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

Shangqing Liu, Cuiyun Gao, Sen Chen, 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 code structure of changed code, 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 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% 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.

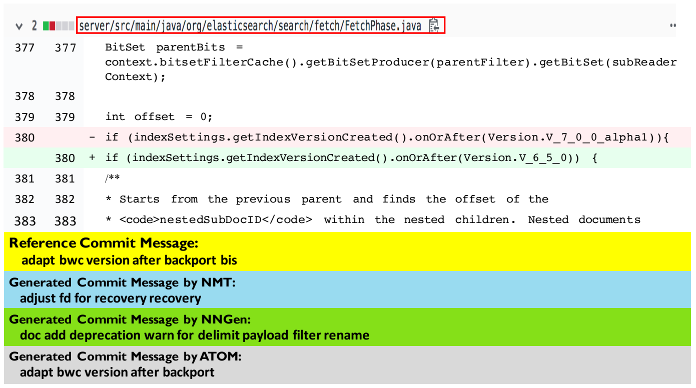
Index Terms—Commit Message Generation, Code Changes, Abstract Syntax Tree

1 Introduction

With software growing in size and complexity, version control systems, e.g., GitHub [1] and TortoiseSVN [2], have been widely adopted in the life cycle of software development. These version control systems greatly reduce the time cost. During software updating, developers are required to submit commit messages to version control systems 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 labor-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, it is non-trivial for readers to extract the precise description behind code changes manually, and developers tend to commit without writing the corresponding messages. 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% of commit messages in more than 23,000 open-source Java projects are empty. During our crawling of 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% of the whole collected commits. Therefore, automated generation of commit messages for code changes is necessitated and helpful for software developers.

Generating accurate commit messages given code



```
v 2 server/src/main/java/org/elasticsearch/search/fetch/FetchPhase.java
      BitSet parentBits =
      context.bitsetFilterCache().getBitSetProducer(parentFilter).getBitSet(subReader
      Context);

      int offset = 0;
    - if (indexSettings.getIndexVersionCreated().onOrAfter(Version.V_7_0_0_alpha1)){
    + if (indexSettings.getIndexVersionCreated().onOrAfter(Version.V_6_5_0)) {
      /**
      * Starts from the previous parent and finds the offset of the
      * <code>nestedSubDocID</code> within the nested children. Nested documents
```

Reference Commit Message:
adapt bwc version after backport bis

Generated Commit Message by NMT:
adjust fd for recovery recovery

Generated Commit Message by NNGen:
doc add deprecation warn for delimit payload filter rename

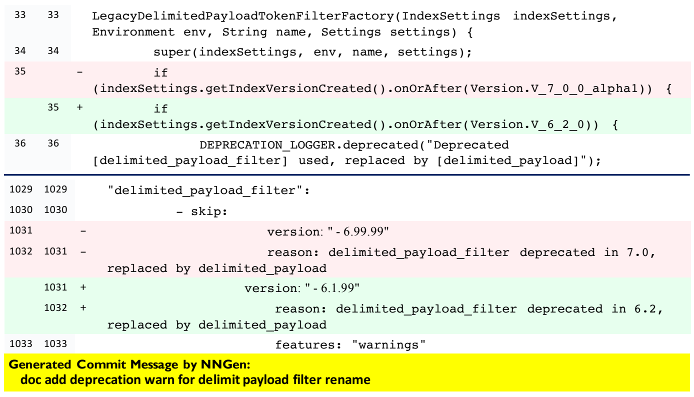
Generated Commit Message by ATOM:
adapt bwc version after backport

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*.

• *Shangqing Liu, Cuiyun Gao, Sen Chen, and Yang Liu are with School of Computer Science and Engineering, Nanyang Technological University, Singapore. E-mail: shangqin001@e.ntu.edu.sg, {cuiyun.gao, chensen, yangliu}@ntu.edu.sg*
• *Lun Yiu Nie is with The Chinese University of Hong Kong, China. E-mail:lynie8@cse.cuhk.edu*
• *Cuiyun Gao is the corresponding author.*

Manuscript received December 7, 2019

1. Meaningless refers to empty, non-ASCII, merge and rollback commits. Merge and rollback commits are removed as they often contain too many lines and these commits can be identified by the keywords ”Merge”, ”Rollback”.



```
    LegacyDelimitedPayloadTokenFilterFactory(IndexSettings indexSettings,
    Environment env, String name, Settings settings) {
            super(indexSettings, env, name, settings);
-           if
    (indexSettings.getIndexVersionCreated().onOrAfter(Version.V_7_0_0_alpha1)) {
+           if
    (indexSettings.getIndexVersionCreated().onOrAfter(Version.V_6_2_0)) {
                  DEPRECATION_LOGGER.deprecated("Deprecated
    [delimited_payload_filter] used, replaced by [delimited_payload]");
```

```
     "delimited_payload_filter":
              - skip:
-                         version: " - 6.99.99"
-                         reason: delimited_payload_filter deprecated in 7.0,
     replaced by delimited_payload
+                      version: " - 6.1.99"
+                          reason: delimited_payload_filter deprecated in 6.2,
     replaced by delimited_payload
                          features: "warnings"
```

Generated Commit Message by NNGen:
doc add deprecation warn for delimit payload filter rename

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

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 problem, Jiang et al. [8] proposed to adopt a neural machine translation (NMT) model for translating code changes into commit messages, where the NMT model is assumed to be able to learn semantic mapping relations between the two sources. However, the generation model prefers frequent tokens, such as ”adjust”, ”recovery” and the generating messages are more unreadable. Furthermore, the NMT model treats code as a flat sequence of tokens, and ignores the syntactic and semantic structure within code snippets, which can be useful for code representations. Some other researchers [7], [11] attempt to reuse the existing commit messages in the collected dataset by using information retrieval approaches to get the best performance. However, the retrieval models ignore its context and mainly rely on token frequencies to retrieve similar source code, which may produce wrong results. For example, as shown in Fig. 1, the retrieval approach, NNGen proposed by Liu et al. [7], focused on similar codes and the message it retrieves may be less related to the current commit. To be more specific, Fig. 2 shows the code changes of retrieved commit whose first part is similar to the example in Fig. 1, however the second part contributes more to its message. In this paper, we aim at representing code semantics by considering code structures, and integrating the advantages of retrieval-based models for more accurate commit message generation.

