Generating summaries for methods of event-driven programs: An Android case study

Alireza Aghamohammadi[1], Maliheh Izadi[1], Abbas Heydarnoori *

Department of Computer Engineering, Sharif University of Technology, Iran

ARTICLE INFO

ABSTRACT

The lack of proper documentation makes program comprehension a cumbersome process for developers. Source code summarization is one of the existing solutions to this problem. Many approaches have been proposed to summarize source code in recent years. A prevalent weakness of these solutions is that they do not pay much attention to interactions among elements of software. An element is simply a callable code snippet such as a method or even a clickable button. As a result, these approaches cannot be applied to event-driven programs, such as Android applications, because they have specific features such as numerous interactions between their elements. To tackle this problem, we propose a novel approach based on deep neural networks and dynamic call graphs to generate summaries for methods of event-driven programs. First, we collect a set of comment/code pairs from Github and train a deep neural network on the set. Afterward, by exploiting a dynamic call graph, the Pagerank algorithm, and the pre-trained deep neural network, we generate summaries. An empirical evaluation with 14 real-world Android applications and 42 participants indicates 32.3% BLEU4 which is a definite improvement compared to the existing state-of-the-art techniques. We also assessed the informativeness and naturalness of our generated summaries from developers' perspectives and showed they are sufficiently understandable and informative.

1. Introduction

During the software development life cycle, various reports and documentation such as requirements specification, architecture documents, design documents, bug reports and so forth need to be generated. Unfortunately, lack of direct motivation in software teams results in inadequate, out-of-date and unqualified documentation. This in return makes program comprehension a difficult and time-consuming task for other team members. Xia et al. (2018) attest to this fact by claiming that on average developers spend about 58% of their time understanding a program (Xia et al., 2018). Source code summarization aids developers in understanding how a program works better and faster. This technique is used for describing the goal or functionality of different parts of a software program, namely methods, classes, or packages in a comment (Badihi and Heydarnoori, 2017). Fig. 1 depicts a code snippet with its accompanied comment.

Many approaches for source code summarization have been proposed over the course of past years. For instance, exploiting knowledge of the crowd (Badihi and Heydarnoori, 2017; Nazar et al., 2016; Guerrouj et al., 2015; Rahman et al., 2015; Wong et al., 2015), information retrieval (Sridhara et al., 2010; Antoniol et al., 2002; Moreno et al., 2013a), machine learning (McBurney et al., 2014; Haiduc et al., 2010b,a; Fowkes et al., 2017; Eddy et al., 2013), neural networks (Allamanis et al., 2016; Iyer et al., 2016; Hu et al., 2018), or even tracking eye-movements of developers (Rodeghero et al., 2014) are among the approaches for addressing this issue. The main limitation of these approaches is that they do not pay much attention to interactions among elements of a software while generating summaries. McBurney and McMillan (McBurney and McMillan, 2016) regarded the context of a program as a critical factor. They defined a method based on its invocations. However, they did not consider events in a software program. Event-driven programs contain a cycle which waits for events. When an event triggers, the program runs it. Therefore, interactions among elements are specified at run-time in these programs. An Android application is an excellent example of an event-driven program. As a result, these approaches cannot be applied to event-driven programs because of characteristics of those programs such as numerous interactions between their elements. Fig. 2 demonstrates the life cycle of Android applications.

Consider a program presenting one `Button` and one `TextBox` in the main display page. A user types an arbitrary text and pushes the `Button` which leads to another page. The new page

* Corresponding author.
E-mail addresses: aaghamohammadi@ce.sharif.edu (A. Aghamohammadi), maliheh.izadi@sharif.edu (M. Izadi), heydarnoori@sharif.edu (A. Heydarnoori).
[1] The first two authors contributed equally to this work.

```
1  // Creates an intent, adds location data to it
2  // as an extra, and starts the intent service
3  // for fetching address.
4  private void startIntentService() {
5      Intent intent = new Intent(this, FetchAddressIntentService.class);
6      intent.putExtra(Constants.RECEIVER, mResultReceiver);
7      intent.putExtra(Constants.LOCATION_DATA_EXTRA, mLastLocation);
8      startService(intent);}
```

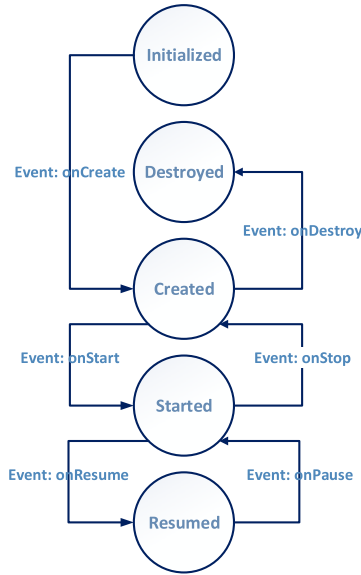**Fig. 1.** An example code snippet with its summary (Anon, 2018a).

Fig. 2. An Android application's life cycle.

shows the user's text. Fig. 3 is a source code that demonstrates the behavior of this program. When a user pushes the `Button`, the functions `sendMessage()` from the `MainActivity` class and `onCreate()` from the `DisplayMessageActivity` class are invoked, respectively. Unlike how trivial it may seem, this is not an easy task for a Java code. Indeed, when a user pushes the `Button`, the Android framework calls the `onClick()` API related to that `Button`. In other words, the developer must set the `onClick` attribute to `sendMessage` for the `Button` in `res/layout/activity_main.xml`. Therefore, finding relations between elements statically is a cumbersome task.

In this paper, we try to solve the problem of generating summaries for methods of event-driven programs by extracting the interactions between their elements at run-time. To this end, we used a deep neural network to generate summaries. Additionally, to capture interactions at run-time, we utilized dynamic call graphs.

The main contributions of this work are:

1. We propose an approach to generate summaries for methods of event-driven programs. The proposed approach exploits deep neural networks and dynamic call graphs as the key components of the solution to produce meaningful summaries which not only address the semantics of the source code but also have a well-formed grammar.
2. Unlike existing work, we introduce a novel technique for generating summaries that concentrates on run-time execution.

The rest of the paper is organized as follows. In Section 2, we review related work. We provide background and preliminary information in Section 3. In Section 4, we describe our proposed approach. In Section 5, we evaluate the proposed approach by answering seven research questions. We assessed our deep neural network model using different evaluation metrics. Furthermore, we set up a user study to evaluate the generated summaries on real-world Android applications. We conducted an experiment with 14 real-world Android applications and 42 participants to measure the quality of our approach. The experimental results show 32.3% BLEU4 and 11.2% METEOR. Next, Section 6 presents threats to the validity of our results. Finally, we conclude this paper and present potential future work in Section 7.

2. Related work

In this section, we review three previous types of approaches to code summarization, namely information retrieval, machine learning, and crowdsourcing. In recent years, neural networks have been used as a new path to source code summarization. As for evaluation, BLEU4 and METEOR has been recently favored over precision and recall measures. The Java programming language is the most popular language used in source code summarization techniques. We present an overview of these approaches in the following.

2.1. Code summarization via information retrieval

Sridhara et al. (2010), proposed an algorithm for automatic description of Java methods. They preprocessed Java methods using the Software Word Usage Model (SWUM). SWUM is a technique for displaying methods of a program in the form of noun, verb, and adverb groups. McBurney and McMillan (2016), introduced an approach for the automatic generation of documents for Java methods based on the context. In summary, this approach uses PageRank (Page et al., 1999) to find the most important methods for the given context. SWUM helps determine what these most important methods do. Finally, natural language generation system generates a human-readable summary. Rodeghero et al. (2014) proposed a method for choosing essential words of a code segment. They analyzed developers' eye-movements and their focused attention while writing summaries for a method, and then used their findings to weight the words subsequently. Antoniol et al. (2002) proposed an approach for improving trace-ability links between a code segment and its document. They utilized the unigram language model and Vector Space Model (VSM). Moreno et al. (2013a,b) presented an approach for summarizing Java classes. They primarily focused on each class content and tasks but did not heed the connection between classes. They first found a class's stereotypes and its methods. Then classified stereotypes into 13 groups. Afterward, using natural language rules, they generated a summary for each class based on a specific format.

```java
public class MainActivity extends AppCompactActivity {
    public static final String EXTRA_MESSAGE = "MESSAGE";
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);}
    public void sendMessage(View view) {
        Intent intent = new Intent(this, DisplayMessageActivity.class);
        EditText editText = (EditText) findViewById(R.id.editText);
        String message = editText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);}}
```

```java
public class DisplayMessageActivity extends Activity {
    protected void onCreate(savedInstanceState) {
        setContentView(R.layout.activity_display_message);
        Intent intent = getIntent();
        String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
        TextView textView = findViewById(R.id.textView);
        textView.setText(message);}}
```

Fig. 3. Source code for the running example (Anon, 2017): the `sendMessage()` method is called whenever a user clicks the button.

