the encoder–decoder architecture. They used two encoders: one for representing lexical information and the other one for representing the syntactical structure of the code. The model went through three stages: data preprocessing, data representation as an AST, and finally, summary generation. The lexical encoder was an RNN, and the syntactical encoder was a tree-RNN representing the AST. The output from the two encoders was used to generate the code summary using a switch network. They applied their model on a dataset of 588,108 Java code comment pairs extracted from 9714 GitHub projects. They achieved 17.04% BLEU-4.

Li et al. [28] proposed the Hybrid-DeepCom and made use of the structural and lexical information from the Java methods for better code comment generation. They had two

encoders; the first one was the code encoder, which was a GRU, and the second was the AST with an SBT traversal encoder. Then they used an attention model to pay attention to words more than others. The authors did not report their results.

In [29], the authors presented a transformer-based encoder–decoder model ComFormer. They used byte-pair-encoding to reduce the out-of-vocabulary problem, where the UNK token is produced for words with low occurrence in the code. In addition, they used a simplified version of the SBT. They applied their framework on a code-only input and code-plus-AST input. They considered three approaches for the two inputs case: the joint encoder, the shared encoder, and the single encoder. The joint encoder uses two separate encoders for code and AST inputs. In the shared encoder approach, the code and the AST share the same encoder after a distinct embedding process. In the single-encoder approach, code and AST undergo single embedding and encoding processes. They tested their technique on a dataset of 588,108 Java methods. They found that their framework reached 43.8% BLEU-4 for a code-only input and 48.437% for a code and AST input with a single-encoder approach.

In [30], the authors proposed a transformer-based encoder–decoder architecture with three inputs: the code, the AST, and the API. They applied a joint encoder architecture. They conducted their experiments on 137,007 Java methods with their comments collected by Hussain et al. in [31]. They compared the transformer-based encoder–decoder to a GRU-based encoder–decoder. They showed that the transformer-based architecture outperforms the GRU-based architecture. They also showed that for both architectures adding AST information decreases the model performance while adding API knowledge increases slightly the BLEU score. Their best BLEU-4 score was 5.28%.

3.2. Motivation

Java code summarization techniques showed various performances depending on the size of the dataset, the type of encoder input, the number of encoders, and the encoder-decoder implementation. A comparison between the comment generation techniques for Java code is provided in Table 1.

Table 1. Comparison of Java code comment generation methods.

Related Work	Year	Input Information	Neural Network	Encoders Number	Dataset Size	Result (BLEU)
[23]	2018	AST	LSTM	1	69,708	38.17%
[26]	2019	AST	LSTM	1	588,108	20.4%
[13] ast-attendgru	2019	Code + AST Code	GRU GRU	2 1	2.1 millions	19.6% 19.4%
[27]	2020	Code + AST	RNN and tree-RNN	2	Java 588,108	17.04%
[29] ComFormer	2021	Code Code + AST	Transformer-based Encoder–Decoder	1 1	Java 588,108	43.801% 48.437%
[30] API2Com	2021	Code Code + AST Code + API Code + AST + API	Transformer-based Encoder–Decoder	1 2 2 3	Java 137,007	5.27% 5.26% 5.28% 5.26%

It was noted that the LSTM with a single AST input in [15] outperformed an RNN with both code and AST inputs on the same dataset in [27]. Additionally, the GRU neural networks [13] with both code and AST inputs outperformed the RNN with two inputs in a small dataset [27]. On the other side [29], AST increased the BLEU score in a transformer model with a single encoder and slightly decreased the BLEU score in a double encoder architecture [30]. The main disadvantage of the transformer models is their inefficiency in processing long sequences [32].

In addition, in [16], Python code comment generation with a Bi-LSTM encoder–decoder showed an improvement over an LSTM-only encoder–decoder proposed in [15]. Bi-LSTM may extend the LSTM capabilities by training the input in forward and backward directions.

In this work, we study the effect of the addition of the AST information to the code information on the performance of a Bi-LSTM double encoder structure for Java code comment generation, and we compare it to the GRU model in [13].