To this end, we propose a novel commit message generation model, named ATOM (**A**bstract syntactic **T**ree-based c**O**mmit **M**essage generation). Instead of directly using code tokens to represent code semantics, we propose a novel code representation approach based on AST (an abbreviation of Abstract Syntax Tree) to encode code changes. Besides, ATOM involves a hybrid ranking module to adaptively prioritize the most relevant commit messages from the generated and retrieved messages to boost the performance of generation module. Specifically, ATOM mainly contains three modules, including a generation module, a retrieval module, and a hybrid ranking module. The generation module encodes the structure of changed code, i.e., AST, to enrich the semantic representation. Furthermore, the hybrid ranking module 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 dataset since AST cannot be constructed in the previous benchmark dataset [8]. We quantitatively evaluate ATOM on our crawled dataset, including $\sim$200k commits in total. Experimental results demonstrate that ATOM can significantly outperform the state-of-the-art models by increasing at least 30.72% in terms of BLEU-4 score [13] (an accuracy measure that is widely used to evaluate text generation systems). Besides, ATOM can enhance the performance of its generation module by 42.99%. Human evaluation done through a user study further confirms the superior performance of ATOM than the baselines. Besides reporting the promising results, we investigate the reason behind the performance of ATOM and the key constraints on accurate commit message generation.

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 dataset, including complete function-level code snippets of $\sim$200k commits from 56 java projects. We clean the dataset by filtering out meaningless (e.g., empty, non-ASCII, merge) commits and make the dataset publicly available [2] 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 and show experimental results in Section 4. A human evaluation is conducted in Section 5 and Section 6 discuss the strengths of ATOM, followed by the related work in Section 7. Finally, Section 8 concludes this paper.

2 Background

In this section, we first introduce several attributes relevant to commit and 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

2. https://drive.google.com/commit_data

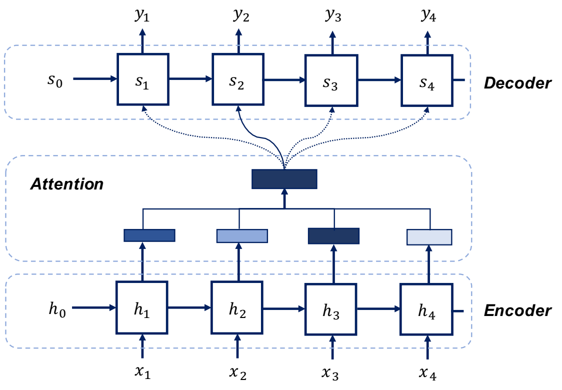


Fig. 3: An encoder-decoder model with an attention mechanism

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

Listing 1: A simple Java code snippet

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 commit change is called `diff` to characterize the difference between two code versions. Usually, a commit change may contain one or multiple chunks with file paths, which can be found at a red rectangle, as shown in Fig. 1, along with the identifier ”diff –git” to indicate which file is changed. The modified codes are wrapped by ”@@” in a chunk with the negative sign ’-’ or positive sign ’+’ together 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.

2.2 Abstract Syntax Tree (AST)

An abstract syntax tree is a high abstraction of source code, which is a tree structure, serves as the intermediate representation of program language. An AST usually contains leaf nodes that represent identifiers and names in the code and non-leaf nodes which can represent some syntact within codes. To be more specific, Fig. 4 shows a simple AST example 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 structures. We can get a total of 106 different non-leaf nodes with *JavaParser* [14], 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 [15], code2vec [16], DeepCom [12], CRF [17], Devign [18].

2.3 Encoder-Decoder Model

The basic structure for NMT [19] used to translate source sequences into targets is encoder-decoder, as shown in Fig. 3.

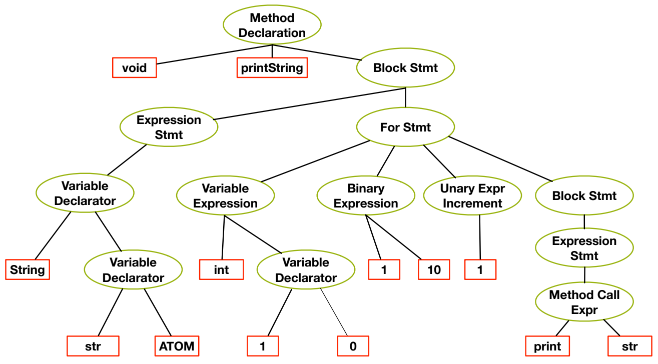


Fig. 4: The AST compiled from Listing 1

The feature vectors generated by encoder are fed into the decoder to generate target sequences. Usually, it consists two RNNs [20] with built-in LSTM cells [21] and attention mechanism [22] [23] for translation.

2.3.1 Recurrent Neural Network (RNN)

RNNs are widely used to capture information from time-series data as their chain-like natures. The loop containing 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. As RNNs are unable to connect information from further back and cannot handle long sequences, some variants e.g., Long Short-term Memory (LSTM) [21] and Gated Recurrent Unit (GRU) [24] are proposed.

2.3.2 Long Short-Term Memory (LSTM)

LSTMs introduced by [21] 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 long-term dependencies than vanilla RNNs. Hence, RNNs built with LSTMs are widely used for sequence models to capture information.

2.3.3 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 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 the decoder makes, 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 [22] or Luong [23] Attention has been widely adopted into neural machine translation [25], reading comprehension [26] and computer vision [27].

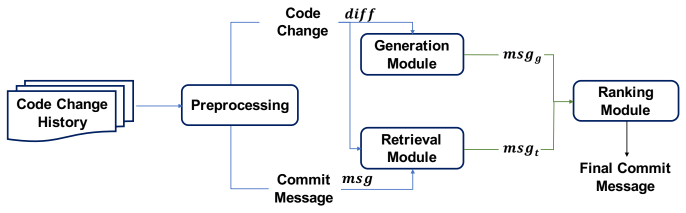


Fig. 5: Architecture of ATOM

2.4 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 architecture. 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 adopted to natural language processing tasks. In [28] and [29], CNN-based neural ranking models are trained to learn high level sentence matching patterns.