2.2. Code summarization via machine learning

McBurney et al. (2014), presented a code summarization method using hierarchical topic modeling. The most abstract description of program's tasks is given in the highest level of the hierarchy in a Hierarchical Document Topic Model (HDTM) algorithm. As one goes down the hierarchy, descriptions become more precise and clear. Authors first formed the call graph with methods as nodes and caller–callee among methods as edges of the graph. Then, HDTM was performed on the graph. Haiduc et al. (2010b), considered code as text and exploited previous text summarization methods for summarizing code snippets as well. They used the VSM and Latent Semantic Indexing (LSI) (K.L.andauer et al., 1998) in their work. Eddy et al. (2013), proposed a code summarization algorithm using hierarchical topic modeling. In fact, this study is a replica of the Haiduc et al. work (Haiduc et al., 2010b), with the distinction that they utilized HPAM instead of VSM and LSI. Programming tools help developers hide or reveal some parts of their code. This feature is known as code folding. Fowkes et al. (2017), introduced an approach for code summarization using code folding and an Abstract Syntax Trees (ASTs).

Recently, researchers have been using neural networks as a method for generating summaries. Iyer et al. (2016), proposed CODE-NN, a neural attention model for summarizing source code. They used LSTM networks for generating descriptions of C# code snippets and SQL queries. Allamanis et al. (2016), introduced a novel attention mechanism using Convolutional Neural Networks (CNN) (Collobert and Weston, 2008). Their goal was to generate the name of a method from its code. Vaswani et al. (2017), were the first to introduce the Transformer. Their model exploits the multi-head attention mechanism while removing recurrence and convolutions from the network. The Transformer is parallelizable and requires significantly less training time. Hu et al. (2018), produced descriptions for Java methods using a sequence-to-sequence model. To improve performance, they exploited the structured form of code and introduced a novel method named SBT to parse ASTs. Allamanis et al. (2018), represented programs with graphs to exploit the syntactic and semantic structure of source code using Gated Graph Neural Networks (GGNN). They evaluated their models based on two tasks of predicting variable names (VarNaming) and predicting variable misuse (VarMisuse). Wan et al. (2018) exploited ASTs and sequential data of code snippets in a reinforcement learning framework. The next word is predicted using the actor network and the critical component of the network evaluates the reward value. CODE2SEQ, proposed by Alon et al. (2019), uses syntactic structure of source code and represents each code snippet as a set of paths in an AST. Using attention mechanism of a sequence-to-sequence model, it selects relevant paths in the decoder. LeClair et al. (2019) also used a sequence-based model, but they treated each input data source separately. This helped the model learn the structure of code independent of other textual information in the code snippet. Haque et al. (2020) claim the information inside a subroutine is not sufficient for summarizing it. Therefore, they proposed a sequence-based model using the attention mechanism to predict the context of subroutines.

2.3. Code summarization via CrowdSourcing

Badihi and Heydarnoori (2017), proposed a code summarization model for Java language using the power of crowdsourcing. They built a web-based system for developers and encouraged them to write summaries for various methods using gamification techniques. Then they collected these summaries and analyzed them to identify the most significant parts of methods from developers' point of view. Nazar et al. (2016), presented a code-by-code summarization approach using crowd source knowledge and supervised learning. First, they extracted code snippets from the most frequently asked questions (FAQ) section of Integrated Development Environment (IDE). Then they used four developers for labeling these code segments. They extracted 21 features. Then, they utilized Support Vector Machine (SVM) and Naïve Bayes algorithms to classify results, and finally generated summaries using these two supervised learning algorithms. Guerrouj et al. (2015), used the context available in posts of Stack Overflow

Q&A website in order to generate code summaries. Using an Island parser, they extracted identifiers from discussions about an element. Rahman et al. (2015), proposed an approach to generate summaries to recommend to developers through analyzing discussions and comments of users on Stack Overflow posts. Wong et al. (2015), introduced a method for automatic documentation of codes using Github and clone detection techniques.

As reviewed above, there are many approaches to code summarization. However, they have their limitations. One major defect of existing solutions is that to the best of our knowledge they do not consider dynamic interactions among elements of a software program. As interactions are triggered at run-time, they cannot be inferred statically. Therefore, to exploit this information to generate better summaries for code snippets, one needs to investigate these codes at run-time. In this work, we utilize these valuable interactions to generate more useful summaries.

Another frequent shortcoming of the existing approaches is related to their evaluations. Most of the current models are evaluated using precision and recall metrics. As shown in Section 5, these metrics lack the validity for evaluating machine translations tasks. That is why we have used BLEU and METEOR to better evaluate the performance of our proposed model.

Moreover, most of these approaches are template-based, that is they generate summaries based on predefined rules. Therefore, these summaries neglect the essential semantics of a task/code, which renders them not very useful for end users in real-world cases. In this work, we have used deep learning methods to overcome this issue and generate more meaningful summaries.

3. Background and terminology

In this section, we provide preliminary information on the notation and methods we have used in our proposed approach. Recently, researchers has turned to applying deep learning methods to various fields of software engineering such as commit message generation (Jiang et al., 2017; Liu et al., 2018), intention mining (Huang et al., 2018), and code search (Gu et al., 2018). Among these fields, is code summarization via deep learning (Hu et al., 2018), which has attained promising results so far. The deep neural network is used as a pre-built model to generate final summaries. The notation of our deep model is as follows[2]:

- x: set of source codes written in Java programming languages.
- $x^{(i)}$: ith source code in the set of x.
- $x_t^{(i)}$: tth token in the above sequence.
- T_s: T is the length of the sequence s.
- y: set of comments written in natural language.
- $y^{(i)}$: ith comment in the set of y.
- $y_t^{(i)}$: tth term in the above sequence.

3.1. Sequence-to-sequence model

Recurrent Neural Network (RNN) is suitable for sequences of inputs (Werbos, 1990). RNN generates sequence $y = (y_1, y_2, \ldots, y_{T_y})$ from input sequence $x = (x_1, x_2, \ldots, x_{T_x})$. Recently, the sequence-to-sequence model has yielded valuable results in the neural machine translation (Cho et al., 2014). In the traditional sequence-to-sequence model (Sutskever et al., 2014), the decoder uses the last hidden state of the encoder as an input for generating the output sequence. The decoder generates summaries in a deep neural network. We aim at finding the sequence $y =$

$(y_1, y_2, \ldots, y_{T_y})$ from the sequence $x = (x_1, x_2, \ldots, x_{T_x})$, given that it applies in Eq. (1):

$$\arg\max_y = \prod_{t=1}^{T_y} P(y_t | x, y_1, y_2, \ldots, y_{t-1}) \tag{1}$$

Suppose there are l layers, and $\overrightarrow{h}_t^{[l]}$ and $\overleftarrow{h}_t^{[l]}$ are the forward and backward states of the RNN for the term t. Therefore, $h_t^{[l]}$ is computed as $h_t^{[l]} = [\overrightarrow{h}_t^{[l]}, \overleftarrow{h}_t^{[l]}]$. We defined a context matrix denoted as C. C_i is the ith column of the context matrix and is called a context vector. C_i indicates how much attention the output term y_i pays to the terms of input sequence $x = (x_1, x_2, \ldots, x_{T_x})$. In other words, each of the members of $x = (x_1, x_2, \ldots, x_{T_x})$ to what extent contributes to generating the output y_i. States of the RNN of the decoder element are denoted as S_i. α_{ij} represents how much should y_i of the decoder element (the ith term of the summary) pay attention to $h_j^{[l]}$ of the decoder element (the jth term of the code).

$$C_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j^{[l]} \tag{2}$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{3}$$

$$e_{ij} = \tanh\left(W_j h_j^{[l]} + U_s S_{i-1}\right)$$

3.2. PageRank algorithm

The PageRank algorithm was developed by Page and Brin (Page et al., 1999) to sort webpages based on their popularity in Google's search engine. McBurney and McMillan (2016) used the same concept for measuring the importance of different methods.

Damping factor, denoted as d, indicates how likely is it for a specific node to be visited through time. It is conventional to set $d = 0.85$ (Brin and Page, 1998). The PageRank algorithm assigns a rank to each node. These ranks determine the importance of nodes in the corresponding graph. Ranks are calculated using Eq. (4) (Page et al., 1999) which r_i shows the rank of n_i:

$$r_i = (1 - d) + d \times \sum_{n_j \in B_i} \frac{r_j}{l_j} \tag{4}$$

In Eq. (4), l_j is the number of outgoing edges from n_j and B_i is the set of nodes which have outgoing edges to n_i.