4. Methodology

In this paper, we propose two neural models for source code summarization for Java methods based on a bidirectional LSTM with an encoder–decoder architecture and an attention mechanism. The first model, model 1, uses two types of information in source code: representation of source code as text and representation of code as an AST, shown in Figure 1. The second model, model 2, uses only one type of information, which is the code represented as text, as shown in Figure 2.

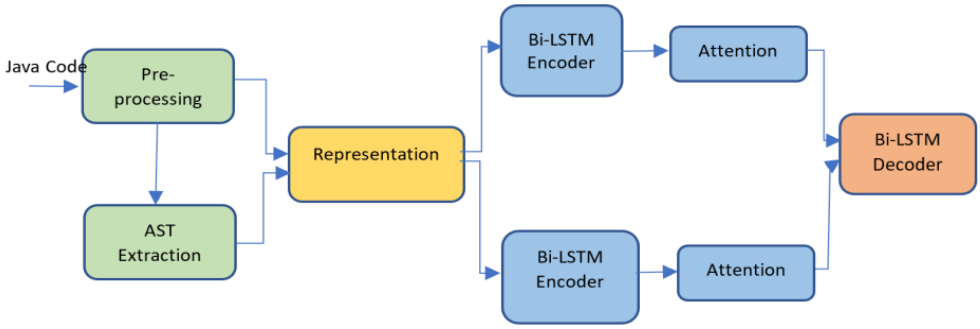

Figure 1. Model 1.

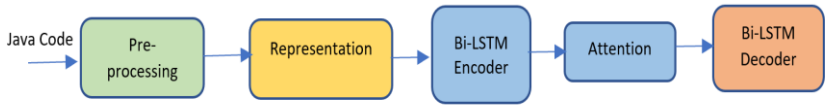

Figure 2. Model 2.

Our framework starts with a preprocessing stage followed by an AST representation and then one hot vector encoding for vectorization. Afterward, the Bi-LSTM encoder is followed by an attention model and a Bi-LSTM decoder.

In the preprocessing phase, we remove auto-generated comments, non-English comments, and underscores and convert everything to camel case. Next, we represent the words in the input data as vectors using one hot vector encoding technique. These vectors are fed to the encoder–decoder model. The encoder reads the input sequences and summarizes the information in a hidden state vector. In every input token, the encoder collects any relevant information and produces a hidden state. Bi-LSTM layered units mix the input with the current state and produce the output. The output vector is the last hidden state produced from the last LSTM unit. The context vector from the encoder is then fed to the attention model. The attention model gives different weights to tokens to give attention to some tokens more than others. The output from the attention model is then used as input to the decoder. The decoder is a layered Bi-LSTM, and each unit generates a hidden state. The decoder calculates the probability for every token and generates the output sequence.

In model 1, we perform all the mentioned steps in the first method; however, the only difference is that we use two encoders instead of one. The first is for the code, and the second is for the AST. In addition, the outputs from the attention models of the code Bi-LSTM encoder and the AST encoder are used as inputs to the same decoder. An example of the input and output is shown in Figure 3.