3 Our approach (ATOM)

In this section, we first briefly introduce the overview of our approach ,ATOM, for generating commit message, and then 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.
- *AST2seq Generation Module.* We encode AST paths from `diffs` with LSTM to represent code changes and use a decoder with attention mechanism to generate a new message msg_g.
- *Retrieval Module.* The retrieval module uses a "`diff-diff` match" approach to retrieve the most relevant commit messages. This approach matches `diff` with all `diffs` in the training set and get the most relevant corresponding message msg_t based on the `diff` similarity distance.
- *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 most relevant message msg_t.

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 *add* and *delete* groups based on the corresponding sign, i.e., "+" and "-". Then we tokenize the `diffs` using pygements [30], and remove meaningless tokens such as punctuations. Consequently, we obtain a list of tokens for the *add* code and *delete* code, denoted as W^+ and W^- respectively, where $W^{+/-} = \{w_1, w_2, ..., w_i\}$ and i means the i-th token in the changed code.

We extract AST paths corresponding to `diffs` based on the basic compilation unit [31], which contains a single class definition and wrapped functions. Hence, we need to retrieve completed functions of `diffs` denoted as *Added* function and *Deleted* function. We use Ctags [32] with file paths and modified line numbers containing in `diffs` to retrieve completed functions in the repository and then parse these functions to obtain ASTs with JavaParser [14]. As all tokens belonging to $W^{+/-}$ are leave node values in an AST, e.g., for any two tokens, w_i and w_j in $W^{+/-}$, we search the shortest distance [3] and denote the path as $x = \{w_i, n_1, ..., n_l, w_j\}$, where n_l means the l-th non-leaf node. Following the procedure, we finally obtain AST paths for the whole *add*/*delete* code, indicated as $X^{+/-} = \{x_1^{+/-}, ..., x_{p/k}^{+/-}\}$ where p, k are the total number of *Added* and *Deleted* AST paths.

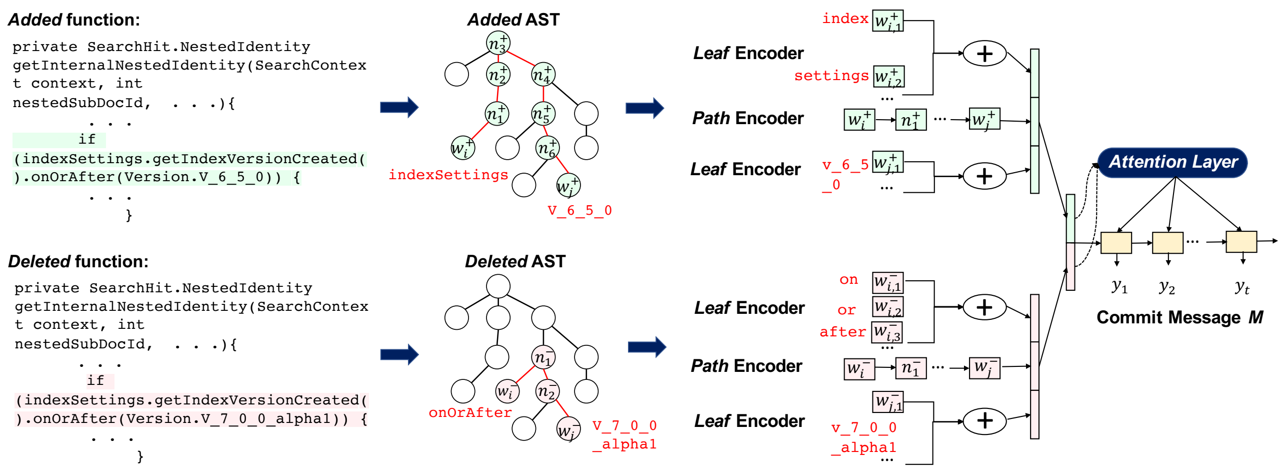
3.2.2 Commit Messages

We extract the first sentence from the commit message as the target sequence as the first sentence is often the summaries of the entire commit [33] [8] [34]. 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 using the NLTK toolkit [35] to reduce the vocabulary set. One lemmatized message is denoted as $M = \{y_1, y_2, ..., y_n\}$ 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 to be applied into long sequences and capture code semantics. Abstract Syntax Tree (AST) is an abstraction of code and has proven useful in representing code semantics [15] [16] [12]. 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 whole 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. Here the shortest distance refers to the minimum edges between two corresponding leave nodes.



```
private SearchHit.NestedIdentity
getInternalNestedIdentity(SearchContex
t context, int
nestedSubDocId,  . . .){
        . . .
        if
(indexSettings.getIndexVersionCreated(
).onOrAfter(Version.V_6_5_0)) {
        . . .
    }
```

```
private SearchHit.NestedIdentity
getInternalNestedIdentity(SearchContex
t context, int
nestedSubDocId,  . . .){
        . . .
        if
(indexSettings.getIndexVersionCreated(
).onOrAfter(Version.V_7_0_0_alpha1)) {
        . . .
    }
```

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 with tokens, e.g., indexSettings, onOrAfter in `diff` and $n_l^{+/-}$ is the type of non-leaf node, e.g., ForStmt, Binary Expression.

3.3.1 *AST Encoder*

Given a set of *Added* and *Deleted* AST paths $X = \{x_1^{+/-}, x_2^{+/-}, ..., x_{p/k}^{+/-}\}$, where $x \in X$ can be represented as $\{w_i, n_1, ..., n_l, w_j\}$ and p, k are the *Added* and *Deleted* AST paths. We encode each path x with bi-directional LSTM and associated leaf tokens w to create a vector representation z, which is concatenated by the path feature denoted as $path\_feat$ and the leaf feature represented as $leaf\_feat$.