4. Proposed approach

In this section, we present our approach to generate summaries for methods of event-driven programs. We consider the `sendMessage()` method described in Section 1 as a running example. This running example is used throughout this paper to show our process of generating summaries.

As shown in Fig. 4, our proposed approach consists of five steps. In the first step, we used a dataset of comment/code pairs from the Github repository. Then, applied a few preprocessing tasks on the data such as deleting blank lines, removing code snippets without summaries, and refining code-words based on the Java naming convention.

In the second step, we built a deep neural network for the comment/code pairs. This model was used to generate the code summaries. The architecture of our proposed deep neural network consists of three components, namely *encoder*, *decoder*, and *attention mechanism*. This kind of network is mainly used for sequential data in summarization and machine translation tasks. We include the encoder element for encoding source code and the decoder element for decoding the encoded output to summaries.
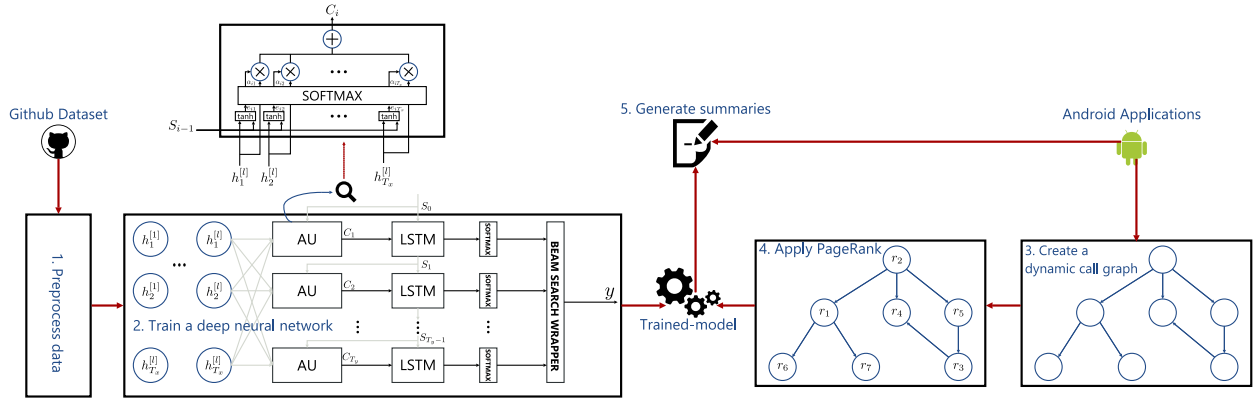
Fig. 4. An overview of the proposed approach for Android applications.

Moreover, we exploit the attention layer to put more emphasis on more important parts of the data as is frequently applied in the NLP field, specifically for machine translation and summarization tasks.

In the third step, we constructed a dynamic call graph of Android applications which were selected to generate summaries for. In the fourth step, the PageRank algorithm was applied to the graph mentioned above to sort the methods of an application and better understand the context of a given method.

We use PageRank to identify the most important information for generating comments. Statically finding relations among methods is not a trivial task. However, during run-time, each block (callable code snippets or methods) is identifiable through using dynamic call graphs. By analyzing the dynamic flow of event-driven programs, through using dynamic call graphs and then ranking them using the PageRank, we can capture long-range dependencies inside the source code and between code elements. Consequently, we can exploit the context and find the higher purpose behind a code snippet more accurately compared to when we just consider the method.

In the end, using our deep neural network of step two and outputs of the PageRank algorithm, we generated human-readable summaries for the selected methods of applications. In the following, we will elaborate more on the steps of our approach.

4.1. Step 1: Preprocess data

In this part, we elucidate the first step of our approach. First, we performed preprocessing steps on comment/code pairs extracted from Github. Hu et al. (2018), extracted more than 500 thousand comment/code pairs from Github and applied a few heuristic methods to extract 69708 pairs from this data. Although the 500 thousand pairs are available online, the preprocessed data are not accessible. As a result, we explored their raw data as a starting point. These source codes are written in Java, and Java programs follow specific naming conventions. The main preprocess steps used in this study are:

1. First, the blank lines (\n) and tabular characters (\t) were removed and replaced by space character.
2. Afterward, we identified and tokenized words with all capital letters that came before words that had capital first-letters. For instance, The following regular expression does the above task:

```
SQLDatabase --> SQL Database
Regular Expression: [A-Z]+(?=[A-Z][a-z])
```

3. Furthermore, words with capital first-letters or all lowercase letters are extracted as well. The corresponding regular expression comes as follows:

```
Regular Expression: [A-Z]?[a-z]+
```

4. Finally, we extracted words that all their letters were capital. We also kept special tokens (e.g., curly brackets and parentheses) in the final preprocessed data.

Fig. 6(a) demonstrates the output of our running example after the first step.

4.2. Step 2: Train a deep neural network

Our deep neural network tries to translate $x^{(i)} = \left(x_1^{(i)}, x_2^{(i)}, \ldots, x_{T_x}^{(i)}\right)$ to $y^{(i)} = \left(y_1^{(i)}, y_2^{(i)}, \ldots, y_{T_y}^{(i)}\right)$ for every i in a comment/code pair. In the following, we describe each component of the architecture in detail.

4.2.1. Encoder, decoder, and attention mechanism

There have been many pieces of research on the semantic representation of terms in a vector format with real numbers, namely Continuous Bag of Words (CBOW) (Mikolov et al., 2013a), SKIP-GRAM (Mikolov et al., 2013c; Goldberg and Levy, 2014; Mikolov et al., 2013), and Global Vectors (GLOVE) (Pennington et al., 2014). The benefit of this approach is that as much as the terms are semantically similar, their vectors are similar as well. Therefore, we used one embedding layer in the encoder and decoder components, for which the weights are tuned during the deep neural network learning phase. However, to reduce the learning time and to obtain more accurate weights, we used the pre-built model introduced in the GloVe website (Anon, 2018b).

Dropout is a simple solution to avoid overfitting (Srivastava et al., 2014). Dropout randomly omits neural network units. We used dropout = 0.2 in the neural network layers similar to Luong et al. research (Luong et al., 2015).

Vanishing gradient is a problem in simple RNNs (Hochreiter, 1998). It happens when a gradient is very small. This hinders changing values of weights and stops the neural network's training. To solve this issue, various methods such as Gated Recurrent Unit (GRU) (Chung et al., 2014) and Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) have been proposed. We applied the latter in this work similar to Luong et al. study (Luong et al., 2015).

Unidirectional RNNs use only past data. However, knowing about the future helps as well. Bidirectional RNNs (BRNN) process sequences on both directions and two different layers (Schuster

and Paliwal, 1997). Graves and Schmidhuber (2005), combined bidirectional recurrent neural networks with LSTM. Moreover, one can stack layers of neural networks to build a deep network (Graves et al., 2013). We have used a stack of BRNNs on top of the embedding layer in the encoder element.

When generating summaries in the decoder, one approach is to test all possible cases, which is definitely costly, with the computational complexity of $O(|V|^{T_y})$ ($|V|$ denotes the vocabularies' set size). Another approach is to use a greedy search algorithm. These algorithms select a term that maximizes the value of $P(y_t|x, y_1, y_2, \ldots, y_{t-1})$ in each step. However, if one utilizes a greedy search algorithm, she cannot change the term in the future. Furthermore, greedy search algorithms do not guarantee to produce good results since the co-occurrence probability of some terms is higher than others.

A better solution is to exploit the beam search algorithm (Graves, 2012). In the beam search algorithm, the $|B|$ top probabilities, are recorded partially for every step. $|B|$ denotes as the width of the beam. This heuristic algorithm does not necessarily optimize results; however, its computational complexity equals to $O(|B| \times |V|)$ which is immensely faster than computing all cases. It is worth mentioning that if $|B| = 1$, this heuristic algorithm acts like a greedy one. As $|B|$ increases, the quality of generated summaries improves, however, the learning time rises as well.

We have used the attention mechanism introduced by Bahdanau et al. (2014). The next step is to train a model on the preprocessed data, which will be used to generate summaries in the final step.

Implementation details for the deep neural network are discussed in the following. The operational environment for the deep neural network was the Ubuntu16.04. Our hardware included 40 processing cores and 64GB RAM. We used Tensorflow library to build the neural network model (Abadi et al., 2016) and Pandas library to preprocess the data (McKinney, 2011). Embedding layers included vectors with dimensions of 300. If the input word already exists in the pre-built model, its weights from the model are used. Otherwise, the corresponding vector to the input word is initialized with uniformly distributed real numbers between -1 and 1. The pre-built model contains about 2.2 million words.