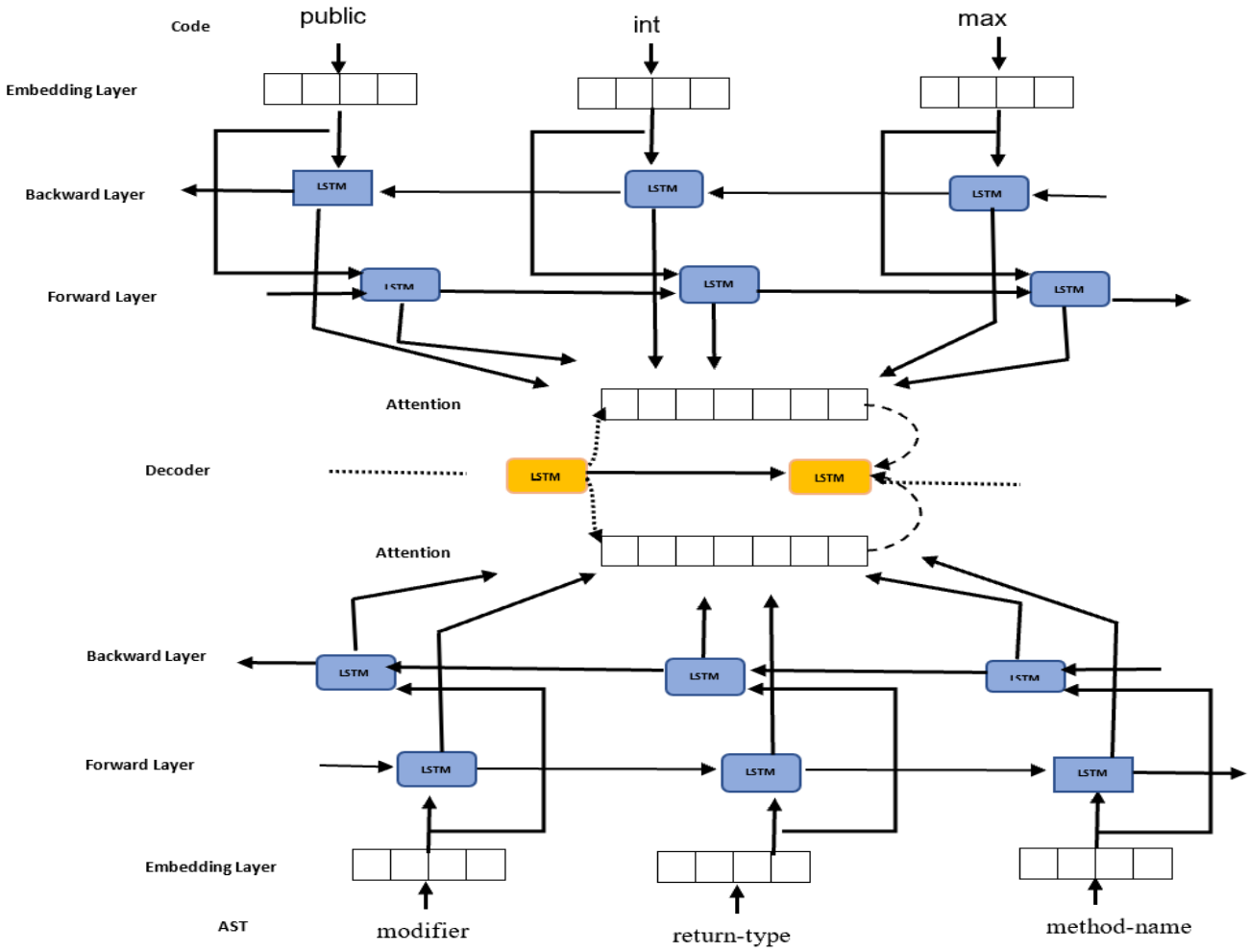

Figure 3. Model 1 example.

4.1. Preprocessing and Tokenization

In the preprocessing phase, we filter the methods preceded by/∗ ∗. Then we find the first sentence to represent the comment. We remove any non-English comments and any auto-generated comments by looking for the "generated by" in the comment. Then we remove the underscores and non-alpha characters, and we set all characters to lowercase. We do not perform any stemming. For representing the AST, we remove every word that is not part of the official Java APIs and replace them with a special token. We represent the code as an AST. The AST represents all syntax-related elements of code snippets. The tree focuses on language rules. Since AST is highly coupled with how a compiler is built, ASTs are generated using language-specific AST parsers such as eclipse AST Parser for Java. After ASTs are obtained, they are traversed using an SBT proposed by Moreno et al. [2]. The SBT traverses the AST to produce a flat representation of the tree. After preprocessing and representing the code as an AST, we encode them using one hot encoding technique. This technique uses a binary form, and it is one of the simplest techniques used in encoding. The whole sequence is represented as zeros except for the index that represents the word.

4.2. Encoder–Decoder Architecture

Our framework has an encoder–decoder architecture; this structure is used in deep neural network-based code comment generation. The encoder plays the role of encoding the input code/text, comment, and the flattened AST vectors into a fixed length, and the decoder is responsible for decoding the source code vector and predicting the corresponding comment. In our framework, we use Bi-LSTM for both the encoder and decoder.