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

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

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

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

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

- **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 path. We split the tokens of the values of leaf nodes e.g., onOrAfter in Fig. 6 into subtokens, on, or, after and then combine the embeddings of these subtokens using 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*

Different from the typical attention mechanism which takes tokens of source sequence as input, the attention mechanism in ATOM combines the learned vector representations of *Added* and *Deleted* paths, i.e., $Z = \{z_1, z_2, ..., z_{p+k}\}$ where $p + k$ is the summation of *Added* and *Deleted* paths. Luong Attention [23] with *general* score function is shown in the equation 10 where α^t is a alignment vector. During decoding, the attention will learn the weight distribution over these paths so that the relevant parts of paths can be captured. Specifically, it will learn how much attention should be given to the path in Z and adaptively aggregate the information from paths.

3.3.3 *Message Decoder*

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

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

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

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

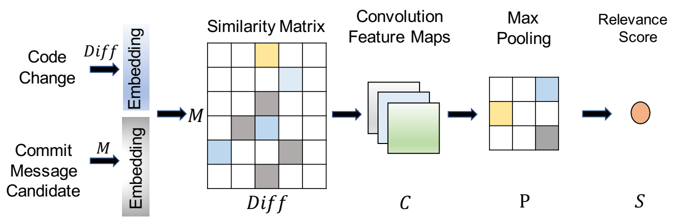


Fig. 7: Architecture of ConvNet

α^t is the variable-length alignment vecotr and its size equals the number of *Added* and *Deleted* paths. Then c_t and h_t are combined to predict the current token y_t [23]:

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

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 given a new `diff` or a `diff` in the test set. We adopt a "`diff-diff` match" strategy for the retrieval. Specifically, we first index all `diffs` in training set using sklearn [36]. Then for each `diff` in the validation and test sets, we compute the cosine similarity with each `diff` in the training set based on their tokens of tf-idf scores [37], and keep the most relevant one training commit message. TF (term frequency) and IDF (inverse document frequency) can be computed by the following equation:

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

where $n_{i,d}$ is the number of i_{th} token in the `diff` 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 whole `diffs` N_{diff}.

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

3.5 Hybrid Ranking Module

To incorporate the retrieval results into the generation module, we design a hybrid ranking module to adaptively rank the output of generation and retrieval. Specifically, the rank scores are based on a similarity matrix computed from `diffs` and messages, followed by a convolutional neural network.

3.5.1 Similarity Matching Matrix

From the retrieval module (described in Section 3.4) and generation module (described in Section 3.3), 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. We design a hybrid ranking module to rank both candidates to find the best one as the final output message [38]. The hybrid ranking module is based on ConvNet, as shown in Fig. 7. Specifically, with any `diff` d in validation and test sets, the ConvNet model first looks up embeddings for tokens in d and y respectively, denoted as $E(d) = [d_1, d_2, ..., d_{L_d}]$ and $E(y) = [y_1, y_2, ..., y_{L_y}]$ 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. Then we obtain the similarity matching matrix D which represents the distance between the `diff` and candidate:

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

where D has the dimension (L_d, L_y) and is fed into a convolutional neural network (CNN) model for predicting the relevance score between the `diff` and each candidate message.

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. The ConvNet model contains convolution and max-pooling operations on the similarity matching matrix D. Let C_{in} denote a 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:

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

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 [39] 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 between diffs and corresponding candidate messages. One possible solution is to manually evaluate these candidates; however, the time and labor 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 similarity metrics, e.g., BLEU-4 [38], [40]. 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 in training ConvNet. The trained ConvNet is fixed during validating and testing processes, so it can predict the more relevant commit messages with higher relevance scores.

4 Experimental Evaluation

In this section, we first introduce the experimental setup, including the subject dataset, experimental settings, evaluation metrics, and comparison methods, and then elaborate on the experimental results, specifically answering the following research questions.

- **RQ1**: What is the performance of ATOM comparing with baseline models?
- **RQ2**: What is the impact of the generation and retrieval module on the performance of ATOM?
- **RQ3**: How accurate is *AST2seq* under a different number of AST paths?
- **RQ4:** What is the impact of different ranking methods?

4.1 Setup

4.1.1 Experimental Dataset

The dataset utilized in previous studies [7], [8], [41] contains no commit ids or complete functions and we can not use this dataset directly as ASTs are not available. We crawled 56 popular projects including Neo4j [6], Structs [42], Antlr4 [43] from GitHub based on the project stars and obtained 715,190 commits in total. 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 contain too many lines, which is not suitable for generation module. So we filter them out to eliminate unrelated information and remain with 629,964 commits. Besides, some commits related to project initialization and fundamental functionality updating contain many changes, we remove them either. 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, i.e., 194,455 commits. We randomly select 10 % of the $\sim$200k data for testing, 10% for validation and the remaining for training.

4.1.2 Experimental Settings

For *AST2seq* method in the generation module, the max number of paths in *Added* and *Deleted* ASTs are set to 80. The embedding sizes for subtokens, paths and target sources are defined as 128. The encoder for path representation is one single-layer bidirectional LSTM, and the decoder is also bidirectional LSTM but with two layers. All dimensions of the hidden states are fixed to be 256. The probability of dropout [44] is set as 0.5 to avoid overfitting. We set the number of epochs equal to 3,000, along with the batch size as 256 and patience 20, a threshold to terminate training for early stopping. The learning rate is equal to 0.0001. During testing, we adopt beam search since it has proven useful in sequence prediction with recurrent neural network [45] and set the beam width as 5. For ConvNet training, we use a 2-D convolutional layer with the number of kernels defined as 16 and kernel size as $(3, 3)$, followed by a relu activation 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 [46] and use Tensorflow 1.12 [47] for our model training and all 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 metrics to evaluate text generation systems such as BLEU-4 [13], ROUGE [48], and Meteor [49]. These metrics have proven useful in measuring semantic similarities between the produced messages and ground truth. BLEU-4 calculates the similarity by computing the n-gram precision of a candidate sequence to the reference, with a penalty for overly short sentences. It is the product of brevity penalty (*BP*) and geometric average of the modified n-gram precisions, computed as:

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

where $N = 4$ and 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, as a modification of BLEU, focuses on recall instead. Specifically, ROUGE-N counts recall based on n-gram and ROUGE-L counts based on the longest common subsequence. In this paper, we set N to 1 and 2 respectively along with ROUGE-L as evaluation metrics. 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),

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

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

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

Furthermore, we also conduct a human evaluation to compare our ATOM with NNGen and NMT.

