ATOM: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking

Shangqing Liu, Cuiyun Gao, Sen Chen, *Member, IEEE*, Lun Yiu Nie, and Yang Liu

Abstract—Commit messages record code changes (e.g., feature modifications and bug repairs) in natural language, and are useful for program comprehension. Due to the frequent updates of software and time cost, developers are generally unmotivated to write commit messages for code changes. Therefore, automating the message writing process is necessitated. Previous studies on commit message generation have been benefited from generation models or retrieval models, but the code structure of changed code, i.e., AST, which can be important for capturing code semantics, has not been explicitly involved. Moreover, although generation models have the advantages of synthesizing commit messages for new code changes, they are not easy to bridge the semantic gap between code and natural languages which could be mitigated by retrieval models. In this paper, we propose a novel commit message generation model, named ATOM, which explicitly incorporates the abstract syntax tree for representing code changes and integrates both retrieved and generated messages through hybrid ranking. Specifically, the hybrid ranking module can prioritize the most accurate message from both retrieved and generated messages regarding one code change. We evaluate the proposed model ATOM on our dataset crawled from 56 popular Java repositories. Experimental results demonstrate that ATOM increases the state-of-the-art models by 30.72 percent in terms of BLEU-4 (an accuracy measure that is widely used to evaluate text generation systems). Qualitative analysis also demonstrates the effectiveness of ATOM in generating accurate code commit messages.

Index Terms—Commit message generation, code changes, abstract syntax tree

1 Introduction

With software growing in size and complexity, code hosting platforms, e.g., GitHub [1] and TortoiseSVN [2], have been widely adopted in the life cycle of software development. These platforms greatly reduce time cost and maintenance cost. However, during the software updating, developers are required to submit commit messages to document code changes. The commit messages, which summarize what happened or explain why the changes were made, are usually described in natural language; thus the messages can help developers capture a high-level intuition without auditing implementation details. Hence, high-quality commit messages are essential for developers to comprehend version evolution rapidly.

However, manually writing commit messages is time-consuming and labour-intensive. First, until now, there is no specification regarding the writing format of commit messages when developers submit commits, and developers tend to follow their own writing styles. Second, developers tend to commit without writing the corresponding messages to make readers difficult to extract the precise description behind code changes manually. For example, according to the report [3] in SourceForge [4], an Open Source community dedicated to creating, collaborating and distributing projects, there are around 14 percent of commit messages in more than 23,000 open-source Java projects that are empty. Among our collected dataset which contains the top-ranked ∼60 projects in terms of star numbers on GitHub, e.g., Junit5 [5] and Neo4j [6], we find that meaningless commit messages[1] also account for around 10 percent of the entire collected commits. Therefore, automated generation of commit messages for code changes is necessitated and meaningful for software developers.

Generating accurate commit messages by given code changes is a challenging task. Several approaches have been exhibited for generating commit message automatically. The rule-based methods, e.g., DeltaDoc [9] and ChangeScribe [10], are able to summarize code changes based on specific customized rules. However, these proposed rules could not easily cover all the cases and the generated messages are verbose, failing to capture the semantics behind a change [7]. To deal with this limitation, Jiang *et al.* [8] proposed a generation-based approach, which adopts a neural machine translation (NMT) model for translating code changes into commit messages. However, the NMT model treats code as a flat sequence of tokens, which ignores the syntactic and semantic information behind programs, thus fail to learn the semantics behind

- *Shangqing Liu and Yang Liu are with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: shangqin001@e.ntu.edu.sg, yangliu@ntu.edu.sg.*
- *Cuiyun Gao is with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518055, China. E-mail: gaocuiyun@hit.edu.cn.*
- *Sen Chen is with the College of Intelligence and Computing, Tianjin University, Tianjin 300350, China , and also with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: ecnuchensen@gmail.com.*
- *Lun Yiu Nie is with the Chinese University of Hong Kong, Hong Kong, China. E-mail: lynie8@cse.cuhk.edu.*

Manuscript received 29 July 2020; revised 11 Oct. 2020; accepted 11 Nov. 2020. Date of publication 17 Nov. 2020; date of current version 16 May 2022. (Corresponding author: Cuiyun Gao.)
Recommended for acceptance by D. Hao.
Digital Object Identifier no. 10.1109/TSE.2020.3038681

1. Meaningless refers to empty, non-ASCII, merge and rollback commits.

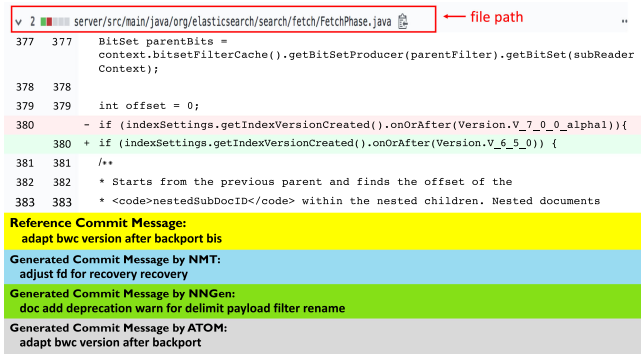


Fig. 1. Example of the retrieved message by NNGen [7], generated messages by NMT [8], and the proposed ATOM for one code change of the commit *41528c0813fe72162408051e3af29ac42b4708f7*.

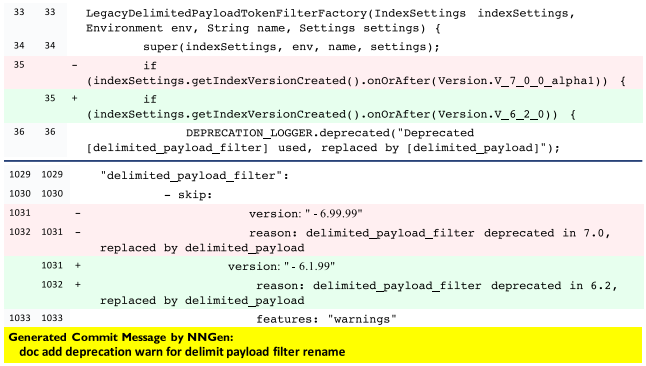


Fig. 2. The code change of retrieved commit by NNGen [7] with its id *c4fe7d3f7248223d5174b36fd4e1678217a6a6ed*.

the code changes. Some other researchers [7], [11] attempt to reuse the existing commit messages in the collected dataset by Information Retrieval to achieve the best performance. However, the retrieval-based approaches may achieve promising performance on similar programs, but are limited by the poorer performance on the programs that are very different from the retrieved database. For example, in Fig. 1, the message produced by retrieval-based approach, i.e., NNGen is unrelated to the code changes. Furthermore, Fig. 2 shows the retrieved commit of the commit in Fig. 1, where contains two parts, separating by a black line. We can see that the first part is similar to the code changes in Fig. 1, but the retrieved messages (e.g., *delimit*, *payload*, and *filter*) are from the second part of code changes. Hence, the retrieval-based approach has no capacity to produce the exact commit messages on the dissimilar programs. Considering retrieval-based and generation-based techniques both have their merits, one intuition is to combine both for generating high-quality commit messages.

To this end, we propose a novel commit message generation model, named ATOM (**A**bstract syntax **T**ree-based c**O**mmit **M**essage generation) for better commit message generation. ATOM contains three modules, 1) a generation module, which encodes the structure of changed code, i.e., Abstract Syntax Tree (AST), to enrich the semantic representation; 2) a retrieval module, which retrieves the most similar commit message based on the text-similarity; 3) a hybrid ranking module, which learns to prioritize the commit messages generated by generation and retrieval modules to further enhance the semantic relevance to the corresponding code changes. To evaluate our proposed ATOM, we crawl and build a new benchmark since AST cannot be constructed in the previous benchmarks [8]. We quantitatively evaluate ATOM on our crawled benchmark, including ∼160k commits in total. Extensive experimental results demonstrate that ATOM can significantly outperform the state-of-the-art approaches by increasing at least 30.72 percent in terms of BLEU-4 score [12] (an accuracy measure that is widely used to evaluate text generation systems). Furthermore, ATOM can enhance the performance of its generation module by 42.99 percent by our well-designed hybrid ranking module.

The main contributions can be summarized as follows:

- We propose a novel generation module based on AST from code changes, named *AST2seq*, to better capture code semantics and encode code changes.
- We design a hybrid ranking module to enhance the output of generation modules, by providing the most accurate commit messages among the generated and retrieved results.
- We provide a new and well-cleaned benchmark, including complete function-level code snippets of ∼160k commits from 56 java projects. We clean the benchmark by filtering out meaningless (e.g., empty, non-ASCII, merge) commits and make the code [13] and benchmark [14] public to benefit community research.
- Extensive quantitative and qualitative experiments including a human evaluation demonstrate the effectiveness and usefulness of our proposed model.

The remainder of this paper is organized as follows. Section 2 presents some basic knowledge about commits and neural networks. Then we describe the details about ATOM in Section 3. Experimental results, human evaluation, and examples are conducted in Section 4. Section 5 gives some discussion about ATOM, followed by the related work in Section 6. Finally, Section 7 concludes this paper.

2 Motivation and Background

In this section, we first introduce several features relevant to the commit, the motivation of our model design, and some deep learning models/mechanisms used in our paper.

2.1 Commit, `diff`, and Commit Messages

Commits are used in Git [1] to record the changes between different versions. As shown in Fig. 1, a commit usually contains a commit message and a change. The commit message is written by developers in a textual format to facilitate the understanding of current changes and the code change is called `diff` to characterize the difference between two code versions. Usually, a `diff` may contain one or multiple chunks with file paths, which can be found at a red rectangle in the upper part in Fig. 1, along with the identifier "diff –git" to indicate the changed file name. The modified codes are wrapped by "@@" in a chunk with the negative sign '-' or positive sign '+' with a line number to denote the deleted or added line of code. Hence, we can summarize the commit in Fig. 1, in "FetchPhase.java file", there is one line of change at line number 380. We refer to the pair of `diff` and its corresponding message as a commit in this work.

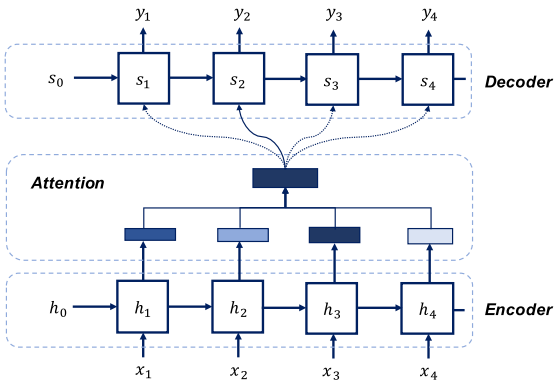


Fig. 3. An encoder-decoder model with an attention mechanism.