The Bi-LSTM duplicates the first recurrent layer, and we obtain a side-by-side layer; then, the first layer takes an input sequence (code vector, flattened AST vector, comment vector) as it is, and the second layer takes the reversed copy sequence. We provide the sequence bidirectionally since it was used in the speech recognition domain, proving that the context of the whole speech is important in interpreting what is said rather than just using a linear interpretation.

4.2.1. Single-Encoder and Multiple-Encoder-Based Comment Generation

In this research, we used both single- and multiple-encoder-based comment generation. In single-encoder-based code summarization algorithms, the encoder consists of only one Bi-LSTM, which processes only one type of information, which is text or code. Multiple-encoder-based code summarization techniques consist of multiple encoders in an encoder–decoder architecture. Each encoder extracts one type of information from the code. Multiple-encoder algorithms produce more accurate code summaries. In our paper, we use two encoders: one that uses the AST to represent the structural information of the code and the other decoder to represent the code/text.

4.2.2. Decoder

The decoder starts with the final state from the encoders of code/text and AST. The decoder converts the final state vector outputted from the encoder into a different arbitrary length vector. The sequence inputted to the encoder is code or AST, and the sequence outputted from the decoder is the comment corresponding to the code/AST.

4.3. Attention

We add the attention mechanism between the encoder and decoder. The attention model assigns higher weights to relevant tokens in the input sequence to the decoder. This improves the performance in the case of long sequences.

4.4. Evaluation

To evaluate the results after the prediction phase, we use the BLEU evaluator. BLEU measures the similarity between the predicted summary and the human written summary. The results of all sentences are summed, and then the average over the entire corpus is used to estimate the quality of the translation. We use the composite BLEU, and we refer to it as BLEUn, where n is the length of the sequences. It measures the similarity of the n-length sequence against a human-written summary.

5. Implementation and Results

In this section, we will first describe our implementation details, the dataset, and the experimental results.

5.1. Implementation Details

To implement the models, we used Anaconda as a platform for package management for Python. We used the Keras library in Python to create Bi-LSTMs via the bidirectional layer wrapper. The wrapper takes the first LSTM layer as an input argument. Keras allows for specifying the way of merging the forward and backward outputs before being passed on to the next layer. We used Google Cloud to run the code and load the data. We created an instance with the following specifications:

- Machine Type: n1-standard-16 (16 vCPUs, 60 GB memory);
- GPUs: 1 × NVIDIA Tesla P100 Virtual Workstation

For our two proposed methods, we went through three phases: first, the training phase, second, the prediction phase, and finally, the evaluation phase. In the training phase, we set the batch size to 32 and the number of epochs to 100. For this paper, we used the same parameter settings as in [13]. Then, we loaded the tokenizers, sequences, and data. We created two models, as mentioned previously in the methodology section. Then, for

the attention mechanism, we took the dot product of the decoder output with the encoder output. The shapes of the encoder and decoder were batch_size, 100, 256, and batch_size, 13, 256, respectively. The 100 and 13 represent the code/AST and the comment lengths. The 256 represents the number of recurrent layers.

5.2. Dataset

We used a corpus that consisted of Java methods; this corpus was collected by LeClair et al. [11]. Their dataset was collected from a repository containing 51 million Java methods from more than 5000 projects. They prepared the repository to create the dataset for the code summarization problem. They organized the extracted and Java methods into an SQL database. Then they filtered the methods that were preceded by/∗ ∗. Then they tried to find the first sentence by looking for the first period, or newline, and then they extracted that sentence. Then they used a language detect library to remove non-English comments. They removed any methods with comments containing the word "generated by" to reduce any auto-generated comments. The resultant dataset consisted of about 2.1 million pairs of methods and comments. They removed underscores and non-alpha characters and set them to lowercase, then split code and comments on camel case. They did not perform any stemming. To obtain the AST, they first used srcML [25] to obtain an XML representation of the methods. Then they converted the XML into a flattened SBT 5representation. They also replaced all words in the code (except official Java classes) with a special token. We split the dataset randomly into 90% for training, 5% for validation, and 5% for testing. The results of the generated code summaries are given in Table 2.