4.2 Comparison Methods

We choose the basic neural machine translation (NMT) models with different attention mechanisms, the retrieval-based method NNGen, the text summarization approach Ptr-Net [50], [51], and the state-of-the-art method CODISUM [41] as baseline models. The details of these models are illustrated below.

- NMT [8], [52], [53]. NMT adopts attention-based RNN encoder-decoder models, described in Section 2.3 to generate commit messages for `diffs`, which treats `diffs` and commit messages as different languages. Jiang et al. [8] uses Bahdanau attention [22] with Nematus [54], a tool for neural machine translation, to produce messages. Another recent approach Commitgen proposed by Loyola et al. [52] leverages Luong [23] attention instead of Bahdanau attention for commit message generation and both of them treat `diffs` as sequences, so we compare with both attention mechanisms denoted as $\text{NMT}_{(\text{Loyola et al.})}$ and $\text{NMT}_{(\text{Jiang et al.})}$ respectively.
- NNGen [7]. NNGen is essentially a retrieval-based approach. It retrieves the most similar top-*k* `diffs` from the training dataset based on a bag-of-words [55] 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.
- Ptr-Net [50], [51]. 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. [51].
- CODISUM [41]. CODISUM is the state-of-the-art approach which jointly models code structure and code semantics to learn the better representations of the code changes as well as combining pointer network [50] to mitigate the OOV issue.

The rule-based methods, e.g., ChangeScribe [10], [56], are used to summarize code changes between two adjacent software versions which can seem as an abstraction of different versions rather than specific code changes. Hence, we filter out these methods and compare with all other existed commit message generation approaches.

4.3 Experimental Results

4.3.1 *What is the performance of ATOM comparing with baseline models?*

Table 1 shows the compared results where $\text{NMT}_{(\text{Jiang et al.})}$ and $\text{NMT}_{(\text{Loyola et al.})}$ refer to the models proposed by Jiang et al. [8] and Loyola et al. [52] respectively. We can find that the $\text{NMT}_{(\text{Loyola et al.})}$ achieves the best performance among all the generation-based models, i.e., except for NNGen. However, the best generation model is still lower than the retrieval-based method NNGen, which demonstrates the effectiveness of the retrieval-based method on message generation tasks. Furthermore, ATOM outperforms all the baseline models in terms of all the metrics. BLEU-4, ROUGE-L and Meteor increases by 30.72%, 44.89%, 35.26%, respectively. This can be attributed to that ATOM can effectively integrate the advantages of generation and retrieval modules. The improved models claiming to capture code semantic patterns such as Ptr-Net and CODISUM present lower performances and the copy mechanism used in Ptr-Net does not improve the prediction accuracy. This may be attributed to the fact that in the code summary scenario, massive identifier names may appear a few times. However, with copy mechanism, these rare identifier names are also considered from source code, which increases the chance of choosing inappropriate tokens, leading to inferior results.

Answer to RQ1: In summary, ATOM outperforms the baseline models and the combination of generation and retrieval modules help model to get higher scores of 10.51, 24.33, 9.55, 22.02, and 18.51 in terms of BLEU-4, ROUGE(1,2,L) and Meteor.

4.3.2 *What is the impact of different modules on the performance of ATOM?*

We also perform experiments to evaluate the impact of respective generation and retrieval modules on the generated commit messages, with results shown in Table 1. We denote the results produced by the retrieval module only as ATOM $_{\text{Ret}}$, and generation module only as ATOM $_{\text{Gen}}$. We find that the performance of retrieval module ATOM $_{\text{Ret}}$ is slightly higher than ATOM $_{\text{Gen}}$, but the overall performance is still lower than the combined model. The gains achieved by our hybrid ranking module range from 12.76% to 56.58% in terms of BLEU-4, ROUGE and Meteor. Hence, incroporating the retrieval results into the generated results will further boost the performance of generation module. In addition, our well-designed generation module *AST2seq* achieves the best performance among all the generation models and BLEU-4 is far superior to the state-of-the-art model CODISUM [41]. So encoding AST to learn code semantics is more effective than sequence models. Finally, ATOM $_{\text{Ret}}$ has a slightly better performance than NNGen on our dataset, as we retrieve the most similar commit message based on tf-idf score. However, NNGen searches top-k training `diffs` only based on term frequency, so some key tokens with low frequency will be ignored.

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 4. In a total of 14,674 testing samples, 8,168 of the results are from the retrieval module, accounting for 55.66% of the whole testing corpus and the other 6,506 are from the generation module (44.34%). Based on the statistics, we can achieve that both retrieval and generation modules are helpful for accurate commit message generation, and they are complement to each other.

Answer to RQ2: In summary, incorporating the retrieval results into generation module can boost the final perfor-

TABLE 1: Comparison results with baseline models and different modules within ATOM

Model		BLEU-4	ROUGE-1			ROUGE-2			ROUGE-L			Meteor
			Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	
Baselines	$NMT_{(Loyola\ et\ al.)}$	5.23	16.35	14.35	14.29	5.02	4.88	4.75	15.73	13.82	12.73	10.37
	$NMT_{(Jiang\ et\ al.)}$	4.81	14.90	13.72	13.40	4.70	4.64	4.48	14.28	13.17	11.95	9.87
	NNGen	8.04	17.79	17.63	16.76	8.18	8.19	7.95	17.11	16.96	15.20	13.68
	Ptr-Net	0.45	13.95	8.11	9.27	1.04	0.67	0.72	13.54	7.89	7.61	4.98
	CODISUM	1.75	16.83	9.77	11.61	2.62	1.78	2.00	16.56	9.61	9.87	8.35
Ours	$ATOM_{Gen}$	7.35	20.32	15.92	16.69	7.14	6.02	6.23	19.58	15.37	14.80	11.82
	$ATOM_{Ret}$	8.52	17.58	17.48	17.56	8.69	8.71	8.46	17.84	17.74	15.93	14.35
	ATOM	**10.51**	**26.11**	**25.45**	**24.33**	**10.00**	**9.91**	**9.55**	**25.22**	**24.61**	**22.02**	**18.51**

TABLE 2: The performance of path set on *AST2seq*. The left column represents the path number for *Added* and *Deleted* paths separately.