The maximum length of summaries and codes are set to 35 and 100 tokens, respectively. In cases where the length of the input code is less than 100, remaining elements of the vector are replaced with zeros. Moreover, four words are pre-allocated namely (PADDING, 0), (UNK, 1), (SOS, 2), and (EOS, 3). UNK refers to an unknown word, which means the word does not exist in the deep neural network dictionary. Furthermore, SOS and EOS represent the start and end of each sentence, respectively.

In the learning process of deep neural networks, the goal is to minimize the loss function. We have used cross-entropy loss function in this study. The set of generated summaries is represented by $(y^{(1)}, y^{(2)}, \ldots, y^{(N)})$, in which N is the number of generated summaries and $y^{(i)}$ denotes as the ith generated summary. Therefore, the cross-entropy loss function can be calculated using Eq. (5):

$$\mathcal{L} = \sum_{i=1}^{N} \sum_{t=1}^{T_{y^{(i)}}} -\log P\left(y_t^{(i)}|y_1^{(i)}, y_2^{(i)}, \ldots, y_{t-1}^{(i)}\right) \tag{5}$$

Exploding gradient is one of the problems with long sequences. To prevent this, given that the gradient's size is more than a specific threshold such as $\tau = 5$, one should decrease its value using the approach introduced by Pascanu et al. (2013). In other words, its value should be updated using Eq. (6):

$$\widehat{g} \leftarrow \frac{\tau}{\|\widehat{g}\|}\widehat{g} \tag{6}$$

Adam is used for parameters' optimization (Kingma and Ba, 2014). As suggested by Kingma and Ba study (Kingma and Ba, 2014), we used $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ as default input values for the Adam algorithm.

Overfitting is another problem that may occur while applying machine learning techniques. It happens when a technique matches too closely with a specific set of data. Therefore, rendering the aforesaid technique unfits for predicting other data sets reliably. To avoid overfitting, we randomly split the deep model's inputs into three categories, namely train (80%), valid (10%), and test (10%) sets.

We generated checkpoints for every epochs during the training phase. Then, the best model for the validation set was selected (in terms of the BLEU score) to evaluate on the final test set.

4.3. Step 3: Create a dynamic call graph

Event-driven programs depend on the occurrence of events at run-time. Consider the running example illustrated in Section 1. The method `sendMessage()` is invoked every time a user clicks the `Button`. However, it is not a trivial task to find out why pushing the button is followed by running the `sendMessage()` method. In this step, we tackle this problem by leveraging the power of dynamic call graphs. Yuan et al. (2017), proposed an approach to generate a dynamic call graph for Android applications. Authors created a tool named Rundroid to create these graphs. This tool not only considers invocations between methods in static time but also recognizes messages transferred between an application and the Android framework at run-time. However, the Rundroid lacks automation, and users have to manually run and test programs to generate call graphs. Therefore, to automatize this task through generating random test with desired time intervals, we used a tool developed by Google, known as Monkey (Anon, 2018d). Fig. 5 depicts a part of the dynamic call graph generated for the running example.

4.4. Step 4: Apply PageRank

We applied the PageRank algorithm to the dynamic call graph generated in the previous step. Consider the dynamic call graph of Fig. 5. This graph consists of 13 nodes denoted as $V = \{n_1, n_2, n_3, \ldots, n_{13}\}$ and 12 edges. Table 1 shows normalized results of the PageRank algorithm applied to the running example.

4.5. Step 5: Generate summaries

In this step, final summaries are generated from the pre-trained model and ranks of nodes in the dynamic call graph produced in the second and fourth steps, respectively.

We first extract methods of the selected application. Suppose `sendMessage()` is one of these methods. First, we applied preprocessing tasks to the `sendMessage()` method. Fig. 6(a) illustrates the output of this step. Then, by using the pre-trained model from the second step, a summary was produced for the preprocessed running example (Fig. 6(b)). From the nodes in the dynamic call graph that have outgoing edges to the selected node (method), we selected the node with the highest rank. In case of
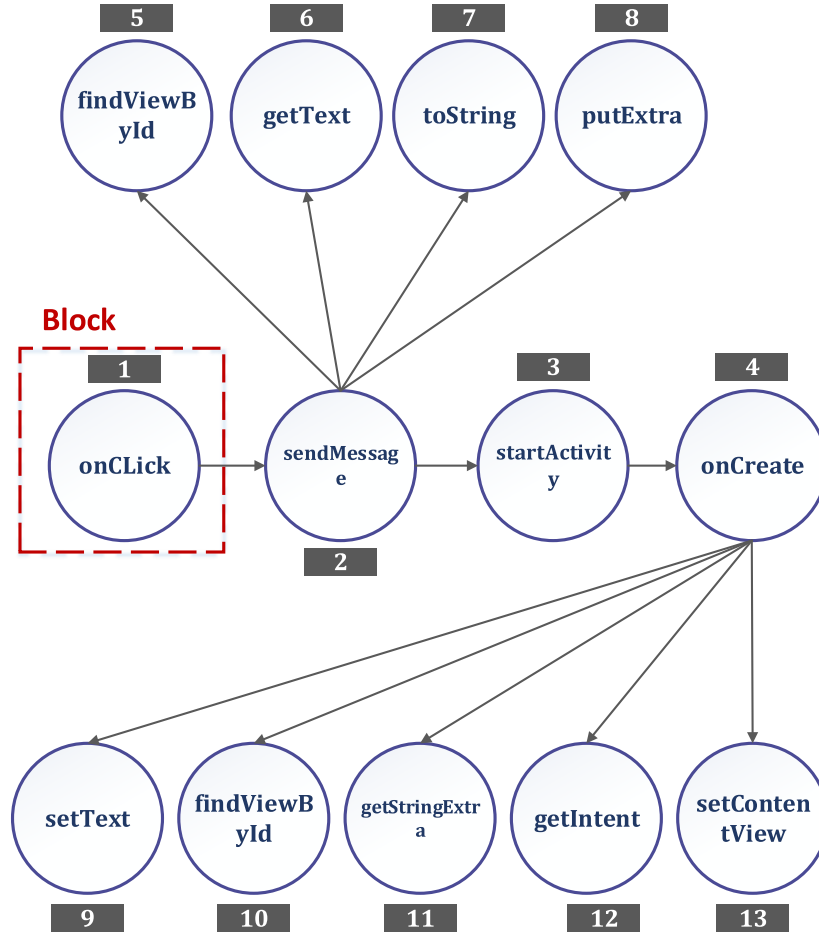
Fig. 5. Call graph of the running example.

Table 1
PageRank results for the running example.

l_i	B_i	r_i
$l_1 = 1$	$B_1 = \{\}$	0.0545
$l_2 = 4$	$B_2 = \{1\}$	0.1009
$l_3 = 1$	$B_3 = \{2\}$	0.0717
$l_4 = 5$	$B_4 = \{3\}$	0.1155
$l_5 = 0$	$B_5 = \{2\}$	0.0717
$l_6 = 0$	$B_6 = \{2\}$	0.0717
$l_7 = 0$	$B_7 = \{2\}$	0.0717
$l_8 = 0$	$B_8 = \{2\}$	0.0717
$l_9 = 0$	$B_9 = \{4\}$	0.0742
$l_{10} = 0$	$B_{10} = \{4\}$	0.0742
$l_{11} = 0$	$B_{11} = \{4\}$	0.0742
$l_{12} = 0$	$B_{12} = \{4\}$	0.0742
$l_{13} = 0$	$B_{13} = \{4\}$	0.0742

a tie, we randomly chose one of them. We call this node a *block*. If the block has a corresponding method in the source code of the program, we use that source code as an input for the pre-trained model. Otherwise, the block is related to the Android framework. In this case, we create a *dummy* method by adding a signature to the block. For instance, in Fig. 5, there is only one node called `onClick()`. Since the `onClick()` is related to the Android framework, we created a dummy method for the `onClick()` element and passed it to the pre-trained model (Fig. 6(c)). As an example, we add "*Public+void+onClick+(+View+view+)+{+}*" as a dummy method for `onClick()` After adding the output of the latter step, the summary for the given method was generated (Fig. 6(d)).

Note that we do not take the Android framework's implementation as the dummy method because of following reasons. First, all the implementations are not open-sourced. Second, implementations are not necessarily purely in Java. Third, it is possible to have multiple implementations for a given block in the Android framework, but finding the appropriate implementation can be very complicated and time-consuming. Therefore, we simply add a signature and build a dummy method. Signature addition is employed to change the format of inputs of our neural network to be like a method.