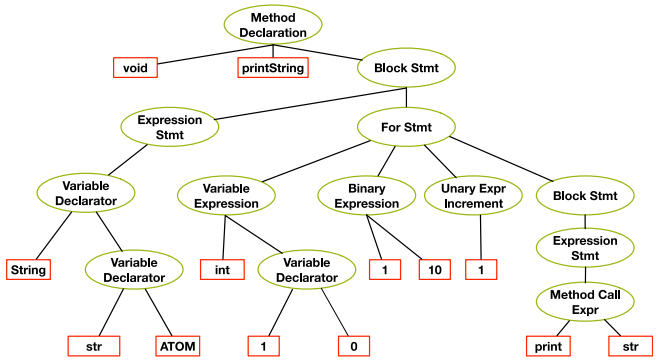


Fig. 4. The AST compiled from Listing 1.

2.2 Motivation

Existing studies on commit message generation [8], [15], [16] generally treat the code changes as a sequence of code tokens and ignore the hierarchical code structure information. The work in other program comprehension tasks such as program vulnerability identification [17], function name prediction [18], and source code summarization [19], [20], have utilized code structure such as Abstract Syntax Tree (AST) to learn code semantics and good performance has been demonstrated. Thus, in this paper, we aim at exploiting AST for better-representing code changes. Since code structures of code changes cannot be directly obtained by parsing functions, the usage of ASTs for the commit message generation task is more challenging. To well capture the semantics of long AST paths, we determine to use bi-directional LSTM [21] which shows effectiveness on representation learning of long-term sequences. However, a potential issue with bi-directional LSTM model is that the model needs to compress all the necessary information of the paths into a fixed-length vector [22]. To alleviate the issue, we follow the previous studies [20], [23] to use the attention mechanism since attention can focus on some important paths to represent code changes.

2.3 Abstract Syntax Tree (AST)

An abstract syntax tree is a high abstraction of source code, which is a tree structure and serves as the intermediate representation of program language. An AST usually contains leaf nodes that represent identifier and literal in the code and non-leaf nodes which can represent some syntactic structure within codes. Specifically, Fig. 4 shows a simple AST with the code snippet in Listing 1, where identifier name e.g., "str", "ATOM" or type e.g., "int", "String" are represented by the values of leaf nodes and non-leaf nodes e.g., "ExpressionStmt", "ForStmt" tend to have more syntactic information. We can get a total of 106 different non-leaf nodes with *JavaParser* [24], which is a tool used for extracting ASTs in Java language. Adopting AST in code comprehension has been proved to get the state-of-the-art performance, such as code2seq [20], code2vec [23], DeepCom [25], CRF [26], Devign [17].

2.4 Encoder-Decoder Model

The basic structure of NMT [27] used to translate source sequences into targets is encoder-decoder, as shown in Fig. 3. The feature vectors generated by encoder are fed into the decoder to generate target sequences. Usually, it consists of two RNNs [28] with built-in LSTM cells [29] and attention mechanism [30], [31] for translation.

Listing 1. A Simple Java Code Snippet

```
public void printString(){
  String str = "ATOM"
  for(int i = 0; i < 10; i++){
    print(str);
  }
}
```

2.4.1 Recurrent Neural Network (RNN)

RNNs are widely used to capture information from time-series data as their chain-like natures. The loop contained in RNNs allows information to be passed from one-time step to the next. At each time t, the unit in RNNs takes x_t and the hidden state h_{t-1}, which is produced by previous time $t-1$ as input to predict the current output y_t. The chain-like structure enables RNNs to learn information from the past, however, they also suffer from long-term dependencies. Since RNNs are unable to connect information from further back and cannot handle long sequences, some variants e.g., Long Short-term Memory (LSTM) [29] and Gated Recurrent Unit (GRU) [32] are proposed.

2.4.2 Long Short-Term Memory (LSTM)

LSTMs introduced by Hochreiter and Schmidhube [29] are explicitly designed with a memory cell to remember important information. The gating mechanism in the memory cell helps LSTMs selectively 'forget' unimportant information, thus allowing more space to take in information and controls when and how to read previous information and write new information. In this way, the memory cell will preserve more long-term dependencies than vanilla RNNs. Hence, RNNs built with LSTMs are widely used for sequence models to capture information.

2.4.3 Attention Mechanism

Attention is proposed to boost the performance of Encoder-Decoder further, as it utilizes all the hidden states of the input sequence rather than the final hidden state as a context vector for the decoder. It creates an attention mapping matrix between each time step of the decoder output to the

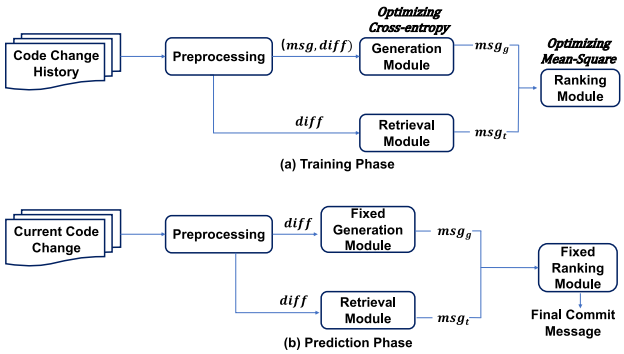


Fig. 5. Architecture of ATOM.

encoder hidden states. The attention weights are trained by a forward neural network to align the scores between the encoder states and the decoder outputs. This means, for each output, it has access to the entire input sequence and dynamically selects specific elements from the input. Hence, the attention mechanism allows the decoder to focus and place more *Attention* on the relevant parts of the input. The Bahdanau [30] or Luong [31] Attention has been widely adopted into neural machine translation [33], reading comprehension [34] and computer vision [35].

2.5 Convolutional Neural Network (CNN)

Consisting of Convolutional Layer, Pooling Layer and Fully-Connected Layer, Convolutional Neural Networks (CNNs) are one of the most common Deep Neural Network architectures. The convolution operations apply kernels to extract features from the feature maps, which allow the network to capture high-level abstract information with a reduced number of parameters. In image processing tasks, for instance, the convolution layers can learn edges, patterns and shapes after training. Similarly, CNN can also be applied to natural language processing tasks. In previous work [36], [37], CNN-based neural ranking models are trained to learn high-level sentence matching patterns.

3 Our Approach (ATOM)

In this section, we provide the overview of our approach ATOM and detail each of the modules.

3.1 Overview

Fig. 5 shows the architecture of our framework, which consists the following components.

- *Preprocessing Module.* The commit message and code changes are processed separately. We extract AST paths corresponding to the code changes by retrieving the completed functions in the repository. We also use the first sentence with lemmatization from the commit message as the target sentence to represent the entire commit message.
- *Generation Module.* We name our generation module as *AST2seq* and encode AST paths from `diffs` with BiLSTM to represent code changes and followed a decoder with an attention mechanism to generate a new message msg_g.
- *Retrieval Module.* The retrieval module uses a "`diff-diff`" match to retrieve the most relevant commit messages. This approach matches `diff` with all `diffs` in the training set and get the most relevant message msg_t based on the cosine similarity.
- *Ranking Module.* To incorporate the retrieval results into the generation module, we train a CNN to adaptively rank the generated message msg_g and the retrieved message msg_t.

At the prediction phase, when a new code change arrives, ATOM generates the corresponding commit message with the trained generation module and ranking module, as shown in Fig. 5b.

3.2 Preprocessing Module

We preprocess code changes and commit messages separately for preparing the input of ATOM.

3.2.1 Code Changes

We first divide code changes `diffs` into *added* and *deleted* groups based on the corresponding sign, i.e., "+" and "-". Then we tokenize the `diffs` with pygments [38], and remove meaningless tokens such as punctuations. Consequently, we obtain a list of tokens for the *added* code and *deleted* code, denoted as W^+ and W^- respectively, where $W^{+/-} = \{w_1, w_2, \ldots, w_i\}$ and i is the ith token in the changed code.

The basic compilation unit [39], containing a single class definition and wrapped functions, is needed to extract AST paths based on the `diffs`. Hence, we retrieve completed functions of `diffs` denoted as *added* function and *deleted* function. We use Ctags [40] with file paths and modified line numbers containing in `diffs` to retrieve completed functions in the repository and then parse them to obtain ASTs with JavaParser [24]. As all tokens belonging to $W^{+/-}$ are the values of leave nodes in an AST, we search the shortest distance[2] for any two tokens, w_i and w_j in $W^{+/-}$ and denote the path as $x = \{w_i, n_1, \ldots, n_l, w_j\}$, where n_l means the lth non-leaf node. Following this procedure, we finally obtain AST paths for the whole *added/deleted* code, indicated as a set of $X^{+/-} = \{x_1^{+/-}, \ldots, x_{p/k}^{+/-}\}$ where p, k are the total number of AST paths respectively.

3.2.2 Commit Messages

We extract the first sentence from the commit message as the target sequence since the first sentence is often the summary of the entire commit [8], [41], [42]. We split the tokens with underlines "_" and replace file names and digits with unique placeholders "<FILE>" and "<NUMBER>" respectively. We also lemmatize each word into its base form by using the NLTK toolkit [43] to reduce the vocabulary set. The lemmatized message is denoted as $M = \{y_1, y_2, \ldots, y_n\}$ where n is the token length of the message.

3.3 Generation Module

Prior work on commit message generation treated `diffs` as a flat sequence of tokens, which is limited by long sequences

2. Here the shortest distance refers to the minimum edges between two corresponding leave nodes.

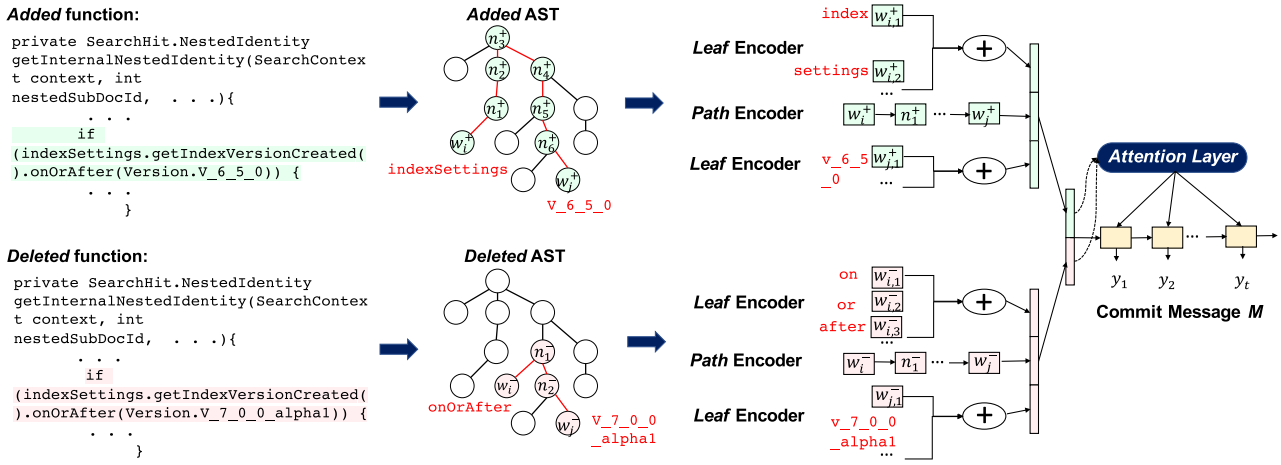


Fig. 6. Architecture of *AST2seq* with the example in Fig. 1, where *added* and *deleted* function denote the completed functions retrieved from the `diff`. The highlight path in *added* or *deleted* AST is one of paths extracted from tokens, e.g., indexSettings, onOrAfter in `diff` and $n_l^{+/-}$ is the non-leaf node, e.g., "ForStmt", "Binary Expression".

and ignores the code structure, e.g., Abstract Syntax Trees (ASTs) of the `diffs`, to capture the semantics. AST is an abstraction of code and has been proved useful in representing code semantics [20], [23], [25]. However, the AST-based approaches mostly extract ASTs on the completed functions to understand the functionality of codes. In our generation module, we extract the AST paths based on the `diffs` for representing code changes. Compared to the sequence-based approaches, our method can generate messages with longer `diffs` and we name as *AST2seq*. The entire architecture of *AST2seq* is illustrated in Fig. 6, involving three main components sequentially: *AST Encoder* for encoding each AST path into its vector representation; *Attention* for dynamically focusing on the relevant AST paths; and *Message Decoder* for generating corresponding commit message of the code change.