# Path	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Meteor
30	4.27	15.55	4.50	13.66	10.22
50	5.96	15.41	5.26	13.92	10.89
100	7.09	16.40	5.98	14.34	11.42
200	6.07	15.47	5.30	14.01	11.01
300	6.04	15.38	5.27	13.95	10.99
80	**7.35**	**16.69**	**6.23**	**14.80**	**11.82**

mance, and the gains range from 12.76% to 56.58% in terms of the validation metrics. Besides, among all the generation baselines, our proposed *AST2seq* can produce more accurate commit messages.

4.3.3 *How accurate is AST2seq under a different number of paths?*

ATOM encodes AST paths based on `diffs` extracted from completed functions to represent code changes, however the `diff` lengths vary depending on the functions in the commit. *AST2seq* sets the max number of paths of each code snippet to 80 in the *Added* and *Deleted* ASTs during training. Fig. 8 shows that nearly 80% of commits have fewer than 80 AST paths in our dataset. We further conduct experiments with different path numbers set, and the results are shown in Table 2. 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 that fewer paths have limited capability in representing code changes. Increasing paths to over 100 does not result in continuously improved performance and the scores show slight decrease when the path numbers are augmented 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 best number 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?*

We design a ConvNet model to incorporate the output of retrieval module into generation module and compare the relevance between candidate messages and `diffs`. The prediction process can be regarded as a regression problem, and can be solved using other machine learning methods. In this section, we evaluate the performance of different methods for the ranking module. We choose XGBoost [57], Support Vector Regression (SVR) [58], GRU [24] and LSTM [21] with or without Attention Mechanism as the baselines for ranking. We compute tf-idf scores of the terms 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 which are then fed into a fully-connected layer for predicting the relevance score. The training loss functions are similar to the definition in CoveNet model. The comparison results are shown in Table 3. We can find that deep learning methods outperform machine learning methods for the relevance prediction. Specifically, XGBoost presents a better performance compared with SVR as it combines a set of classification and regression trees (CART) [59], which can gradually reduce prediction errors by each iteration. The performance of sequential deep learning approaches is inferior to ConNect, which indicates that ConvNet can effectively learn representations of messages and `diffs` and better predict their relevance.

TABLE 3: The performance of different ranking methods

Model	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Meteor
XGBoost	8.93	17.17	7.78	15.48	13.82
SVR	8.73	16.88	7.61	15.22	13.46
GRU	9.34	17.37	8.17	15.72	14.10
LSTM	9.32	17.51	8.22	15.85	14.18
LSTM+Att	9.33	17.49	8.18	15.82	14.16
ConvNet	**10.51**	**24.33**	**9.55**	**22.02**	**18.51**

TABLE 4: Percentage of final results prioritized from retrieved and generated messages.

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

Answer to RQ4: For predicting relevance between messages and `diffs`, ConvNet is superior to traditional machine learning models, e.g., XGBoost and SVR, and sequential deep learning models, e.g., GRU and LSTM. It can well prioritize the message candidates based on the relevance to the `diffs` and thus boost the model performance.

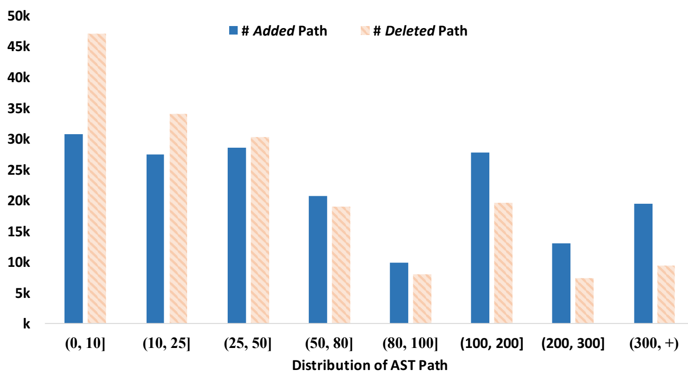


Fig. 8: The distribution of AST paths shown in the dataset. Each bar represents the number of commits that has the number of AST paths in a specific interval. For example, the leftmost blue bar represents almost 30,000 commits in our dataset have less than 10 *Added* AST paths by our preprocess.

5 Human Study

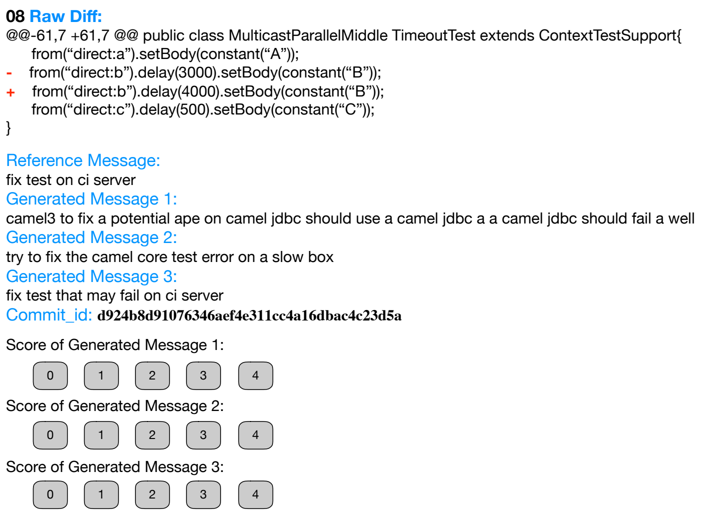
We conduct a human evaluation to evaluate ATOM and compare ATOM with the best retrieval model NNGen [7] and the best generation model NMT [8] in terms of BLEU-4 score. We invite 4 Ph.D students and 2 master students from the department of computer science to participate in our survey. None of participants are co-authors of this paper and they all have software development experience in Java programming language (raging 1 $\sim$ 5 years).