5. Evaluations

In this section, we present the results of both qualitative and quantitative evaluation of our proposed approach. First, the pre-trained deep neural network was assessed using BLEU4 and METEOR metrics. Then, using an empirical study, we examined the usefulness of our approach in aiding developers understand the objective of methods. This qualitative assessment was performed on 14 Android applications and 42 methods, using three highly skilled experts for generating reference summaries and 42 Android developers for evaluating generated summaries.

5.1. Evaluations of deep neural network

We first present Research Questions (RQs), evaluation metrics and the evaluation process. Afterward, we discuss results of our evaluations and analyze them subsequently.

```
public void send message ( view view )
{ ... start activity ( intent ) ; }
public void sendMessage(View view) {
   ...
   startActivity(intent);}
```

(a) Output of the first step on the running example

```
//Sends a message to the specified service
public void sendMessage(View view) {
   ...
   startActivity(intent);}
```

(b) Output of the second step on the running example

```
//Called whenever a view has been clicked
public void onClick(View view) {}
```

(c) Pass the highest node block as an input to the pre-trained model

```
//Sends a message to the specified service
//Called whenever a view has been clicked
public void sendMessage(View view) {
   ...
   startActivity(intent);}
```

(d) Generated summaries for the given method

Fig. 6. The outputs of applying different steps of the proposed approach on the running example.

5.1.1. Research questions

To evaluate our deep neural network, we answer the following four questions:

RQ1: How much the proposed model has been successful in learning comments/codes sets?

RQ2: What is the precision of the generated summaries by the proposed model?

RQ3: What proportion of reference summaries were retrieved as the final generated summaries?

RQ4: How well has the proposed deep neural network performed compared to the other baseline deep neural networks?

5.1.2. Evaluation metrics

Here, we investigate the evaluation metrics used in this study, namely BiLingual Evaluation Understudy (BLEU) and Metric for Evaluation of Translation with Explicit ORdering (METEOR).

BLEU is used for automated evaluation of machine translation algorithms (Papineni et al., 2002). Since code summarization is a type of translation of code snippets to human-readable summaries, BLEU can be used for evaluating abstractive code summarization. BLUE refines precision by valuing each term exactly for as many times as it has appeared in the reference translations. We compute the precision for different values of n in n-grams denoted as P_i in Eq. (7). For instance, P_1 considers every term separately. P_2 has a similar concept as P_1, except that it computes the precision of bi-grams. The final score is calculated using Eq. (7). In this equation, BP is a penalty for short summaries, which are identified using Eq. (8). In Eq. (8), r and c are the lengths of reference and generated summaries, respectively.

$$\text{BLEU} = BP \times \exp\left(\frac{\sum_{i=1}^{N} \log(P_i)}{N}\right) \tag{7}$$

$$BP = \begin{cases} 1 & c > r \\ \exp\left(1 - \frac{r}{c}\right) & c \leq r \end{cases} \tag{8}$$

METEOR was proposed to mitigate BLEU's shortcomings (Banerjee and Lavie, 2005). METEOR focuses mainly on recall, unlike BLEU which pays more attention to precision. METEOR

Table 2
Statistical Information of Extracted Pairs.

	Mean	Q1	Q2	Q3	# Unique tokens
Comment length	11.25	7	9	14	44934
Code length	33.49	15	26	47	22525

is based on the term-to-term mapping of the generated summary with its corresponding reference summary.

METEOR is calculated using Eq. (9). In Eq. (9), R is the refined recall, P is the refined precision and PN is the penalty (which is issued for having only unigrams).

$$\text{METEOR} = \frac{10RP}{R+P} \times (1 - PN) \tag{9}$$

In Eq. (10), C is the number of common chunks between the generated and reference summaries. In cases that the reference and generated summaries are identical, there is only one chunk. On the other hand, if there exists only uni-grams, number of chunks equals to the number of term-to-term mappings denoted as M_u.

$$PN = 0.5 \times \left(\frac{C}{M_u}\right)^3 \tag{10}$$

5.1.3. Evaluations setup

Our deep neural network uses Github data as an input resource. Some of the pairs did not have enough comment length. Therefore, we removed pairs with comments shorter than four words. Furthermore, some of the comments are too long to be used for training deep neural networks. Ying and Robillard (2014), claimed that most of the summaries are less than three sentences. Moreno et al. (2013a), stipulated that summaries with less than 20 terms are suitable for comment generation. Consequently, comments with more than 35 words were removed from the pairs. Similarly, source codes with more than 100 tokens were removed. Some of the comments neither were written in English nor were produced by a human. We removed these automatically generated comments as well. Finally, by applying a few minor heuristics (e.g., converting tokens to lowercase), we selected 71257 pairs of comment/code. Table 2 describes statistical information about these pairs.
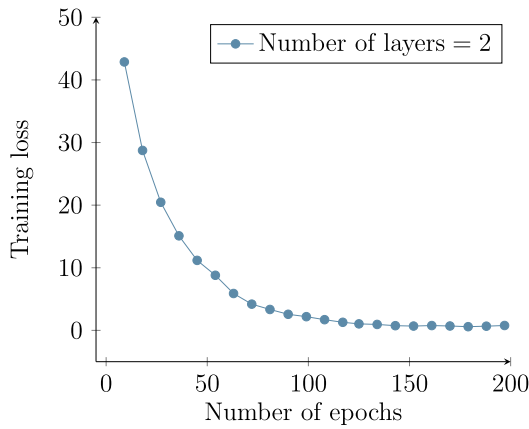
Fig. 7. Cross-entropy loss values based on different epochs.

Table 3
Results of the Proposed Deep Neural Network with Different Parameters.

# of layers	Pre-trained embedding layer	SBT	BLEU4	METEOR	Perplexity
4	✓	✗	30.8	11.3	3.53
4	✗	✗	31.0	6.7	5.16
3	✓	✗	30.7	11.8	1.95
2	✓	✗	30.9	12.7	**1.84**
2	✗	✓	30.3	7.8	5.47
2	✓	✓	**31.4**	**13.1**	2.18

Table 4
Comparing the proposed model to the baselines.

Approach	METEOR	BLEU4
Iyer et al. (2016)	18.4	20.0
Vaswani et al. (2017)	8.0	7.8
Allamanis et al. (2018)	18.3	20.5
Alon et al. (2019)	**19.1**	21.0
LeClair et al. (2019)	18.5	20.5
Haque et al. (2020)	18.9	21.3
Proposed approach	13.1	**31.4**

Table 5
Set of android applications used in the study.

Application name	# of lines	# of methods	# of classes
Tister	423	14	8
Hashpass	429	8	2
Munchlife	631	17	9
Justsit	849	43	13
Blinkenlightsbattery	851	61	14
Autoanswer	999	50	13
Anycut	1095	60	18
Dofcalculator	1321	14	9
Divideandconquer	1824	156	28
Passwordmakerpro	2824	282	67
TippyTipper	2953	148	36
Tokenlist	3680	225	43
Httpmon	4939	392	86
Remembeer	5915	257	54

5.1.4. Evaluations results

To answer the RQ1, we used *perplexity* metric (Brown et al., 1992). Perplexity estimates how well a deep neural network can perform on a training dataset. It is calculated using Perplexity = $\exp(\mathcal{L})$, in which $\mathcal{L}$ is the cross-entropy loss function. Table 3 shows the best perplexity values in the 10 last epochs. Moreover, Fig. 7 presents cross-entropy loss function based on different epochs.

To answer the RQ2, we used the BLEU4 metric. The maximum number of terms for generated summaries is 35, which is considered short. Therefore, based on the suggestion of Papineni et al. (2002), we used the maximum four-grams in calculating the value of this metric. Table 3 illustrates BLEU4 results based on different parameters. We set the number of epochs, batch size, and beam width in all cases to 200, 512, and 50, respectively. We achieved the best BLEU4 score, 31.4, using a network of two layers, employing both pre-trained embedding layer and SBT.

To answer the RQ3, we used METEOR metric. We achieved the best METEOR score, 13.1, using a network with the same properties mentioned above. Table 3 presents the values of discussed metrics for different parameters.

To answer the RQ4, we used six code summarization techniques proposed by Haque et al. (2020), LeClair et al. (2019), Alon et al. (2019), Allamanis et al. (2018), Vaswani et al. (2017), and Iyer et al. (2016) as our baselines. We selected these approaches because they are the state-of-the-art in the field of code summarization ranging from traditional sequence-based neural networks such as the RNN to the Transformer. Furthermore, they have been published in the leading venues of both SE and NLP. We used the publicly available implementation provided by Haque et al. (2020). Table 4 presents results of the proposed deep neural network's performance in compared with the baselines.

5.1.5. Quantitative analysis of results

According to Table 3, cross-entropy loss function has decreased 2.18 per word in perplexity. This indicates that the proposed model has efficiently performed on the training dataset.