3.3.1 AST Encoder

Given a set of *added* and *deleted* AST paths $X = \{x_1^{+/-}, x_2^{+/-}, \ldots, x_{p/k}^{+/-}\}$, where $x \in X$ can be represented as $\{w_i, n_1, \ldots, n_l, w_j\}$ and p, k are the *added* and *deleted* AST paths. We encode each path x with a bi-directional LSTM to create a vector representation z. Here we use bi-directional LSTM is to expect the bi-directional LSTM can capture the long-term semantics in each AST path.

- *Path Representation.* The types of nodes e.g., "ForStmt", "IfStmt" that make up an AST path x is limited to 106 and we represent these node types with an embedding matrix E^{nodes} and then encode the path e.g., $\{w_i^-, n_1^-, n_2^-, w_j^-\}$ in Fig. 6 into a bi-directional LSTM to obtain the dense representation $h_{w_i}, \ldots, h_{w_j}$ and use the final states of LSTM as path representation

$$h_{w_i^+}, \ldots, h_{w_j^+} = LSTM(E^{nodes}_{w_i^+}, \ldots, E^{nodes}_{w_j^+}) \quad (1)$$

$$path\_feat^+ = [h^{\leftarrow}_{w_i^+}; h^{\rightarrow}_{w_j^+}] \quad (2)$$

$$h_{w_i^-}, \ldots, h_{w_j^-} = LSTM(E^{nodes}_{w_i^-}, \ldots, E^{nodes}_{w_j^-}) \quad (3)$$

$$path\_feat^- = [h^{\leftarrow}_{w_i^-}; h^{\rightarrow}_{w_j^-}]. \quad (4)$$

- *Leaf Representation.* As the values of start leaf node w_i and end leaf node w_j of an AST path also appear in the `diff`, we incorporate them for representing a completed path. We split the tokens of the values in leaf nodes e.g., "onOrAfter" in Fig. 6 into subtokens, "on", "or", "after" and then combine the embeddings of these subtokens with summation to represent a leaf token

$$leaf\_feat_{w^+} = \sum_{s \in split(w^+)} E^{subtokens}_{w^+}[s] \quad (5)$$

$$leaf\_feat_{w^-} = \sum_{s \in split(w^-)} E^{subtokens}_{w^-}[s]. \quad (6)$$

To represent a completed path $x^{+/-}$, we aggregate the path representation and leaf representation by employing a fully connected layer

$$z^{+/-} = layer([leaf\_feat_{w_i^{+/-}}; path\_feat^{+/-}; leaf\_feat_{w_j^{+/-}}]). \quad (7)$$

Finally, we concatenate p *added* and k *deleted* paths of vector z for representing a `diff`

$$Z = [z_p^+; z_k^-]. \quad (8)$$

3.3.2 Attention

Given a set of *added* and *deleted* AST path representations $Z = \{z_1, z_2, \ldots, z_{p+k}\}$, where $p + k$ is the summation of *added* and *deleted* paths, we need to focus on some important paths which can capture the information to represent the entire code changes. Hence, attention is needed to learns how much focus "attention" should be given to each AST path.

We use Luong Attention Mechanism [31], which is shown in the Equation (10). During decoding period, the attention will learn the weight distribution over these paths to capture the important paths.

3.3.3 Message Decoder

We take the average of vector representations of *added* and *deleted* paths, i.e., $Z = \{z_1, z_2, \ldots, z_{p+k}\}$, as an initial hidden state of the decoder, that is

$$h_0 = \frac{1}{p+k}\sum_{i=1}^{p+k} z_i. \tag{9}$$

At each decoding step t, a context vector c_t is computed based on Z and current hidden state h_t in the decoder

$$\alpha^t = softmax(h_t W_\alpha Z), \qquad c_t = \sum_{i}^{p+k} \alpha_i^t z_i. \tag{10}$$

α^t is the variable-length alignment vector whose size equals the number of *added* and *deleted* paths. Then c_t and h_t are combined to predict the current token y_t [31]

$$p(y_t|y < t, z_1, \ldots, z_{p+k}) = softmax(W_s tanh(W_c[c_t; h_t])). \tag{11}$$

The loss function we adopted in *AST2seq* is softmax cross entropy with logits

$$Loss = y\log(\frac{e^{logits}}{\sum_r e^{logits}}), \tag{12}$$

where r is the commit message vocabulary size, y is the true token of message and $logits$ is the output of decoder module.

3.4 Retrieval Module

The retrieval module aims at retrieving relevant commit messages from the training set. We adopt a "`diff-diff`" match for the retrieval. Specifically, we first index all `diffs` in training set with sklearn [44]. Then for each `diff` in the validation and test sets, we compute the cosine similarity in the training set based on their tokens of tf-idf scores [45], and keep the most relevant one commit message (first-ranked) from the training set. Term frequency (TF) and inverse document frequency (IDF) can be computed by the following equation:

$$tf_{i,d} = \frac{n_{i,d}}{\sum_{i \in W} n_{i,d}} \quad idf(i) = log\left(\frac{N_{diff}}{df_i}\right), \tag{13}$$

where $n_{i,d}$ is the number of i_{th} token in the d and W is the set of distinctive tokens. In the second equation, df_i is the number of `diffs` that contains i_{th} token in the entire `diffs` and N_{diff} is the total number of `diffs`.

The retrieved commit message serves as one candidate for the final generated message, and will be fed into the ranking module together with the message produced by the generation module.

3.5 Hybrid Ranking Module

From the retrieval module (described in Section 3.5) and generation module (described in Section 3.4), we can get two commit message candidates $y \in \{msg_t, msg_g\}$ where msg_t is the first-ranked retrieved commit message and msg_g is the generated message. To predict which candidate is better, we can train a binary classifier based on popular models such as XGBoost [46] and LSTM [29]. However, since `diffs` may contain tokens that tend to appear in the generated messages, e.g., tokens related to function name and variable name, the relevance between `diffs` and candidate messages would be useful for the final message prediction. Inspired by Liu *et al.* [47], we design a similarity matching matrix to measure the relevance between `diff` and the corresponding candidate message, and adopt ConvNet model to learn their relevance score. Experiments in Section 4.3.4 show that ConvNet outperforms typical classifiers (e.g., XGBoost [46] and LSTM [29]).

3.5.1 Similarity Matching Matrix

For any `diff` d, we first looks up embeddings for tokens in d and y respectively, denoted as $E(d) = [d_1, d_2, \ldots, d_{L_d}]$ and $E(y) = [y_1, y_2, \ldots, y_{L_y}]$ where L_d and L_y are the lengths of d and y respectively. Note that the embedding matrixes for `diffs` and messages are trained separately, but their dimension sizes are equal. The interaction matching matrix D is computed by the following equation:

$$D = E(d) \times E^\top(y), \tag{14}$$

where D has the dimension with (L_d, L_y) and is used as the input of ConvNet to predict a matching score.

3.5.2 ConvNet Model

ConvNet is designed to find the correlation between `diff` and message and give a better output among commit message candidates which contains convolution and max-pooling operations on the similarity matching matrix D. Let C_{in} denote the number of input channels, H is the height of input plane and W is width, which in our initial settings equal to 1, L_y and L_d respectively. The convolution operation $out(C_{out})$ on the input $input$ with size (C_{in}, H, W) and output size $(C_{out}, H_{out}, W_{out})$ can be expressed as

$$out(C_{out}) = \sigma(bias(C_{out}) + \sum_{k=0}^{C_{in}-1} weight(C_{out}, k) * input(k), \tag{15}$$

where σ is the activation function, and $*$ is the valid dot product operator. Max-pooling operation with input size (C, H, W) is conducted after the convolution operation, which can be expressed as

$$\begin{aligned} out(C, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \\ input(C, stride[0] \times h + m, stride[1] \times w + n), \end{aligned} \tag{16}$$

where (kH, kW) is the kernel size and $stride[\cdot]$ is the tuple of the sliding blocks over the input, $stride[0]$ and $stride[1]$ represent the block height and width correspondingly. Finally we feed the output produced by ConvNet into a fully connected layer to compute the relevance score between `diff`

d and message y. We use Mean Square Error Loss function [48] to optimize the loss values in the form of

$$Loss = |Y' - Y|^2, \tag{17}$$

where Y' is the output of ConvNet and Y is true relevance score.

3.5.3 Training for ConvNet

One challenging part in the hybrid ranking module is how to well define the true relevance scores Y between `diffs` and corresponding candidate messages. One possible solution is to manually evaluate these candidates, however, the time and labour cost would be very intensive and it is not applicable for end-to-end training. To enable an end-to-end training process, we propose to build upon the evaluation metrics, e.g., BLEU-4 [47], [49]. Specifically, we score these two candidate messages by comparing them with the ground truth using BLEU-4, and the scores will serve as our optimization target Y for the model train. The trained ConvNet can predict the commit message from $\{msg_t, msg_g\}$, where a higher score means higher relevance of the message to the `diff`.

4 Experimental Evaluation

In this section, we conduct experiments to evaluate the effectiveness of ATOM and compare it with some state-of-the-art approaches.