5.1 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, NMT, 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.

5.2 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, 1 means two messages have no or some shared tokens but with on 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


08 Raw Diff:

```
@@-61,7 +61,7 @@ public class MulticastParallelMiddle TimeoutTest 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"));
}
```

Reference Message:
fix test on ci server
Generated Message 1:
camel3 to fix a potential ape on camel jdbc should use a camel jdbc a a camel jdbc should fail a well
Generated Message 2:
try to fix the camel core test error on a slow box
Generated Message 3:
fix test that may fail on ci server
Commit_id: d924b8d91076346aef4e311cc4a16dbac4c23d5a

Score of Generated Message 1:
0 1 2 3 4
Score of Generated Message 2:
0 1 2 3 4
Score of Generated Message 3:
0 1 2 3 4

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.

TABLE 5: Results of human evaluation where "Low", "Medium", and "High" refer to low-quality, medium-quality, and high-quality messages respectively.

Model	Low	Medium	High	Mean Score
NNGen	49%	27.58%	23.42%	1.59
NMT	58.1%	25.86%	16.04%	1.29
ATOM	**44.35%**	**29.06%**	**26.59%**	**1.68**

contains corresponding scores for the messages generated by NMT, NNGen, and ATOM, respectively. We treat a score of 0 and 1 as low quality, a score of 2 as medium quality, and a score of 3 and 4 as high quality and we calculate the mean score as the final score. Table 5 shows the results of our human evaluation. The mean score of ATOM is 1.68, which is slightly higher than NNGen, but is much higher than NMT. The percentage of low quality messages generated by ATOM is the lowest 44.35%. Meanwhile, ATOM can generate more high quality messages which occupied 26.59%. The performance of NNGen is better than NMT justified the effectiveness of NNGen [7].

Overall, our results show that besides automatic evaluation metrics e.g. BLEU-4, ROUGE, Meteor, ATOM can also have a higher performance on human evaluation.

6 Discussion

In this section, we discuss the strengths of *AST2seq* compared with NMT approaches, then provide more details about our dataset compared with Jiang's [8] , and finally discuss the limitations of ATOM.

6.1 Strengths of *AST2seq*

Previous studies, e.g., NMT [8], [52], Ptr-Net [51] treat `diff` as a flat sequence of tokens, which ignores code syntax information. To address this limitation, CODISUN [41] extracted code structure and code semantics based on identifying all the class/method/variable names and segmenting with the corresponding placeholders. In this way, they achieved BLEU-4 of 2.19 on Jiang's [8] dataset. Although they get the highest BLEU-4 over NMT methods, the 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. 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 with structure information can seem as a refinement of original source codes and the recurring patterns that suggest semantics 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 models [8], [41], [51], [52], they need to set maximum sequence, e.g., 100 tokens in total [8] for effective learning, which will filter out a commit with too many chunks. Hence the sequence-based models cannot translate a commit with long sequences. However *AST2seq* can effectively addresses 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 and will be truncated when the actual paths exceed this threshold. By this way, *AST2seq* is able to decode longer `diffs` and the results about the performance among different `diff` length are shown in Table 6, where the left column is the `diff` lines rather than token length. The BLEU-4 of over 2.9k samples within 10 lines `diffs` is 11.37 and it takes up 19.9% in the whole testset. When diff lines increase to 100+, the performance only decreases by 2.00, 2.26, 2.12, 2.21, and 2.64 in terms of BLEU-4, ROUGE-1, ROUGE-2, 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 and *AST2seq* uses ASTs to encode code changes addressing the limitation of sequence length.

To sum up, *AST2seq* takes advantage of code structure into the encoder and has the ability to handle longer `diffs`, which is superior to the existing work.

6.2 Our Dataset

We crawl our dataset from 56 popular java projects ranked by star numbers. We have devoted substantial efforts to clean the dataset and further 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 completed functions based on the file paths and modified line numbers in the `diffs`. 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 we get and num is the number of extracted functions.
- Finally, the noisy commits, e.g., bot messages, which refers to messages generated by development tools and trivial messages with little information in Liu et al. [7] have been filtered automatically, as we keep commits modified in *.java* files and these boot messages and trivial messages most exist in configuration files, e.g., **.md*, **.gitrepo*. Hence, we have higher quality data to describe code changes compared with Jiang's [8].

The dataset not only contains basic commit information, but also contains the completed functions altered by commits. Hence with this dataset, we can boost some other researches, e.g., automatic code summarization, code recommendation, knowledge graph construction based on commits.

The dataset we prepared contains adequate information compared with Jiang's [8], and we make it publicly at https://drive.google.com/commit_data to benefit community research.

6.3 Limitations

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 effort 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 retrieval module is incorporated into 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 dataset, we still need to spend time and effort to do the preprocessing and model tuning. This can seem 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 by training data, the generation process is relatively low-cost and convenient.

6.4 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

TABLE 6: Results of `diffs` with different lines rather than tokens. For example the upper left **1-10** in the `diff` Lines column represents the commits with at most 10 lines of `diffs`.

`diff` lines	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Meteor	Number	Ratio
1-10	**11.37**	**25.88**	**10.84**	**23.48**	**20.41**	2920	19.90%
10-30	9.93	23.00	9.01	20.77	17.71	3435	23.41%
30-50	9.33	24.15	9.29	21.83	18.63	2652	18.07%
50-100	10.86	24.94	9.97	22.56	19.19	**3670**	**25.01**%
100+	9.37	23.62	8.72	21.27	17.77	1996	13.60%

indicates the effectiveness of our proposed approach. Another threat to validity is about the human evaluation in Section 5. 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. Idealy, 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]. Finally, we only compare ATOM on our dataset with the baseline methods and get the 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 dataset to illustrate the effectiveness of ATOM.

7 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.

7.1 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], [56], [60] 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. [60] 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], [56] 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], [41], [52] 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, [8] directly adopt NMT model to conduct the translation. [61] incorporates the context of code changes into the NMT model. [41] 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.