Furthermore, according to Fig. 7, the model has not progressed significantly after epoch number 170. Therefore, we believe increasing the epoch number to more than 200 does not improve the performance of the model very much. It is worth mentioning that our proposed model does not necessarily compete with Hu et al. (2018). In fact, our model can complement their approach. To demonstrate this, we applied SBT to our model and obtained higher scores in terms of BLEU4 (31.4) comparing to the previous result without SBT (30.9). Table 4 compares the proposed neural network with the state-of-the-art. Our model significantly improved BLEU4 comparing to those approaches. In particular, comparing to Haque et al. (2020), our model improved BLEU4 from 21.3 to 31.4. However, except from Vaswani et al. (2017), other approaches have better METEOR than our model (13.1 compared to 19.1).

5.2. Evaluations of generated summaries

Here, we evaluate the usefulness of our model in aiding Android applications' comprehension using an empirical study.

5.2.1. Research questions

To evaluate generated summaries, we investigate the following questions:

RQ1 Considering the reference summaries, how accurate have been the generated summaries?

RQ2 How well has the proposed model performed compared to other approaches?

RQ3 How good is the quality of the generated summaries?

5.2.2. Evaluations setup

We used a dataset of 14 open-source Android applications with different sizes and features from previous studies (Mirzaei et al., 2016; Deng et al., 2017) to evaluate generated summaries. Table 5 presents some information about these applications.

First, we randomly selected three methods from each application and two experts who are not the authors of this paper,

worked with the 14 applications and investigated the related parts of the source code to know more about the context of each selected method. Then, wrote summaries for the 42 methods independently. After that, they compared their comments in a session and chose a common comment for each method. In case of disagreement, they asked a third expert for his opinion and they all came to an agreement for the comments of these methods together. It is worth mentioning that the two experts' average Android programming experience was 5.5 years and the third expert was a senior Android developer with 8 years of experience. Also, there was about 18% of disagreement between the first two experts, which all were resolved with the help of the third expert. Afterward, using the proposed model and the baseline approaches, we generated candidate summaries for each method. Note that our baselines are not specifically designed for event-driven programs, so the comparison may not be completely fair. However, these are the closest approaches to our problem that we are aware of so far. We designed an online questionnaire to qualitatively assess the performance of these models. Each question in the questionnaire contains (1) a method, (2) its reference summary, and (3) a generated summary by one of the baselines. Fig. 8 presents a question from the questionnaire. We asked the participants to score the generated summary based on the method itself and the reference summary provided to them. The method along with its reference summary help the participants obtain a better understanding of each method and its context. Note that by knowing what exactly each method does, participants were stricter in evaluating the generated summaries. This helped us remove the chance of careless or wrong scoring. Note that the generated summaries were presented in random orders, to remove any bias from the experiment. That is participants did not know which comment was written by which approach. As both the number of methods and baselines in our study are high, we could not evaluate each method with all the six baselines plus our approach using all participants. This would require a very long questionnaire which subsequently would result in a decline in the accuracy of evaluations. Therefore, we randomly assigned the questions to them and made sure to cover each of the 42 methods at least by 7 participants for each baseline.

The 42 participants were majored in computer science, with an average of 7.6 years of general programming experience, 5.8 years of Java programming experience, and 4.3 years of Android programming experience. It took each participant on average 73 min to finish the questionnaire. We analyzed the generated summaries from two perspectives; their *informativeness* and *naturalness* (Iyer et al., 2016);

Informativeness: What proportion of the important parts of the code does the generated summary cover.

Naturalness: How smooth and human-readable is the generated summary. Naturalness also takes into account the syntax of each sentence.

Participants scored the generated summaries for each method based on a 1–5 star scaling. Description of each score is as follows:

- Informativeness:

  1. The output does not describe the method's functionality to any extent.
  2. Only insignificant parts of the code are covered in the summary.
  3. Some important parts of the code are covered in the summary.
  4. Most of the important parts are covered in the summary.

Fig. 8. A sample question from the questionnaire.

Table 6
Comparing the proposed model to the baselines.

Approach	BLEU4	METEOR
Iyer et al. (2016)	11.4	17.9
Vaswani et al. (2017)	8.6	9.7
Allamanis et al. (2018)	13.1	**18.0**
Alon et al. (2019)	11.1	16.4
LeClair et al. (2019)	12.0	16.9
Haque et al. (2020)	12.1	16.0
Proposed approach	**32.3**	11.2

  5. All significant and essential parts of the code are well summarized.

- Naturalness:

  1. The output is not readable by humans at all.
  2. The output is barely understandable with many errors.
  3. The output is understandable but has noticeable syntax errors.
  4. The output is understandable but has negligible syntactical errors.
  5. The output is completely understandable with no syntactical errors.

5.2.3. Evaluations results

To answer the RQ1, we calculated BLEU4 and METEOR metrics for each method. We compared our new results to the Iyer et al. (2016), Vaswani et al. (2017), Allamanis et al. (2018), Alon et al. (2019), LeClair et al. (2019), and Haque et al. (2020) approaches to answer the RQ2. Tables 6 presents the comparison results of these approaches.

To answer the RQ3, Figs. 9 and 10 present the distribution of informativeness and naturalness variables. The mean scores for informativeness and naturalness of our proposed approach for all participants are 3.51 and 4.36. The results indicate that if one only considers rating 5 (perfect summaries) and rating 4 (good enough summaries with negligible mistakes) as desirable outcomes of a summarization method, our approach outperforms all the others regarding both informativeness and naturalness.

5.2.4. Quantitative analysis of results

According to Table 6, average BLEU4 and METEOR of the proposed approach are 32.3 and 11.2, respectively. To investigate whether there is a significant difference between the results of our proposed approach and other existing approaches, we followed the guideline and the tool provided by Herbold (2020). We
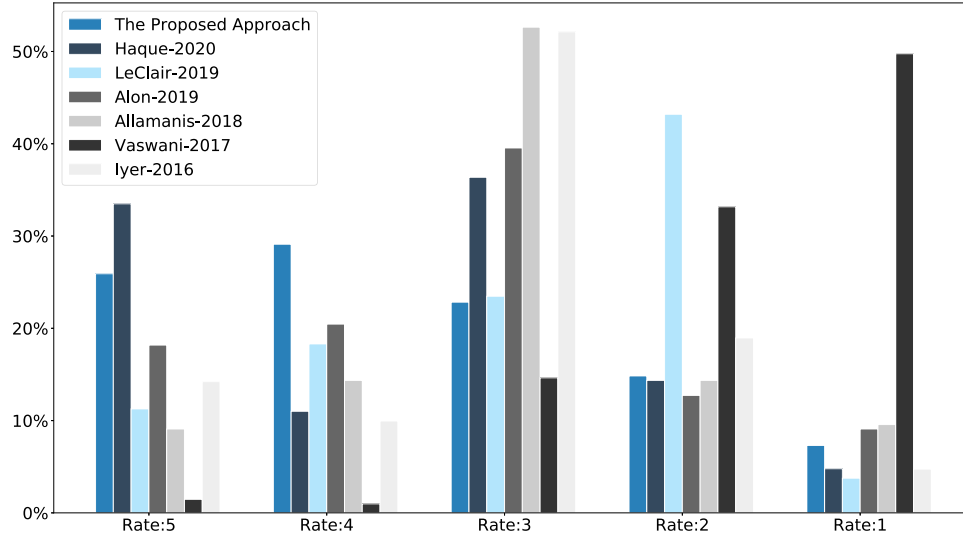
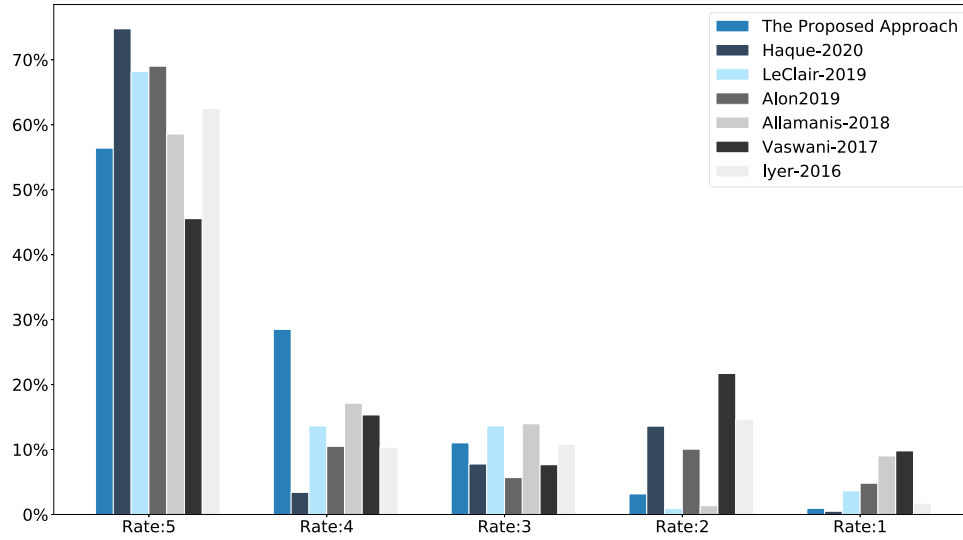**Fig. 9.** Distribution of informativeness.