4.1 Setup

4.1.1 Experimental Benchmark

The dataset utilized in previous works [7], [8], [50] contains no commit ids or complete functions and we can not use directly as ASTs are not available. We crawled 56 popular projects including Neo4j [6], Structs [51], Antlr4 [52] from GitHub based on the "project stars". The raw messages from this dataset are quite noisy since some commits are empty or contain non-ASCII messages. Furthermore, the merge or rollback commits may contain too many lines, which is not suitable for the generation module. So we filter them out to eliminate unrelated information and remain with 628,887 commits. Additionally, some commits related to project initialization and fundamental functionality updating contain many changes, we remove them as well. Specifically, we set the thresholds of chunks as 5 and leave with 438,665 commits. As we need to extract the modified ASTs from java functions so we keep commits with *.java* files and remain 197,968 commits. After removing message length greater than 20 and the same contents of the commits, we keep ~160k samples finally and similar to Jiang *et al.* [8]'s work, randomly select 10 percent for testing, 10 percent for validation and the remaining for training. For more details about our benchmark, please refer to Section 5.2.

4.1.2 Experimental Settings

For *AST2seq* in the generation module, the max number of paths in *added* and *deleted* ASTs are set to 80 (with more details illustrated in Section 4.3.3). The embedding sizes for subtokens, paths and target sources are defined as 128. The bidirectional LSTM is utilized for encoder layer and LSTM is used for the decoder. All dimensions of the hidden states in the encoder and decoder are fixed to 256. The probability of dropout [53] is set as 0.4 to avoid overfitting. We set the number of epochs equal to 3,000, along with the batch size as 256 and patience , a threshold to terminate training for early stopping, as 20. The learning rate is equal to 0.0001. During testing, we use beam search with beam width as 5 since it has proven useful in sequence prediction with recurrent neural network [54]. For ConvNet training, we adopt a 2-D convolutional layer with the number of kernels defined as 16 and kernel size as $(3, 3)$, followed by a ReLU function and a max-pooling layer with $stride$ size equal to $(2, 2)$. After the max-pooling operation followed by fully connected layers to convert the vector into score values. The optimizer we choose for *AST2seq* and ConvNet is Adam [55]. We use Tensorflow 1.12 [56] and Pytorch 1.4 [57] for our model training. Hyperparameters such as learning rate, embedding size, encoder and decoder layer numbers, and kernel sizes are tuned with grid search on the validation set [58]. The remaining hyperparameters (e.g., beam width and batch size) are configured the same as those in Code2seq [20]. The experiments have been conducted on servers with 36 cores and 4 Nvidia Graphics Tesla P40 and M40.

4.1.3 Evaluation Metrics

We evaluate our proposed ATOM with widely-used automatic metrics such as BLEU-N [12], ROUGE-L [59], and Meteor [60]. These metrics have been proved in measuring text similarities between the produced messages and ground truths.

BLEU-N computes the n-gram precision of a candidate sequence to the reference, with a penalty for overly short sentences. BLEU-1/2/3/4 correspond to the scores of unigram, 2-grams, 3-grams and 4-grams, respectively

$$BLEU - N = BP * exp\left(\sum_{n=1}^{N} w_n log p_n\right), \tag{18}$$

where $N = 1, 2, 3, 4$, uniform weights $w_n = 1/N$, and

$$BP = \begin{cases} 1, & \text{if } c > r. \\ e^{1-r/c}, & \text{if } c \leq r. \end{cases} \tag{19}$$

where c is the length of the candidate sequence and r is the length of the reference sequence.

ROUGE-L provides F-score based on Longest Common Subsequence (LCS). It compares the similarity between two given texts in automatic summarization evaluation.

Meteor modifies the precision and recall computation, replacing them with a weighted F-score based on mapping unigrams and a penalty function for incorrect word order

$$Meteor = F_{mean}(1 - Penalty), \tag{20}$$

where F_{mean} is computed with unigram precision (P) and unigram recall (R)

TABLE 1
Comparison Results With Baseline Models and Different Modules Within ATOM

Methods		BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
Baselines	$\text{NMT}_{(\text{Luong})}$	13.12	8.01	6.11	5.23	12.73	10.37
	$\text{NMT}_{(\text{Bahdanau})}$	12.78	7.66	5.72	4.81	11.95	9.87
	NNGen	16.91	12.01	10.03	8.04	15.20	13.68
	Ptr-Net	5.80	1.72	0.73	0.45	7.61	4.98
	CODISUM	7.82	3.61	2.22	1.75	9.87	8.35
	Commit2Vec	12.72	7.78	6.09	5.38	13.54	10.43
Ours	ATOM_{Gen}	15.97	10.70	8.83	7.35	14.80	11.82
	ATOM_{Ret}	17.74	12.65	10.55	8.52	15.93	14.35
	ATOM	**23.88**	**15.61**	**12.17**	**10.51**	**22.02**	**18.51**

$$F_{mean} = \frac{10PR}{R + 9P}, \tag{21}$$

and $Penalty$ is levied for fragmented matches as the ratio of matched chunk number to matched unigram number

$$Penalty = 0.5 * (\frac{\#chunks}{\#unigrams\_matched})^3. \tag{22}$$

4.2 Comparison Methods

We evaluate the proposed ATOM against baseline models including the state-of-the-art approaches. We divide them into two groups: 1) Retrieval-based approach: NNGen [7]; and 2) Generation-based approach: NMT [8], [15], [16], Ptr-Net [61], CODISUM [50] and Commit2Vec [62]. For the implementation, we reproduce NNGen [7] by using the same algorithm and settings according to the original paper. The source code of NMT , Ptr-Net and CODISUM is available online [63], [64], [65] and we utilize the default settings described in the corresponding papers. For Commit2Vec, we try our best to replicate the code for commit message generation according to the paper and make the replication publicly available [13] The details of these approaches are illustrated below.

- NNGen [7]. NNGen is a retrieval-based approach which retrieves the most similar top-k `diffs` from the training dataset based on a bag-of-words [66] model and prioritizes the `diff` candidates in terms of BLEU-4 scores. NNGen regards the message of the `diff` with the highest BLEU-4 score as the result.
- NMT [8], [15], [67]. NMT adopts attention-based RNN encoder-decoder models, described in Section 2.4 to generate commit messages for `diffs`. Jiang *et al*. [8] uses Bahdanau attention [30] to produce messages. Another approach Commitgen proposed by Loyola *et al*. [15] leverages Luong [31] attention instead of Bahdanau for commit message generation. We compare both attention mechanisms denoted as $\text{NMT}_{(\text{Luong})}$ and $\text{NMT}_{(\text{Bahdanau})}$.
- Ptr-Net [61], [68]. Ptr-Net (an abbreviation of Pointer network) is a typical text summarization approach, which can copy the Out-of-Vocabulary (OOV) words such as variable and method names from source code to the generated messages. Ptr-Net has proven effective in generating rational commit messages for code changes by Liu *et al*. [61].
- CODISUM [50]. CODISUM is the state-of-the-art approach which employs the normalized code changes in which the identifiers are unified with corresponding placeholders for learning the representations of code changes as well as combining pointer network [68] to mitigate the OOV issue.
- Commit2Vec [62]. Commit2Vec feeds the *added* and *deleted* AST paths to a fully-connected layer to encode code changes for classifying security-related commits. Although Commit2Vec is targeted at binary classification, the encoding mode of code changes can be adopted for various downstream tasks including commit message generation, so we also consider Commit2Vec as one baseline. Since the source code is not publicly available, we tried our best to reproduce the model according to the paper."

4.3 Experimental Results

We present the experimental results and analysis through the following research questions.

4.3.1 *What is the Performance of ATOM Comparing With Baseline Approaches?*

Table 1 shows the results of our approach against the baselines. We can find that ATOM outperforms the baselines by a significant margin. The improved models such as Ptr-Net and CODISUM, which claimed to capture code semantics, have lower performances. In essence, they treat code as a flat sequence of tokens, failing to capture the semantics behind the code. Commit2Vec, which also adopts AST paths to represent code changes, presents lower performance than ATOM in terms of all the evaluation metrics. The lower performance may be attributed to that Commit2Vec utilizes a fully-connected layer to represent code changes and could fail to capture the sequential information in the *added/deleted* AST paths. In our proposed *AST2seq*, bi-directional LSTMs are involved to incorporate the sequential information of the *added/deleted* AST paths for better representing code changes. The retrieval-based approach NNGen has a higher performance than pure generation approaches, demonstrating the effectiveness of the retrieval-based method on message generation tasks. Finally, ATOM improves all the baseline approaches by 30.72, 44.89, and 35.26 percent in

TABLE 2
Percentage of Final Results Prioritized From Retrieved and Generated Messages

Modules	Number	Percentage (%)
Retrieval	8,168	55.66%
Generation	6,506	44.34%

terms of BLEU-4, ROUGE-L, and Meteor respectively. This can be attributed to that ATOM can effectively integrate the advantages of generation and retrieval modules.

Answer to RQ1. *In summary, ATOM improves the baseline approaches by 30.72, 44.89, 35.26 percent in terms of BLEU-4, ROUGE-L, and Meteor respectively.*

4.3.2 What is the Impact of Individual Modules on the Performance of ATOM?

We also perform experiments to evaluate the impact of individual generation module and retrieval module on the generated commit messages, with the results shown in Table 1. No ranking model is included in the two variants of ATOM. We denote the results produced only by the retrieval module as ATOM_{Ret}, which uses TF-IDF to retrieve the most similar commit message (see Section 3.4), and generation module as ATOM_{Gen}, which only uses *AST2seq* for commit message generation (see Section 3.3). We find that the performance of ATOM_{Ret} is slightly higher than ATOM_{Gen}, but the overall performance is still lower than the combined model ATOM. The gains achieved by our hybrid ranking module range from 12.76 to 56.58 percent in terms of BLEU-4, ROUGE-L and Meteor. Hence, ATOM incorporates the retrieval results into the generated results by the hybrid ranking module will further boost the performance. In addition, ATOM_{Gen} achieves the best performance among all the generation approaches, i.e., NMT, Ptr-Net, and CODISUM, which proves that utilizing AST to learn code semantics is more powerful than simple sequential models. Finally, compared with the retrieval-based approach NNGen, ATOM_{Ret} has slightly better performance, since we retrieve the most similar commit message based on the weight of tokens. Hence, some important tokens with low frequency will be considered, which is superior to NNGen.

As ATOM outputs the commit messages produced by either generation or retrieval module, we also analyze the proportions of the messages from each module, with statistics shown in Table 2. In a total of 14,674 testing samples, 8,168 of the results are from the retrieval module, accounting for 55.66 percent of the entire testing corpus and the remaining 6,506 are from the generation module (44.34 percent). Based on the statistics, we can conclude that both retrieval and generation modules are helpful for accurate commit message generation, and they are complementary to each other.