7.2 Code Summarization

Code summarization aims to generate brief natural language descriptions for code snippet and it evolves from rule-based [62] [63] [64], retrieval-based [65] [66] [67] to learning-based [68] [69] [70] 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. [63] 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. [64] predifine rules to combine information to generate comments for Java classes.

Information retrieval approaches are widely used in summary generation tasks. Haiduc et al. [65] 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. [71] 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. [72] apply topic modeling 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 [67] 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. [69] propose CODE-NN, an attentional LSTM encoder-decoder network to generate C# and SQL descriptions. Hu et al. [73] 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. [70] 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 [15] 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.

8 Conclusion

Automatically generating commit messages is necessitated. Existing studies either translate `diffs` with sequence-based models 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. Substantial experiments based on our dataset have demonstrated the effectiveness of ATOM and ATOM outperforms the state-of-the-art approaches with 10.51, 24.33, 9.55, 22.02 and 18.51 in terms of BLEU-4, ROUGE(1,2,L) and Meteor. 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.

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 *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 422–431.

[4] Sourceforge. [Online]. Available: https://sourceforge.net/

[5] "junit5," https://github.com/junit-team/junit5.

[6] "neo4j," 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 373–384.

[8] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 135–146.

[9] R. P. L. Buse and W. Weimer, "Automatically documenting program changes," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 33–42.

[10] L. F. Cortes-Coy, M. L. Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014*, 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 *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, 2017, pp. 414–423.

[12] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*. ACM, 2018, pp. 200–210.

[13] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.

[14] Javaparser documentation. [Online]. Available: https://javaparser.org/

[15] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[16] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[17] ——, "A general path-based representation for predicting program properties," in *ACM SIGPLAN Notices*, vol. 53, no. 4. ACM, 2018, pp. 404–419.

[18] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[19] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

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

[21] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

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

[23] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[24] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[26] O. Levy, M. Seo, E. Choi, and L. Zettlemoyer, "Zero-shot relation extraction via reading comprehension," *arXiv preprint arXiv:1706.04115*, 2017.

[27] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple object recognition with visual attention," *arXiv preprint arXiv:1412.7755*, 2014.

[28] B. Hu, Z. Lu, H. Li, and Q. Chen, "Convolutional neural network architectures for matching natural language sentences," in *Advances in neural information processing systems*, 2014, pp. 2042–2050.

[29] L. Pang, Y. Lan, J. Guo, J. Xu, S. Wan, and X. Cheng, "Text matching as image recognition," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[30] pygments documentation. [Online]. Available: http://pygments.org/

[31] P. Niemeyer and J. Knudsen, *Learning Java*. " O'Reilly Media, Inc.", 2005.

[32] Exuberant ctags documentation. [Online]. Available: http://ctags.sourceforge.net/

[33] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 320–323.

[34] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[35] Natural language toolkit nltk documentation. [Online]. Available: https://www.nltk.org/

[36] Scikit-learn. [Online]. Available: https://scikit-learn.org/stable/

[37] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.
[38] L. Yang, J. Hu, M. Qiu, C. Qu, J. Gao, W. B. Croft, X. Liu, Y. Shen, and J. Liu, "A hybrid retrieval-generation neural conversation model," *arXiv preprint arXiv:1904.09068*, 2019.
[39] W. James and C. Stein, "Estimation with quadratic loss," in *Breakthroughs in statistics*. Springer, 1992, pp. 443–460.
[40] 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." 2018.
[41] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 2019, pp. 3975–3981.
[42] "structs," https://github.com/fatih/structs.
[43] "antlr4," https://github.com/antlr/antlr4.
[44] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, 2016, pp. 1050–1059.
[45] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 1171–1179.
[46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[47] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.
[48] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.
[49] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.
[50] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, 2017, pp. 1073–1083.
[51] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, "Generating commit messages from diffs using pointer-generator network," in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 299–309.
[52] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, 2017, pp. 287–292.
[53] S. van Hal, M. Post, and K. Wendel, "Generating commit messages from git diffs," *arXiv preprint arXiv:1911.11690*, 2019.
[54] R. Sennrich, O. Firat, K. Cho, A. Birch, B. Haddow, J. Hitschler, M. Junczys-Dowmunt, S. Läubli, A. V. M. Barone, J. Mokry *et al.*, "Nematus: a toolkit for neural machine translation," *arXiv preprint arXiv:1703.04357*, 2017.
[55] D. M. Christopher, R. Prabhakar, and S. Hinrich, "Introduction to information retrieval," *An Introduction To Information Retrieval*, vol. 151, no. 177, p. 5, 2008.
[56] M. L. Vásquez, L. F. Cortes-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, 2015, pp. 709–712.
[57] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.
[58] H. Drucker, C. J. Burges, L. Kaufman, A. J. Smola, and V. Vapnik, "Support vector regression machines," in *Advances in neural information processing systems*, 1997, pp. 155–161.
[59] R. J. Lewis, "An introduction to classification and regression tree (cart) analysis," in *Annual meeting of the society for academic emergency medicine in San Francisco, California*, vol. 14, 2000.
[60] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016*, 2016, pp. 103–112.
[61] P. Loyola, E. Marrese-Taylor, J. A. Balazs, Y. Matsuo, and F. Satoh, "Content aware source code change description generation," in *Proceedings of the 11th International Conference on Natural Language Generation, Tilburg University, The Netherlands, November 5-8, 2018*, 2018, pp. 119–128.
[62] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, "Using stereotypes in the automatic generation of natural language summaries for c++ methods," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 561–565.
[63] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.
[64] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 23–32.
[65] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 223–226.
[66] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*. IEEE, 2010, pp. 35–44.
[67] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 380–389.
[68] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning*, 2016, pp. 2091–2100.
[69] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.
[70] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 397–407.
[71] 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 *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568247
[72] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in *Proceedings of the 22nd international conference on program comprehension*. ACM, 2014, pp. 291–294.
[73] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2269–2275. [Online]. Available: https://doi.org/10.24963/ijcai.2018/314