Fig. 10. Distribution of Naturalness.

conducted a statistical analysis for 7 approaches with 42 samples. We used the non-parametric Friedman test to investigate difference between the median values of the approaches (Friedman, 1940). We employed the post-hoc Nemenyi test to determine which aforementioned differences are statistically significant (Nemenyi, 1962). The Nemenyi test used *critical distance* (CD) to evaluate which one is significant. If the difference is greater than CD, then the two approaches are statistically significant different.

Fig. 11 depicts the results of tests for BLEU4. The Friedman test rejects the null hypothesis that there is no difference between median values of the approaches. Consequently, we accept the alternative hypothesis that there is a difference between the approaches. Based on Fig. 11 and the post-hoc Nemenyi test, we cannot say that there are significant differences within the following approaches: (the proposed approach, Allamanis et al. (2018), Haque et al. (2020) and Iyer et al. (2016)); (Allamanis et al. (2018), Haque et al. (2020), Iyer et al. (2016), LeClair et al. (2019), and Alon et al. (2019)); (LeClair et al. (2019), Alon et al. (2019), and Vaswani et al. (2017)). All of the other differences are statistically significant.

Fig. 12 shows the results of tests for METEOR. The Friedman test rejects the null hypothesis that there is no difference between median values of the approaches. Consequently, we accept the alternative hypothesis that there is a difference between the approaches. Based on the Fig. 12 and the post-hoc Nemenyi test, we cannot say there are significant differences within the following approaches: (Allamanis et al. (2018), Iyer et al. (2016), Alon et al. (2019), Haque et al. (2020), and LeClair et al. (2019)); (Iyer et al. (2016), Alon et al. (2019), Haque et al. (2020), (LeClair et al., 2019), and the proposed approach); (the proposed approach and Vaswani et al. (2017)). All of the other differences are statistically significant.

Sample means for informativeness and naturalness of the proposed approach are $\overline{X}_{\text{inf}} = 3.51$ and $\overline{X}_{\text{nat}} = 4.36$, respectively. We applied t-distribution to estimate mean and standard deviation of informativeness and naturalness results. Using Eq. (11), and the confidence level of 95% ($\alpha = 0.05$),

$$\hat{\sigma}_{\overline{X}_{\text{inf}}} = \frac{s_{\text{inf}}}{\sqrt{N}} = 0.19$$
$$\hat{\sigma}_{\overline{X}_{\text{nat}}} = \frac{s_{\text{nat}}}{\sqrt{N}} = 0.13.$$
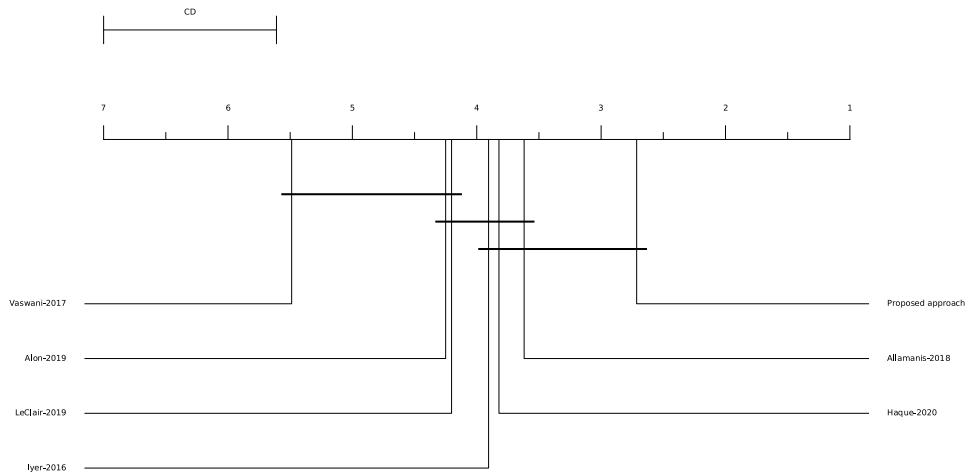
$$(11)$$

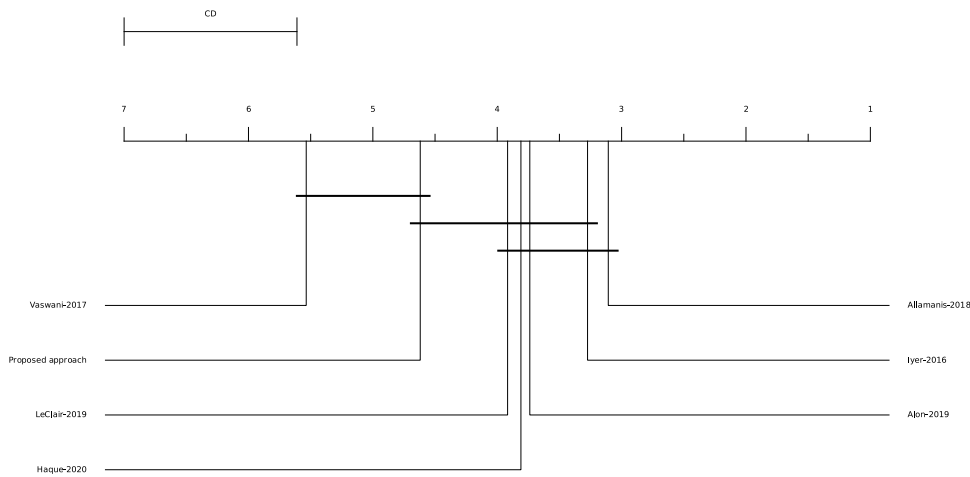**Fig. 11.** The results of hypothesis testing for BLEU4.

Fig. 12. The results of hypothesis testing for METEOR.

We can compute equation (12),

$$\mu = \overline{X} \pm t_{(N-1=41, \alpha=0.05)} \times \hat{\sigma}_{\overline{X}}. \tag{12}$$

Therefore, we can conclude that

$$\begin{aligned} \hat{\sigma}_{\overline{X}_{\text{inf}}} &= 3.51 \pm 0.38 \\ \hat{\sigma}_{\overline{X}_{\text{nat}}} &= 4.36 \pm 0.27. \end{aligned} \tag{13}$$

Eq. (13) shows that with the confidence level of 95%, by increasing the number of participants, in the worst case the mean scores for informativeness and naturalness will be greater than 3.13 and 4.09, respectively.

5.2.5. Qualitative analysis of results

According to Figs. 9, 10 and our definition of informativeness and naturalness metrics, we conclude as follows:

1. Participants in 25.9% of cases reported that generated summaries cover all essential parts of the codes.
2. Participants in 55.0% of cases reported that in the worst case, generated summaries cover many salient features of the code.
3. Participants only in 22.1% of cases reported that generated summaries are not related to the codes or document just trivial code snippets.
4. Participants in 45.0% of cases reported that generated summaries have neglected a few necessary parts of the codes.

5. Participants in 56.3% of cases reported that generated summaries are human-readable and do not have any syntactical error.
6. Participants in 84.8% of cases reported that in the worst case, the generated summaries have minor syntactical errors.
7. Participants in 11.0% of cases reported that generated summaries are human-readable but have major syntactical errors.
8. Finally, participants only in 4.1% of cases reported that generated summaries are barely human-readable.

5.2.6. Threats to validity of the empirical study

In this section we discuss the threats to validity of our empirical study. We evaluated the quality of final summaries extracted from 14 Android applications and 42 methods. It is reasonable that the quality of extracted methods affects results. To reduce this threat, we sampled randomly from extracted methods. Moreover, 42 individuals performed our qualitative assessment. Therefore, the outcome of this section depends on characteristics of the individuals taking the questionnaire, such as their mood, the time it took them to fill the questionnaire, and other natural factors. To reduce this threat, we tried to have a large number of participants. Note that for a large-scale evaluation, we need a dataset of Android methods, their comments and the APK file as the requirement for the Rundroid tool. However, to the best of

```
1  // Gen: Returns the elapsed time of states
2  // Gen: Called when the activity is updated
3  // Ref: Return how much time has passed since the starting time in milliseconds.
4  public long getElapsed() {
5      return System.currentTimeMillis() - this.start;
6  }
```

Fig. 13. A utility function extracted from the HTTPMON application.

our knowledge, available datasets only contain Java methods, and unfortunately, we were unable to collect such a large dataset for Android methods. It is worth mentioning that one can collect a dataset for a large-scale study; however, this is not a trivial task. But we plan to address this issue in future work. Moreover, we investigated whether there is a significant difference between the results of our proposed approach and other existing approaches using the Friedman and post-hoc Nemenyi test. We have reported the results in Section 5.2.4. The results indicate that there is a significant difference between our approach and other baselines regarding performance based on BLEU and METEOR. Also, in the same section, we proved that with the confidence level of 95%, by increasing the number of participants, in the worst case the mean scores for informativeness and naturalness of our approach will be greater than 3.13 and 4.09 (scale of 5 stars), respectively. We believe all these can attest to the good performance of our approach in a large-scale study.