Answer to RQ2. *In summary, ATOM incorporates the retrieval results into generation module to boost the final performance, and the improvement range from 12.76 to 56.58 percent in BLEU-4. Furthermore, among all the generation approaches, our proposed* AST2seq *can learn more semantics in the commits to produce high-quality messages.*

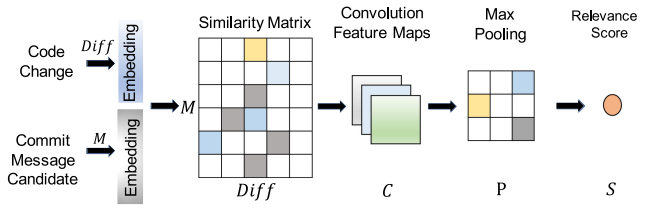


Fig. 7. Architecture of ConvNet.

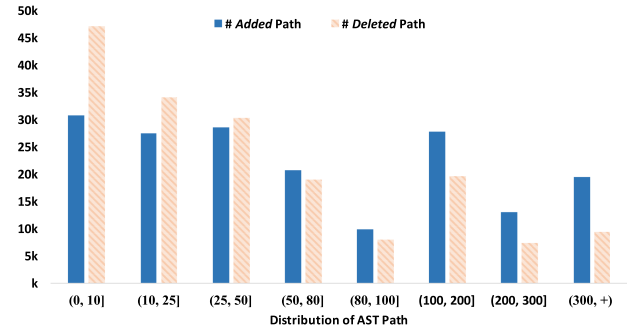


Fig. 8. The distribution of AST paths shown in the dataset. Each bar represents the number of commits that has the number of AST paths in a specific interval. For example, the leftmost blue bar represents almost 30,000 commits in our dataset have less than 10 *added* AST paths by our preprocess.

4.3.3 How Accurate is AST2seq Under a Different Number of Paths?

Our generation module *AST2seq* encodes AST paths based on `diffs` to represent code changes, however, the number of paths vary depending on the length of `diffs`. In this paper, we set the max number of paths to 80 for the *added* and *deleted* ASTs during training respectively. From Fig. 8, we can see that nearly 80 percent of commits have fewer than 80 AST paths in our dataset. In this RQ, we analyze the impact of different numbers of AST paths on the model performance. Specifically, we truncate the ASTs with longer paths to be the experimental number of ASTs. For example, to examine the results when taking the number of AST paths as 30, we randomly select 30 paths for the ASTs with real paths larger than 30. The results are illustrated in Table 3. As can be seen, the optimal value of the path number in our experiment is 80 and BLEU-4, ROUGE(1,2,L) and Meteor achieves 7.35, 16.69, 6.23, 14.80 and 11.82 respectively. Furthermore, few path numbers tend to show worse results, e.g., when the path is set as 30, the performance decreases dramatically to 4.27. It can be attributed to fewer paths have limited capability in representing code changes. Increasing paths to over 100 do not result in continuously improved performance and the scores show a slight decrease when the paths augmented from 200 to 300. In addition, large numbers of paths will be a heavy burden for model training. Hence, we can conclude that 80 is an optimal value to represent `diffs`.

Answer to RQ3. *Overall, the optimal value of AST paths for effectively representing* `diffs` *is 80. Adding fewer or more paths cannot contribute much to the performance.*

4.3.4 What is the Impact of Different Ranking Methods?

ATOM designs a ConvNet to incorporate the output of retrieval module into generation module *AST2seq* to get better results. However, the hybrid ranking module can be regarded as a regression problem, and be solved with other

TABLE 3
The Performance of Path Set on *AST2seq*

# Path	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
30	11.53	6.57	4.80	4.27	13.66	10.22
50	13.67	8.70	6.83	5.96	13.92	10.89
80	**15.97**	**10.70**	**8.83**	**7.35**	**14.80**	**11.82**
100	15.11	10.01	8.19	7.09	14.34	11.42
200	13.89	8.83	6.88	6.07	14.01	11.01
300	13.72	8.77	6.85	6.04	13.95	10.99

The left column represents the path number for added *and* deleted *paths separately.*

TABLE 4
The Performance of Different Ranking Methods

Methods	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
XGBoost	17.61	12.21	10.01	8.93	15.48	13.82
SVR	16.99	11.83	9.74	8.73	15.22	13.46
GRU	17.64	12.48	10.37	9.34	15.72	14.10
LSTM	17.70	12.49	10.36	9.32	15.85	14.18
LSTM+Att	17.74	12.51	10.37	9.33	15.82	14.16
ConvNet	**23.88**	**15.61**	**12.17**	**10.51**	**22.02**	**18.51**

alternatives. In this section, we evaluate the performance of different methods for the ranking module. We choose XGBoost [46], Support Vector Regression (SVR) [69], GRU [32] and LSTM [29] with or without Attention Mechanism as the baselines for ranking. We compute tf-idf scores for the tokens in messages and `diffs` as features for training XGBoost and SVR. For the other baselines, we concatenate the hidden states of messages and `diffs` as the feature representations and then fed into a fully-connected layer for predicting the relevance score. The training loss functions are similar to the definition in ConvNet and all the hyper-parameters are well-tuned by grid search [58]. The comparison results are shown in Table 4. We can find that deep learning methods i.e., GRU, LSTM, LSTM+Att, outperform machine learning methods i.e., XGBoost, SVR. Specifically, XGBoost presents a better performance as compared to SVR as it combines a set of classification and regression trees (CART) [70] to gradually reduce prediction errors by each iteration. The superior performance of ConvNet is because ConvNet adopts the similarity matching matrix (Section 3.5.1) to directly capture the relevance between `diffs` and candidate messages, for message ranking instead of concatenating their respective representations.

Answer to RQ4. *For predicting relevance between candidate messages produced by generation module and retrieval module, ConvNet is superior to traditional machine learning models, e.g., XGBoost and SVR, and sequential deep learning models, e.g., GRU and LSTM.*

4.4 Human Study

We conduct a human evaluation to evaluate ATOM with the best retrieval model NNGen [7] and the best generation model $\text{NMT}_{(\text{Luong})}$ [15]. We invite 4 PhD students and 2 master students from the department of computer science to participate in our survey. None of the participants is co-authors of this paper and they all have software development experience in Java programming language (raging $1 \sim 5$ years).

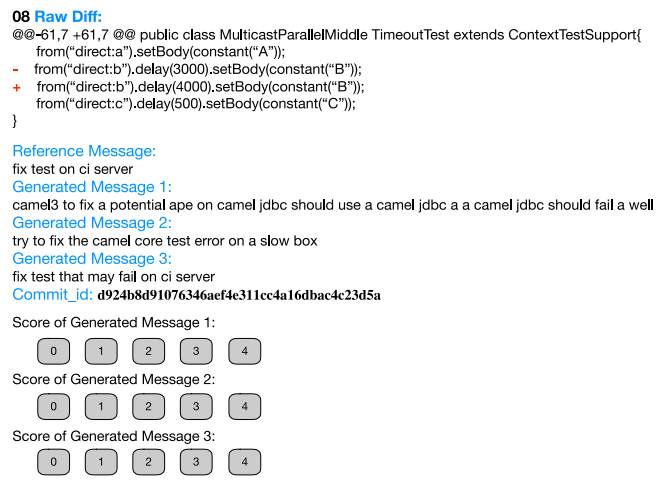


Fig. 9. A case of the questionnaire, provided with RAW Diff, followed by Reference Message and Generated Messages to score. We also provide commit id in case of participants to search on the Internet.

4.4.1 Survey Design

We randomly selected 100 commits from the test dataset for each participant to read and assess. In our questionnaire, each question first presents the code changes of one commit, i.e., its `diff`, its reference message, and messages produced by NNGen, $\text{NMT}_{(\text{Luong})}$, and ATOM respectively. Each participant is asked to give three quality scores between 0 to 4 to indicate the semantic similarities between the reference message and the three generated messages. Lower scores mean the generated messages are less identical to the reference messages. Fig. 9 shows one question in our survey. Participants are told the first message is the reference message, but the others are not aware of which message is generated by which approach and the three messages are randomly ordered. They are asked to score each generated message separately. Furthermore, we provide the commit id to help participants to search related information through the Internet.

4.4.2 Survey Results

Each code change and commit message pair is evaluated by 6 participants. Our scoring criterion is listed at the beginning of each questionnaire to guide participants, which follows Liu *et al.*'s work [7], e.g., score 0 means two messages have no shared tokens and score 1 denotes they have some shared tokens, but without semantic similarity. Score 2 can have some similar information but lacking important parts and score 3, 4 denotes two messages are very similar in semantics or even identical. We finally obtain 600 pairs of scores from our human evaluation. Each pair contains corresponding scores for the messages generated by $\text{NMT}_{(\text{Luong})}$, NNGen, and ATOM, respectively. Table 5 shows the score distribution of the generated commit messages based on the three methods. We can find that ATOM receives the best score and improve the average scores of $\text{NMT}_{(\text{Luong})}$ (1.32)

TABLE 5
The Score Distribution of the Generated Commit Messages by NNGen, $NMT_{(Luong)}$, and ATOM

Methods	0	1	2	3	4	≥ 3	≥ 2	≤ 1	Avg. (STD)	BLEU-4	PCC
NNGen	22	28	27	15	8	23	50	50	1.59 (0.68)	8.81	0.17
$NMT_{(Luong)}$	32	26	26	10	6	16	42	58	1.32 (0.83)	6.17	0.06
ATOM	18	25	31	16	10	26	57	43	1.75 (0.59)	10.23	0.21

The standard deviation is illustrated beside the average score.

and NNGen (1.59) to 1.75. Furthermore, our approach can generate more high-quality messages (scores $\geq$ 3) than NNGen and $NMT_{(Luong)}$. By comparing with the quantitative evaluation results of the 100 sampled commits, i.e., the BLEU-4 scores listed in the last column of Table 5, we can observe that they are consistent with the human ranking results. We then employ Pearson Correlation Coefficient (PCC) [71] to compute the correlations between the manual annotations and corresponding BLEU-4 scores for the 100 commits. The results also show that the messages generated by ATOM receive the most consistent scores between human study and the automated evaluation, with PCC score at 0.21. For the messages generated by NNGen and $NMT_{(Luong)}$, the PCC scores are relatively lower, at 0.17 and 0.06, respectively.

We also conduct inter-rater agreement analysis [72] on the manual annotations to observe the consistency among participants. We find that the agreement rates for the studied three approaches NNGen, $NMT_{(Luong)}$, and ATOM are 51.78, 45.32, and 62.60 percent, respectively. Considering each commit is annotated by six participants and the difficulty of the task, the agreement rates are reasonable and acceptable [7]. The result also implies that the generated messages by ATOM present the highest quality among all the generated messages.

Overall, by our human study, we can conclude that ATOM can produce more semantically related results with the ground-truths.