Table 2. Examples of generated code comments.

Code	Human Comment	BiLSTM + Code + ST	BiLSTM + Code
public string obtain header field string name return response header obtain header name	returns the name of the specified header field	<s> returns the header field value </s> <NULL> <NULL> <NULL> <NULL> <NULL>	<s> returns the value of the specified field as a string </s> <NULL>
public string obtain string a name parse string s string parameters obtain a name if s null s length 0 return s 0 return null	returns the first parameter value	<s> returns the string value of the given parameter </s> <NULL> <NULL> <NULL>	<s> returns the string value of the given parameter </s> <NULL> <NULL> <NULL>
public void read string a response line throws http exception parse response line a response line	reads a response line from a string	<s> reads a response line from the client </s> <NULL> <NULL> <NULL> <NULL>	<s> reads a response line from the http server </s> <NULL> <NULL> <NULL>

5.3. Results and Discussion

We evaluated our models using BLEU. BLEU measures the similarity of the machine-translated text to a reference text. We used three cumulative BLEU scores (BLEU-1 (B1), BLUE-3 (B3), and BLEU-4 (B4)) in addition to the composite B score. Bn refers to the calculation of individual n-gram scores at all orders from 1 to n and weighing them. The n-gram score is the evaluation of matching grams. We measured the results after taking the predicted summaries as vectors, and then we converted them to sentences to evaluate the prediction and obtain the BLEU result.

After evaluating our models, we obtained 38.42% BLEU-1 for Bi-LSTM using code only and 38.96% BLEU-1 for Bi-LSTM using code and AST. The results for both models were very close, and the reason for that was that most of the training methods did not have complex structures and were very similar in the generated syntax tree.

We compared our models with the model in [13], where a GRU neural network was used. The comparison is shown in Table 3.

Table 3. Comparison with [13].

		B3	B4	B
[13] ast-attendgru	GRU + Code	14.8%	11.3%	19.4%
	GRU + Code + AST	14.9%	11.4%	19.6%
Our work	Bi-LSTM + Code	15.06%	11.64%	19.61%
	Bi-LSTM + Code + AST	14.99%	11.45%	19.62%

As shown in Table 3, our method scored higher than [13] for B3, B4, and B scores. The reason for this was the use of Bi-LSTM. Bi-LSTM uses all the available information; it uses the past and future information in a specific framework. Since our problem falls under text summarization and text translation, it has been shown that the use of past and future information enhances the quality of prediction. We also noted that the models with AST and without AST gave close BLEU results similar to those in [13]. This could be explained by the fact that summaries could be implied from method signatures in some cases, but it could benefit from AST in case the method text is not clear. The generated comments (shown in Table 2) from our methodologies are very similar to human comments and give the exact meaning when the method is short. Model 2 makes use of only the code, and it is useful if summaries are related to method functionality more than the code logic. Model 1 makes use of the syntax along with the code; this increases the chance of predicting the summary when it is about the code logic more than its functionality and what it does. Both models use Bi-LSTM, which was used in many problems in machine translation because of its good performance and how it considers past and future information.

The BLEU score of this work and previous works is still low and needs enhancements. This work and previous works do not consider the context of the methods. For example, a method named calculateTotal() in a class named Bill has a different meaning from a calculateTotal() method in a class named Grade. Considering the place and context of the method may help in generating better summaries. In addition, we did not look at the caller methods and their summaries; this information may help developers in tracing their code effectively.

6. Conclusions

Source code summarization is critical when it comes to maintenance and code fixes. Developers find it difficult to write documentation given the tight schedules. This research contributes to solving this issue by automating code summarization. The existing frameworks for code summarization, such as Javadoc, only use the method signature to generate comments. However, our methodologies use deep learning and existing code summaries written by humans to generate comments that are human-like and describe the core functionality of the code rather than just the signature. Similar works exist where they use deep learning techniques to generate code summaries. We propose two methodologies using deep learning. The first method uses only one type of information, which is the code information, and the second uses two types of information: one is the structure represented in an AST, and the other information is the code. We use the encoder–decoder architecture and an attention model to enhance the quality of predicting long sequences. The encoder and decoder are both built using Bi-LSTM.