4.5 Examples

We show some examples to analyze the strengths and weaknesses of ATOM. Two examples of the generated messages by ATOM, NNGen, $NMT_{(Luong)}$, and the ground truth are illustrated in Table 6. From Example 1, we can find that both messages generated by NNGen and $NMT_{(Luong)}$ fail to describe the code change. For NNGen, since it directly recommends the message of the `diff` from the training set, it may fail when no relevant `diffs` appear in the training set. The generated message by NNGen contains words such as "camel3" and "npe" which are obviously unrelated to the `diff`. For $NMT_{(Luong)}$, it uses a sequence of code tokens as input, which may not accurately capture the semantics of a code change. As shown in Example 1, $NMT_{(Luong)}$ does not recognize that the `diff` is used to fix test. In contract, ATOM utilizes ASTs to capture the semantics of the `diff` and can generate a more accurate commit message.

As shown in Example 2 of Table 6, all the generated messages fail to detail that the code change is related to "jmstype header". This may be because the textual information, e.g., logs, in code changes is not well exploited and attended. In future, we will adopt text mining techniques such as part-of-speech analysis to fully capture the semantics in textual information of `diffs`.

5 Discussion

In this section, we describe the strengths of *AST2seq* as compared to NMT approaches, then provide more details about our benchmark compared with Jiang's [8] , and present the difference with Code2seq and Commit2Vec. Then we give a discussion about OOV issue in ATOM and finally discuss the limitations of ATOM.

5.1 Strengths of *AST2seq*

Previous studies, e.g., NMT [8], [15], Ptr-Net [61] treated `diff` as a flat sequence of tokens, which ignored code semantic information. To address this limitation, CODISUM [50] extracted code structure and code semantics based on identifying all the class/method/variable names and segmenting with the corresponding placeholders. By this way, they achieved BLEU-4 of 2.19 on Jiang's [8] dataset. Although they claimed that they achieved the highest BLEU-4 over NMT methods on Jiang's dataset, the

TABLE 6
Examples of Generated Commit Messages

Example	Example 1	Example 2
Commit_id	d924b8d91076346aef4e311cc4a16dbac4c23d5a	47a7eabe91fc8e8d26518bd092e62cdc7570d9af

diff	`@@ -61,7 +61,7 @@`
	`public class MulticastParallelMiddleTimeoutTest extends ContextTestSupport {`
	`from("direct:a").setBody(constant("A"));`
	`- from("direct:b").delay(3000)`
	`.setBody(constant("B"));`
	`+ from("direct:b").delay(4000)`
	`.setBody(constant("B"));`
	`from("direct:c").delay(500)`
	`.setBody(constant("C"));`
	`}`
		`@@ -183,7 +183,7 @@`
		`public class JmsBinding {`
		`try {`
		`map.put("JMSType", jmsMessage.getJMSType());`
		`} catch (JMSException e) {`
		`- LOG.trace("Cannot read JMSReplyTo header. Will ignore this exception.", e);`
		`+ LOG.trace("Cannot read JMSType header. Will ignore this exception.", e);`
		`}`

Ground-Truth	fix test on ci server	oracleaq do not support jmstype header
NNGen	camel3 to fix a potential npe on camel jdbc	oracleaq do not work
$NMT_{(Luong)}$	try to fix the camel core test error on a slow box	ensure stack trace be in trace log of exception
ATOM	**fix test that may fail on ci server**	**oracleaq do not support**

TABLE 7
Results of diffs With Different Lines Rather Than Tokens

diff lines	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor	Number	Ratio
1-10	**25.04**	**16.89**	**13.22**	**11.37**	**24.44**	**20.35**	1996	13.60%
10-30	22.62	14.70	11.58	10.10	21.98	17.79	3435	23.41%
30-50	23.16	14.62	10.96	9.17	22.86	18.50	2652	18.07%
50-100	24.26	16.02	12.53	10.87	23.65	19.28	**3670**	**25.01**%
100+	21.63	13.75	10.68	9.24	22.52	17.70	2921	19.90%

For example the upper left 1-10 *in the* diff *Lines column represents the commits with at most 10 lines of* diffs.

performance is still far away from satisfaction, which encourages us to do further exploration.

Many methods with the same functionality by a different implementation tend to have different surface forms, which is particularly common in the "For" and "While" statements. However, NMT-based approaches essentially treat diffs as a sequence of tokens, which hinders from capturing the semantics as the diverse expression format. However, AST is a high abstraction of code snippet and it transfers methods from plain text to tree structure. In many cases, methods with the same functionality share similar AST structures. Therefore, encoding AST to learn code semantics can seem as a refinement of original source codes and the recurring patterns might be easier to capture.

In addition, to easily capture semantics among diffs to represent code changes, another advantage for *AST2seq* is the ability to handle longer code changes. In sequence-based approaches [8], [15], [50], [61], they need to set maximum sequence, e.g., 100 tokens in total [8] for effective learning, which will lead to filter out a commit with too many chunks. Hence the sequence-based models cannot translate a commit with long sequences. However, *AST2seq* can effectively address this limitation. We extract paths between leaf nodes and combine them to represent code changes instead of treating diffs as a flat sequence. The number of sampled paths in *added* and *deleted* ASTs is set to 80 separately and the larger will be truncated. By this way, *AST2seq* is able to handle longer diffs and the results about the performance among different diff length are shown in Table 7, where the left column is the diff lines rather than token length. The BLEU-4 within 10 lines diffs is 11.37 and it takes up 13.60 percent in the whole testset. When diff lines increase to 100+, the performance only decreases by 2.13, 1.92, and 2.65 in terms of BLEU-4, ROUGE-L, and Meteor. Moreover, the performance with lines within 50-100 is better than the lines within 10-30 and 30-50. Therefore, the performance will not decrease dramatically along with the increased diff lines. Hence, *AST2seq* uses ASTs to encode code changes addressing the limitation of sequence length.

To sum up, AST2seq *utilizes AST into the encoder to learn the semantics behind the code changes and can handle longer* diffs*, which is superior to the existing approaches.*

5.2 Our Benchmark

We crawl our benchmark from 56 popular java projects ranked by "star numbers". We have devoted substantial efforts to clean the dataset and compared with Jiang's dataset [8], ours is able to serve as more research purposes.

Specifically, we store commits in a format file with various attributes including "commit_id", "subject", "commit message", "diff", and "file_changed". Note that the "subject" refers to the first sentence extracted from the commit messages, which can be seen as the summary of a message [8]. "file_changed" is the number of files that the current commit made.

Moreover, we also provide the extracted *added/deleted* functions from commits. For each commit, we extract the related functions. We name these functions in a format of "project_id_positive(negative)_num.java", where "project" represents the commit belonging to which project, "id" is the hash value and "positive/negative" denotes the *added/deleted* functions and "num" is the number of extracted functions.

Finally, the noisy commits [7], e.g., bot messages, which refers to messages generated by development tools and trivial messages, which contains little information, have been filtered automatically, since we keep commits modified in *.java* files and these boot messages and trivial messages most exist in configuration files, e.g., **.md*, **.gitrepo*. Furthermore, we remove the same content of the commits to ensure the benchmark has a higher quality compared with Jiang's [8].

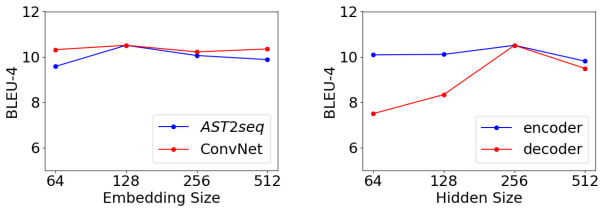
The benchmark contains the basic commit information and the completed functions altered by commits. Hence with this dataset, we can boost some other researches, e.g., code summarization, code recommendation, knowledge graph construction based on commits.

The benchmark we prepared contains adequate information as compared to Jiang's [8], and we make it publicly available [14] to benefit community research.

5.3 Novelty of ATOM

Our generation module *AST2seq* is inspired by Code2-seq [20], however, it has the following major differences.

- *Input Handling*. Although Code2seq adopts ASTs to encode source code for tasks such as code captioning, code documentation, and code summarization, it exploits function-level ASTs. For the proposed *AST2seq*, only partial function fragments, i.e., code changes, are considered for generating commit messages, which is more challenging. To construct ASTs for the code changes, we retrieve the whole function, including both *added* and *deleted* functions, and extract the AST paths corresponding to the changed



(a) Embedding size of *Ast2seq* and ConvNet. (b) Hidden Size in encoder and decoder.

Fig. 10. Effect of different parameter settings.

code. To the best of our knowledge, we are the first to incorporate ASTs for the commit message generation.

- *Model Design.* Code2seq employs a bi-directional LSTM to encode function-level ASTs. For *AST2seq*, *added* and *deleted* code fragments are treated separately. *AST2seq* first learns the representations of *added* and *deleted* code based on their respective ASTs by using two bi-directional LSTMs. Then the code change representations, i.e., `diffs`, are obtained by concatenating the two learned features.

In summary, although both AST2seq *and Code2seq utilize ASTs and bi-directional LSTM to learn code representations, they are different in input code handling and model design.*

5.4 Influence of Parameter Setting

Besides the number of AST paths (discussed in Section 4.3.3), we also analyzed the influence of other parameters, including the the embedding size [73] in *AST2seq* and ConvNet, and the number of hidden size [73] in encoder and decoder, on the model performance. The experimental results are depicted in Fig. 10. As shown in Fig. 10a, when the embedding sizes of *AST2seq* and ConvNet equal to 128, ATOM achieves the best performance. According to Fig. 10b, the number of hidden size in decoder can influence the model performance more obviously than the parameter in encoder. For example, ATOM shows a dramatic increase when the number of hidden size in the decoder increases from 64 to 256; however, the corresponding fluctuation for the parameter in encoder is marginal. We define the numbers of hidden size in encoder and decoder as 256 due to the good performance in the experiment.

5.5 OOV Issue

Some existing studies [50], [64] utilize strategies such as copy mechanism [74] for alleviating the Out-Of-Vocabulary (OOV) issue, i.e., some tokens in the test set have not appeared in the fixed vocabulary built during training. The OOV issue could lead the proposed model to generating wrong commit messages in our scenario. The copy mechanism [74] can learn to copy some tokens directly from the input `diffs` instead of only generating tokens based on the fixed vocabulary, and thereby mitigating the OOV issue. However, for our benchmark dataset, the vocabulary built during training contains 90,969 unique tokens from the `diffs` and commit messages, and only 1,930/14,674 (13.2 percent) samples in the test set involve the tokens not appearing in the vocabulary, i.e., the OOV tokens. Based on the observation, we suppose that using strategies such as copy mechanism might not improve much on ATOM.

TABLE 8
Time Costs of NNGen, $\text{NMT}_{(\text{Luong})}$ and ATOM

Methods	Device	Training Time	Testing Time
NNGen	CPU	N/A	308 secs
$\text{NMT}_{(\text{Luong})}$	Tesla P40	11 hours	188 secs
ATOM	Tesla P40	16 hours	257 secs

Since NNGen does not need training, its training time is marked as "N/A".

5.6 Limitations

5.6.1 Model Complexity

ATOM encodes code changes based on AST to represent code semantics and further designs a ranking module for more accurate commit message generation. It contains two modules involved with deep learning approaches, which cost time and efforts to tune the best models. The complexity of extracting AST paths from functions based on `diffs` is far more than treating `diffs` as sequences during the preprocessing. Furthermore, the output produced by the retrieval module is incorporated into the generation module to make the final decision, which is a complicated pipeline and the workload is much bigger than the previous work. Once ATOM is applied to the new benchmark, we still need to spend time and efforts to finish the preprocessing and model tuning. This can be considered as a limitation of ATOM, however, it is inevitable for all deep learning approaches and once *AST2seq* and ConvNet are fixed with the best parameters, the generation process is relatively low-cost and convenient.

We provide a deep analysis on the efficiency by comparing ATOM with the best retrieval-based approach, NNGen, and the best generation-based approach, $\text{NMT}_{(\text{Luong})}$. The comparison experiments are conducted on the same server with 36 cores and Nvidia Graphics Tesla P40 with 22 GB memory. The comparison results are listed in Table 8. As can be seen from the table, ATOM costs more time on training and testing than $\text{NMT}_{(\text{Luong})}$, which reflects the more complexity of ATOM than $\text{NMT}_{(\text{Luong})}$. Although training is unnecessary for NNGen, it spends the most time (308 secs) on testing since GPU cannot be used for acceleration.

Although ATOM takes much time to tune the hyperparameters to get a best model, once the model is fixed, its application is efficient.

5.6.2 Dataset Partition

Split by Project. In this paper, we follow the prior studies on commit message generation [8], [15], [16], [50], [61] by splitting dataset by commit. According to LeClair and McMillan's study on code summarization [75], splitting dataset by "function" (in analogy with "commit" in our study) might cause information leakage from test set projects into the training or validation sets and should be

TABLE 9
The Performance of Different Approaches Based on the Dataset Split by Project

Methods	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
NNGen	5.02	1.39	0.42	0.15	0.05	0.04
$NMT_{(Luong)}$	4.48	0.98	0.00	0.00	0.04	0.03
ATOM	5.44	2.06	1.28	0.82	0.07	0.05

TABLE 10
The Performance of Different Approaches Split by Timestamp

Methods	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-L	Meteor
NNGen	10.28	4.83	3.15	2.49	10.34	8.41
$NMT_{(Luong)}$	6.69	2.70	1.25	1.47	9.45	6.11
ATOM	10.87	5.99	4.19	3.53	11.14	9.53

avoided. Following the study, we evaluate the performance of ATOM based on the dataset split by project instead of by commit. We compare ATOM with the best retrieval-based model NNGen and the best generation-based model $NMT_{(Luong)}$ in the study.

The comparison results are illustrated in Table 9. We find that ATOM outperforms baseline models and the performance of all models decreases compared to the performance when splitting dataset by commit. Although a reduced performance is reasonable and expectable based on the dataset split by project [75], the magnitude of the decline is extremely obvious in our scenario. This indicates that splitting dataset by project may not be applicable for the evaluation of the commit message generation task. We further analyze the reason behind the extremely poor performance when splitting dataset by project from three aspects: method, task, and benchmark.

- *Method.* Deep learning-based models generally require massive data to learn the prior knowledge. When splitting the dataset by project, no prior knowledge of the project in the testset will be learned during training. Thus, the performance are expected to be worse than dataset partition by commit.
- *Task.* The code summarization task studied in [75] is different from code commit message generation task. In code summarization, the Java projects in the experimental dataset adopt some similar functions, e.g., "*setter*" and "*getter*" [76]. So this part of knowledge from other projects can be helpful for summarizing code of an unknown project. However, in code commit message generation, code changes in one project may not appear in other projects, which hinders the knowledge adaption from the other projects.
- *Benchmark.* In the benchmark dataset, we filter out bot messages, trivial messages, and the same samples (see Section 5.2). The filtered messages tend to possess similar templates, e.g., "This commit renames a file" and "Changes to a package". Removing these messages increase the difficulty of generating commit messages based on the dataset split by project.

Split by Timestamp. The dataset splitting strategy based on either commit (following the prior studies [41], [50], [61] or project would render the training and test sets mingle with commit messages written at different timestamps. That is, the commits in the training set may be written after some commits in the test set, causing the model to learn "from the future". Such scenario may be unrealistic since "future data" are unavailable in practice. To mitigate the issue, we adopt another dataset splitting strategy, i.e., according to the committed timestamps. Specifically, for each project, we rank the commits in chronological order, and treat the earliest 90 percent as training set and the rest as test set. Comparison results based on the new dataset splitting strategy is illustrated in Table 10. As can be seen, ATOM outperforms the baseline approaches with respect to all the evaluation metrics. Besides, the achieved scores are relatively lower than when splitting the dataset by commit. The reduced performance may be because developers of one project generally write commits for code changes related to different functionalities at different timestamps. For example, during a period of time, the developers may focus on enhancing one functionality of the project, so more commits related to this functionality are written; while for a later period, more commits for a different functionality are posted. Some examples are illustrated in this link [77]. Using past commits for training may hinder the trained model to produce an accurate result for a later commit since the later commit may be related to a new functionality. Moreover, the developers serving for one functionality may be changed to writing another different functionality. The changing commit styles may also influence the prediction accuracy of the trained model. However, the data partition strategy can laterally verify the generalizability of a proposed model for code commit message generation task, and we encourage the future research to consider such data partition strategy during evaluation.

To sum up, ATOM shows superior effectiveness than baseline models when splitting dataset by project or timestamp. However, due to the characteristics of the commits, the best practise to split dataset is based on commit.

5.7 Threats to Validity

One of the threats to validity is about the collected dataset. Our dataset contains more information than Jiang's [8] with more volumes, but more data is always beneficial to deep learning models. With the dataset we crawled so far, we have already achieved the best performance, which indicates the effectiveness of our proposed approach. Another threat to validity is about human evaluation in Section 4.4. We ask 6 participants to evaluate the quality of 100 randomly selected commit messages according to the criterion [7]. However, we cannot guarantee the judgements of participants are fully in line with the criterion. Ideally, the scores obtained from 6 participants are more reliable than those labeled by 3 participants, which is a common strategy adopted by prior work [7]. Furthermore, the reproduction of NNGen [7] may introduce bias to the experimental results. To alleviate the threat, we have tried our best to read the paper carefully and consulted the authors about

the details to ensure our reproduction is correct. Finally, we only compare ATOM on our dataset with the baseline methods and get state-of-the-art performance. As Jiang's [8] dataset does not provide commit ids, we cannot extract the *Added* and *Deleted* ASTs to encode changes. Hence, we cannot verify the effectiveness of ATOM on Jiang's dataset. But we compared all existing generation and retrieval approaches with ours on our benchmark to illustrate the effectiveness of ATOM.

6 Related Work

Our work is inspired by two research lines of studies, including code commit message generation and code summarization. In this section, we discuss the most related work and compare them with ATOM.

6.1 Code Commit Message Generation

Previous commit message generation studies can be mainly categorized into three types according to the methodology: rule-based, retrieval-based, and deep-learning-based. Initial studies [9], [10], [78], [79] rely on pre-defined rules or templates to establish the connections between code changes and natural languages. For example, Buse *et al.* [9] use the templates based on control flows to generate commit messages. Shen *et al.* [79] extract code changes based on defined types of changed methods and corresponding formats (e.g., "replace <old method name> with <new method name>" is a defined format for renaming a method). ChangeSribe [10], [78] further takes the impact set of a commit into account along with the commit stereotype and type of changes using pre-defined metrics, then fills a pre-defined commit message template with the extracted information. Such rule-based approaches can be limited by the manually specified rules or templates, and may work inefficiently for the code changes not applicable to the rules.

The retrieval-based approaches [7], [11] regard a newly-arrived `diff` as a query and reuse the commit messages of the most similar code changes. Huang *et al.* [11] use the syntactic similarity and semantic similarity of code changes as a measurement to retrieve existing commit messages. NNGen [7] reuses the message of the nearest neighbour by computing the cosine similarity of `diff` vectors constructed by a bag-of-words model, which extends to include both codes changes and non-code changes. For these approaches, simply retrieving messages as the targets cannot guarantee the consistency of the variable/method names. Besides, the mapping relations between `diffs` and commit messages are not fully exploited.

Deep-learning-based approaches [8], [15], [50] treat code changes and commit messages as two different languages, and design neural machine translation (NMT) models to translate code changes into commit messages. For example, Jiang *et al.* [8] directly adopt NMT model to conduct the translation. Jiang *et al.* [16] also adopt NMT model to conduct the translation but with all the commit messages formatted based on ChangeScribe [10], [78]. CODISUM [50] propose to combine both code structure and code semantics to enrich the representations of code changes for a better generation, and use CopyNet to mitigate the OOV issue. Although the results for these approaches are promising, they still do not explicitly bridge the gap between code and natural languages.

Compared with the above works, ATOM encodes ASTs to represent code changes and fully takes advantages from both retrieved methods and deep-learning-based methods by involving a hybrid ranking module to boost the performance further, resulting in more accurate commit message generation than all the above works.

6.2 Code Summarization

Code summarization aims to generate brief natural language descriptions for code snippet and it evolves from rule-based [80], [81], [82], retrieval-based [83], [84], [85] to learning-based [86], [87], [88] approaches. Pre-defining some basic rules based on the important content from codes is one of the most common approaches for the generation. Sridhara *et al.* [81] design a framework with traditional program and natural language analysis to tokenize function/variable names to summarize the Java method. Furthermore, based on this framework, Moreno *et al.* [82] predefine rules to combine information to generate comments for Java classes.

Information retrieval approaches are widely used in summary generation tasks. Haiduc *et al.* [83] use Vector Space Model (VSM) and Latent Semantic Indexing (LSI), an information retrieval method, to index top-k terms from a function and find the most similar terms based on cosine distances as the summary. Rodeghero *et al.* [89] further improve the performance by improving the subset selection process by modifying the weights of the keywords from the codes based on the result of an eye-tracking study. McBurney *et al.* [90] apply topic modelling and design a hierarchy to organize the topics in source code, with more general topics near the top of the hierarchy to select keywords and topics as code summaries. Clocim [85] applies code clone detection to find similar codes and uses its comments directly.