We applied our model to a dataset consisting of 2.1 million Java codes and comment pairs written by a human. Our research scored higher than [13], where the authors' model was based on GRU, and like [13], we found that AST information does not improve the model performance in all situations. Improvement depends on the method signature's clarity of words. This result is similar to the findings of [13,30]. In addition, BiLSTM gives slightly better results than GRU.

This paper should be enhanced to support multiple programming languages. It is suggested to include extra encoders that extract and use other source code features other

than syntax and text. We suggest using the whole Java class instead of just the method to obtain a better context of the targeted method. We also suggest that we turn this work into a plugin in existing development environments such as Eclipse and IntelliJ.

Author Contributions: Conceptualization, S.A. and L.B.; methodology, S.A. and L.B.; software, S.A.; validation, S.A.; formal analysis, S.A. and L.B.; writing—original draft preparation, S.A.; writing—review and editing, L.B.; supervision, S.A. and L.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Prince Sultan University. The APC was funded by Prince Sultan University.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. De Souza, S.C.B.; Anquetil, N.; de Oliveira, K.M. A study of the documentation essential to software maintenance. In Proceedings of the 23rd Annual International Conference on Design of Communication Documenting & Designing for Pervasive Information—SIGDOC'05, Coventry, UK, 21–23 September 2005. Available online: https://www.academia.edu/18057445/A_study_of_the_documentation_essential_to_software_maintenance (accessed on 2 December 2021).
2. Moreno, L.; Aponte, J.; Sridhara, G.; Marcus, A.; Pollock, L.; Vijay-Shanker, K. Automatic generation of natural language summaries for Java classes. In Proceedings of the IEEE International Conference on Program Comprehension, San Francisco, CA, USA, 20–21 May 2013; pp. 23–32. [CrossRef]
3. Mcburney, P.W.; Mcmillan, C. Automatic Documentation Generation via Source Code Summarization of Method Context. 2014. Available online: http://www.nd.edu/~pmcburne/summaries/ (accessed on 2 December 2021).
4. Roehm, T.; Tiarks, R.; Koschke, R.; Maalej, W. How do professional developers comprehend software? In Proceedings of the International Conference on Software Engineering, Zurich, Switzerland, 2–9 June 2012; pp. 255–265. [CrossRef]
5. Hellendoorn, V.J.; Devanbu, P. Are deep neural networks the best choice for modeling source code? In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; Volume F130154; pp. 763–773. [CrossRef]
6. Zhao, L.; Zhang, L.; Yan, S. A Survey on Research of Code Comment Auto Generation. *J. Phys. Conf. Ser.* **2019**, *1345*, 032010. [CrossRef]
7. Zhang, C.; Wang, J.; Zhou, Q.; Xu, T.; Tang, K.; Gui, H.; Liu, F. A Survey of Automatic Source Code Summarization. *Symmetry* **2022**, *14*, 471. [CrossRef]
8. Kajko-Mattsson, M. A Survey of Documentation Practice within Corrective Maintenance. *Empir. Softw. Eng.* **2004**, *10*, 31–55. [CrossRef]
9. Wong, E.; Yang, J.; Tan, L. AutoComment: Mining question and answer sites for automatic comment generation. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, 11–15 November 2013; pp. 562–567. [CrossRef]
10. Rahman, M.M.; Roy, C.K.; Keivanloo, I. Recommending insightful comments for source code using crowdsourced knowledge. In Proceedings of the2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, 27–28 September 2015; pp. 81–90. [CrossRef]
11. Wei, B. Retrieve and refine: Exemplar-based neural comment generation. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, 11–15 November 2019; pp. 1250–1252. [CrossRef]
12. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Liu, X. Retrieval-based neural source code summarization. In Proceedings of the International Conference on Software Engineering, Seoul, Republic of Korea, 26–28 June 2020; pp. 1385–1397. [CrossRef]
13. Leclair, A.; Jiang, S.; McMillan, C. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In Proceedings of the International Conference on Software Engineering, Montreal, QC, Canada, 25–31 May 2019; pp. 795–806. [CrossRef]
14. Zhang, C.; Zhou, Q.; Qiao, M.; Tang, K.; Xu, L.; Liu, F. Re_Trans: Combined Retrieval and Transformer Model for Source Code Summarization. *Entropy* **2022**, *24*, 1372. [CrossRef]
15. Wan, Y.; Zhao, Z.; Yang, M.; Xu, G.; Ying, H.; Wu, J.; Yu, P.S. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018.

16. Ye, W.; Xie, R.; Zhang, J.; Hu, T.; Wang, X.; Zhang, S. Leveraging Code Generation to Improve Code Retrieval and Summarization via Dual Learning. In Proceedings of the Web Conference 2020-Proceedings of the World Wide Web Conference, WWW 2020, Taipei, Taiwan, 20–24 April 2020; pp. 2309–2319. [CrossRef]
17. Young, T.; Hazarika, D.; Poria, S.; Cambria, E. Recent trends in deep learning based natural language processing. *IEEE Comput. Intell. Mag.* **2018**, *13*, 55–75. [CrossRef]
18. Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, Sardinia, Italy, 13–15 May 2010.
19. Chung, J.; Gülçehre, Ç.; Cho, K.; Bengio, Y. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv* **2014**, arXiv:1412.3555.
20. Hochreiter, S.; Schmidhuber, J. Long Short-Term Memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
21. Cho, K.; Van Merriënboer, B.; Gulcehre, C.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Proceedings of the EMNLP 2014—2014 Conference on Empirical Methods in Natural Language Processing, Doha, Qatar, 25–29 October 2014. [CrossRef]
22. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing Source Code using a Neural Attention Model. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016—Long Papers, Berlin, Germany, 7–12 August 2016; Volume 4, pp. 2073–2083. [CrossRef]
23. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In Proceedings of the International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018. [CrossRef]
24. Hu, X.; Li, G.; Xia, X.; Lo, D.; Lu, S.; Jin, Z. Summarizing source code with transferred API knowledge. In Proceedings of the IJCAI International Joint Conference on Artificial Intelligence, Stockholm, Sweden, 13–19 July 2018; pp. 2269–2275. [CrossRef]
25. Chen, Q.; Zhou, M. A neural framework for retrieval and summarization of source code. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018. [CrossRef]
26. Shido, Y.; Kobayashi, Y.; Yamamoto, A.; Miyamoto, A.; Matsumura, T. Automatic Source Code Summarization with Extended Tree-LSTM. In Proceedings of the International Joint Conference on Neural Networks, Budapest, Hungary, 14–19 July 2019. [CrossRef]
27. Zhou, Z.; Yu, H.; Fan, G. Effective approaches to combining lexical and syntactical information for code summarization. *Softw. Pract. Exp.* **2020**, *50*, 2313–2336. [CrossRef]
28. Li, B.; Yan, M.; Xia, X.; Hu, X.; Li, G.; Lo, D. DeepCommenter: A deep code comment generation tool with hybrid lexical and syntactical information. In Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, 8–13 November 2020; Volume 20, pp. 1571–1575. [CrossRef]
29. Yang, G.; Chen, X.; Cao, J.; Xu, S.; Cui, Z.; Yu, C.; Liu, K. ComFormer: Code Comment Generation via Transformer and Fusion Method-based Hybrid Code Representation. In Proceedings of the 2021 8th International Conference on Dependable Systems and Their Applications, DSA 2021, Yinchuan, China, 11–12 September 2021; pp. 30–41. [CrossRef]
30. Shahbazi, R.; Sharma, R.; Fard, F.H. API2Com: On the Improvement of Automatically Generated Code Comments Using API Documentations. In Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 20–21 May 2021; pp. 411–421. [CrossRef]
31. Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; Brockschmidt, M. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. September 2019. Available online: http://arxiv.org/abs/1909.09436 (accessed on 2 December 2021).
32. Lin, T.; Wang, Y.; Liu, X.; Qiu, X. A Survey of Transformers. *AI Open* **2022**, *3*, 111–132. [CrossRef]