In addition, some researchers try to generate summaries by learning-based approaches. Iyer *et al.* [87] propose CODE-NN, an attentional LSTM encoder-decoder network to generate C# and SQL descriptions. Hu *et al.* [91] further incorporate an additional encoder layer into the NMT model to learn API sequence knowledge. They first train an API sequence encoder using an external dataset, then apply the learned representation into the encoder-decoder model to assist generation. Wan *et al.* [88] also incorporate an abstract syntax tree as well as sequential content of code snippets into a deep reinforcement learning framework to translate python code snippets. Code2seq [20] model represents a code snippet as the set of paths in its AST to decode language sequences and the results outperform state-of-the-art NMT models. Different from code summarization, we aim at generating code changes, a higher target compared with the whole function summary.

7 Conclusion

Automatically generating commit messages is necessitated. Existing studies either translate `diffs` with sequence-based methods or retrieval-based methods. In this paper, we propose our ATOM to encode AST paths of `diffs` for code representation to generate commit messages. Furthermore,

we integrate the advantages of retrieval-based models by a hybrid ranking module to prioritize the most accurate message from both retrieved and generated messages. Substantial experiments based on our benchmark have demonstrated the effectiveness of ATOM and ATOM increases the state-of-the-art approaches by *30.72 percent* in BLEU-4. In future work, we plan to design a detailed specification to keep commits with higher quality and apply our proposed approach to other tasks such as code summarization, code documentation, and even source code generation.

Acknowledgment

We gratefully acknowledge the support of NVIDIA AI Tech Center (NVAITC) to our research. This work was supported by Singapore Ministry of Education Academic Research Fund Tier 1 (Award No. 2018-T1-002-069), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE003-0001, NRF Investigatorship NRFI06-2020-0022, and the National Natural Science Foundation of China under project No. 62002084.

References

[1] Git. [Online]. Available: http://git-scm.com
[2] Tortoisesvn. [Online]. Available: https://tortoisesvn.net/
[3] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 422–431.
[4] Sourceforge. [Online]. Available: https://sourceforge.net/
[5] Junit5. [Online]. Available: https://github.com/junit-team/junit5
[6] neo4j. [Online]. Available: https://github.com/neo4j/neo4j
[7] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 373–384.
[8] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 135–146.
[9] R. P. L. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proc. 25th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2010, pp. 33–42.
[10] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Proc. 14th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, 2014, pp. 275–284.
[11] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo, "Mining version control system for automatically generating commit comment," in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2017, pp. 414–423.
[12] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.
[13] The code of ATOM. [Online]. Available: https://github.com/shangqing-liu/ATOM
[14] The data of ATOM. [Online]. Available: https://zenodo.org/record/4077754#.X4K2b5MzZTY
[15] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 287–292.
[16] S. Jiang, "Boosting neural commit message generation with code semantic analysis," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 1280–1282.
[17] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," 2019, *arXiv: 1909.03496*.
[18] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017, *arXiv: 1711.00740*.
[19] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," 2018, *arXiv: 1811.01824*.
[20] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2018, *arXiv: 1808.01400*.
[21] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," in *Proc. 9th Int. Conf. Artif. Neural Netw.*, 1999, pp. 850–855.
[22] C. Gao, J. Zeng, X. Xia, D. Lo, M. R. Lyu, and I. King, "Automating app review response generation," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 163–175.
[23] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, 2019, Art. no. 40.
[24] Javaparser documentation. [Online]. Available: https://javaparser.org/
[25] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. 26th Conf. Program Comprehension*, 2018, pp. 200–210.
[26] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.
[27] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
[28] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*.
[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[30] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*.
[31] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," 2015, *arXiv:1508.04025*.
[32] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014, *arXiv:1412.3555*.
[33] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[34] O. Levy, M. Seo, E. Choi, and L. Zettlemoyer, "Zero-shot relation extraction via reading comprehension," 2017, *arXiv: 1706.04115*.
[35] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple object recognition with visual attention," 2014, *arXiv:1412.7755*.
[36] B. Hu, Z. Lu, H. Li, and Q. Chen, "Convolutional neural network architectures for matching natural language sentences," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 2042–2050.
[37] L. Pang, Y. Lan, J. Guo, J. Xu, S. Wan, and X. Cheng, "Text matching as image recognition," in *Proc. 13th AAAI Conf. Artif. Intell.*, 2016, pp. 2793–2799.
[38] Pygments documentation. [Online]. Available: http://pygments.org/
[39] P. Niemeyer and J. Knudsen, *Learning Java*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005.
[40] Exuberant Ctags documentation. [Online]. Available: http://ctags.sourceforge.net/
[41] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *Proc. 25th Int. Conf. Program Comprehension*, 2017, pp. 320–323.
[42] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 631–642.
[43] Natural language toolkit NLTK documentation. [Online]. Available: https://www.nltk.org/
[44] Scikit-learn. [Online]. Available: https://scikit-learn.org/stable/
[45] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Inf. Process. Manage.*, vol. 39, no. 1, pp. 45–65, 2003.
[46] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 785–794.
[47] L. Yang *et al.*, "A hybrid retrieval-generation neural conversation model," 2019, *arXiv: 1904.09068*.
[48] W. James and C. Stein, "Estimation with quadratic loss," in *Breakthroughs in Statistics*. Berlin, Germany: Springer, 1992, pp. 443–460.

[49] Y. Song, R. Yan, C.-T. Li, J.-Y. Nie, M. Zhang, and D. Zhao, "An ensemble of retrieval-based and generation-based human-computer conversation systems," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 4382–4388.
[50] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, 2019, pp. 3975–3981.
[51] Structs. [Online]. Available: https://github.com/fatih/structs
[52] Antlr4. [Online]. Available: https://github.com/antlr/antlr4
[53] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1050–1059.
[54] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1171–1179.
[55] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[56] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.
[57] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 8026–8037.
[58] L. Franceschi, M. Donini, P. Frasconi, and M. Pontil, "Forward and reverse gradient-based hyperparameter optimization," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 1165–1173.
[59] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Proc. Text Summarization Branches Out*, 2004, pp. 74–81.
[60] S. Banerjee and A. Lavie, "Meteor: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.
[61] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, "Generating commit messages from diffs using pointer-generator network," in *Proc. 16th Int. Conf. Mining Softw. Repositories*, 2019, pp. 299–309.
[62] R. C. Lozoya, A. Baumann, A. Sabetta, and M. Bezzi, "Commit2Vec: Learning distributed representations of code changes," 2019, *arXiv: 1911.07605*.
[63] The code of NMT. [Online]. Available: https://github.com/tensorflow/nmt
[64] The code of Ptr-Net. [Online]. Available: https://zenodo.org/record/2542706#.XxPkmvgzZTa
[65] The code of codisum. [Online]. Available: https://github.com/SoftWiser-group/CoDiSum
[66] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[67] S. van Hal, M. Post, and K. Wendel, "Generating commit messages from git diffs," 2019, *arXiv: 1911.11690*.
[68] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 1073–1083.
[69] H. Drucker, C. J. Burges, L. Kaufman, A. J. Smola, and V. Vapnik, "Support vector regression machines," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 1997, pp. 155–161.
[70] R. J. Lewis, "An introduction to classification and regression tree (CART) analysis," in *Proc. Annu. Meeting Soc. Academic Emergency Med.*, 2000.
[71] K. Pearson, "Notes on regression and inheritance in the case of two parents," *Proc. Roy. Soc. London*, vol. 58, pp. 240–242, 1895.
[72] K. A. Hallgren, "Computing inter-rater reliability for observational data: An overview and tutorial," *Tuts. Quantitative Methods Psychol.*, vol. 8, no. 1, 2012, Art. no. 23.
[73] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "CORE: Automating review recommendation for code changes," in *Proc. IEEE 27th Int. Conf. Softw. Anal. Evol. Reeng.*, 2020, pp. 284–295.
[74] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," 2017, *arXiv: 1704.04368*.
[75] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," 2019, *arXiv: 1904.02660*.
[76] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2019, pp. 602–614.
[77] The examples of dataset split by timestamp. [Online]. Available: https://docs.google.com/spreadsheets/d/1oOuPZObIrBPTcf9lpea9HIlQr6sG_LfspRBRJ_lbhdY/edit#gid=1589976611
[78] M. Linares-Vásquez, L. F. Cortes-Coy, J. Aponte, and D. Poshyvanyk, "ChangeScribe: A tool for automatically generating commit messages," in *Proc. 37th IEEE/ACM Int. Conf. Softw. Eng.*, 2015, pp. 709–712.
[79] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *Proc. 40th IEEE Annu. Comput. Softw. Appl. Conf.*, 2016, pp. 103–112.
[80] 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 *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 561–565.
[81] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2010, pp. 43–52.
[82] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proc. 21st Int. Conf. Program Comprehension*, 2013, pp. 23–32.
[83] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 223–226.
[84] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 35–44.
[85] E. Wong, T. Liu, and L. Tan, "CloCom: Mining existing source code for automatic comment generation," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reeng.*, 2015, pp. 380–389.
[86] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.
[87] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 2073–2083.
[88] Y. Wan *et al.*, "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 397–407.
[89] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 390–401. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568247
[90] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in *Proc. 22nd Int. Conf. Program Comprehension*, 2014, pp. 291–294.
[91] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 2269–2275. [Online]. Available: https://doi.org/10.24963/ijcai.2018/314

Shangqing Liu received the master's degree from the College of Computer Sciences and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China, 2018. He is currently working toward the PhD degree in the School of Computer Science and Engineering, Nanyang Technological University (NTU), Singapore. He was an intern with NTU, from December 2017 to June 2018. His research interests include AI for SE, graph neural network, and natural language processing.

Cuiyun Gao received the BEng degree from the Department of Communication Engineering, Shanghai University, Shanghai, China, in 2014, and the PhD degree from the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, in 2018. She is currently an assistant professor with the Harbin Institute of Technology, Shenzhen. Her research interests include software repository mining and natural language processing. She had published more than 20 peer-reviewed publications in conferences and journals in her area of expertise. She also served as reviewer for many top-tier conferences and journals.

Sen Chen (Member, IEEE) received the PhD degree in computer science from the School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in June 2019. He was a research assistant professor with Nanyang Technological University (NTU), Singapore, from July 2020, a research fellow from 2019 to June 2020, and a research assistant from 2016 to June 2019. He is currently a tenured associate professor with the College of Intelligence and Computing, Tianjin University. His research interests include mobile security, AI security, open-source security, and software testing.

Lun Yiu Nie received the bachelor's degree from the Chinese University of Hong Kong, Hong Kong, in 2020. He is currently working toward the master's degree in computer science in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests mainly include AI for systems and natural language processing.

Yang Liu received the bachelor's and PhD degrees in computer science from the National University of Singapore (NUS), Singapore, in 2005 and 2010, respectively, and continued with his postdoctoral work with NUS. He is currently an full professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His research focuses on software engineering, security, cyber-physical systems, and formal methods. Particularly, he specializes in software verification using model checking techniques, leading to the development of a state-of-the-art model checker, process analysis toolkit (PAT).