As mentioned above, 42 individuals performed our qualitative assessment of the generated summaries for the selected real-world Android applications' methods. Because the number of participants is limited, we cannot extend our results to the rest of the developers' community. To reduce this threat, we have tried to select a well-distributed sample of developers to assist in the evaluation phase. We also investigated whether there is a significant difference between our results and other existing approaches' results. We performed the Friedman and post-hoc Nemenyi test and reported the results.

5.3. Discussion

Here, we discuss the limitations of our proposed approach. The quality of generated comments would be restricted when a target method did not represent the characteristics of event-driven programs. Helper functions (e.g., math functions), utility functions (e.g., logging) and interfaces are among these methods. Fig. 13 demonstrates a utility function which calculates the time elapsed from the start. This function is used in various methods to compute the elapsed time. Indeed, there is no meaningful context for this method. Consequently, the second generated comment does not relate to the method's functionality or context. Our proposed approach fail to generate high quality comments for this method. One way to handle this situation is to apply threshold analysis. For example, if the block rank is less than given threshold, it means that the method does not have meaningful context and one could simply ignore that block. However, in practice, determining the threshold value automatically is not a trivial task, and can be addressed in future work.

6. Threats to validity

In this section, we review threats to the validity of our research findings, categorizing possible threats into four groups of internal, external, construct and dependability ones (Feldt and Magazinius, 2010).

6.1. Internal validity

Internal validity asks whether the variables used in the proposed approach affect the outcomes and whether they are the only influential factors in the study (Feldt and Magazinius, 2010).

The dynamic call graph in Rundroid is constructed based on tests that are run on Android applications. These tests are run manually in the original version of the study (Yuan et al., 2017). Therefore, how the tests are run and their runtime impact results. To reduce this threat, authors generated 5000 random events using the Monkey tool to minimize human intervention in the tests.

We believe the quality of code snippets affects the quality of generated summaries as well. Logically training the models on high-quality source codes can help the model generate better summaries. However, not all real-life projects benefit from high-quality source code. Moreover, quality is a subjective concept and can be interpreted differently in various cases. Therefore, it is not a trivial task to collect a high-quality dataset of code snippets.

In case of a tie in Section 4.5, we randomly choose a node. We did not investigate the effects of this choice since they were rare cases that a tie happened. So we believe it was not necessary. But we acknowledge that different decisions in these cases can provide different outcome.

6.2. External validity

External validity includes how expandable are results of a study, can they be used in other contexts, and do cause and effect relationships hold with other conditions as well (Feldt and Magazinius, 2010).

In this study, we have used Rundroid to build call graphs. Rundroid is developed for generating call graphs in the Android framework. Therefore, it is not suitable for usage in other event-driven programs. To reduce this threat, we plan to investigate and use other tools in near future. In the first and second phase of the proposed approach, we have used deep neural networks. The deep neural network architecture can be employed in other contexts, namely other natural language processing fields.

Moreover, in the evaluation phase, we have only used the Android framework's examples as event-driven programs. Thus, it is not guaranteed that our approach will perform the same on other event-driven programs such as web-based programs. In future, we plan to address this threat by evaluating our model on other event-driven platforms.

6.3. Construct validity

Construct validity includes theoretical concepts and discussions of the experiment and the use of appropriate evaluation metrics (Feldt and Magazinius, 2010).

Theoretical concepts used in this work, have been already evaluated and proved by the academic society. The proposed approach is a combination of different methods in a new context. We have evaluated the generated summaries not only by valid and reliable quantitative metrics but also through human qualitative judgment. Results indicate that the employed approach has been successful in generating summaries.

6.4. Dependability

Dependability validity answers to questions such as whether the findings are compatible, and whether the experiment and its results are reproducible (Feldt and Magazinius, 2010).

Compatibility We evaluated the final generated summaries quantitatively and qualitatively. As shown in previous sections, their outcomes are compatible.

Reproducibility We have used deep neural networks (which are inherently based on probability) to generate the summaries of event-driven programs' methods. To reduce this threat, we have set the number of epochs to 200. This is because cross-entropy loss function is almost stable after the 170th epoch and did not decrease in our experiments. Also, the preprocessed input data is available online for other researchers at https://github.com/ase-sharif/deep-code-document-pairs. It is worth mentioning that we have tried our best to minimize human intervention in all steps to make results more independent and reliable.

7. Conclusions and future work

Code summarization is a useful technique for helping developers comprehend and maintain software programs more efficiently. There are different approaches for summarizing code segments, namely utilizing information retrieval, machine learning, and crowdsourcing knowledge. However, existing approaches do not take into account the interactions between different parts of the code while the program is running. Through exploiting deep neural networks and dynamic generation of the call graph, we tried to overcome the deficiencies of previous work and generate summaries that are more suitable. Results of the proposed approach were evaluated both qualitatively and quantitatively. We used BLEU4, METEOR, precision and recall metrics for quantitative assessment and an online questionnaire for assessing the informativeness and naturalness of generated summaries from developers' perspectives as a means of qualitative assessment. Our results indicate that the proposed approach outperforms state-of-the-art techniques.

As for future work, one of the conventional solutions while using the sequence-to-sequence models is to employ a convolutional layer in the encoding component (Kalchbrenner and Blunsom, 2013). Adding this layer helps the deep neural network attain additional information about the words around each word. The use of a convolutional layer has improved results in machine translation studies. We intend to exploit this layer in future and analyze its effect on the proposed model.

Moreover, the Android framework is only one example of event-driven applications. In the future, we are going to examine other frameworks to evaluate the proposed approach and expand our findings.

CRediT authorship contribution statement

Alireza Aghamohammadi: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing - original draft. **Maliheh Izadi:** Conceptualization, Methodology, Software, Validation, Formal analysis, Writing - original draft. **Abbas Heydarnoori:** Project administration, Supervision, Methodology, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P.A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., 2016. Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th Conference on Symposium on Operating Systems Design and Implementation. USENIX Association, pp. 265–283.

Allamanis, M., Brockschmidt, M., Khademi, M., 2018. Learning to represent programs with graphs. In: Proceedings of the 6th International Conference on Learning Representations, ICLR 2018. OpenReview.net.

Allamanis, M., Peng, H., Sutton, C.A., 2016. A convolutional attention network for extreme summarization of source code. In: Proceedings of the 33nd International Conference on Machine Learning, vol. 48. JMLR.org, pp. 2091–2100.

Alon, U., Brody, S., Levy, O., Yahav, E., 2019. Code2seq: Generating sequences from structured representations of code. In: Proceedings of the 7th International Conference on Learning Representations, ICLR 2019. OpenReview.net.

Anon, 2017. Start another activity. https://developer.android.com/training/basics/firstapp/starting-activity. (Accessed 20 August 2017).

Anon, 2018a. Android play location. https://github.com/googlesamples/android-play-location/blob/master/LocationAddress/java/app/src/main/java/com/google/android/gms/location/sample/locationaddress/MainActivity.java. (Accessed 14 October 2018).

Anon, 2018b. GloVe: Global vectors for word representation. https://nlp.stanford.edu/projects/glove/. (Accessed 17 July 2018).

Anon, 2018c. Recurrent neural networks and sequence models. https://youtu.be/efWlOCE_6HY. (Accessed 15 October 2018).

Anon, 2018d. UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey. (Accessed 21 July 2018).

Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E., 2002. Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. 28 (10), 970–983. http://dx.doi.org/10.1109/tse.2002.1041053.

Badihi, S., Heydarnoori, A., 2017. CrowdSummarizer: Automated generation of code summaries for Java programs through crowdsourcing. IEEE Softw. 34 (2), 71–80. http://dx.doi.org/10.1109/ms.2017.45.

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

Banerjee, S., Lavie, A., 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/Or Summarization. Association for Computational Linguistics, pp. 65–72.

Brin, S., Page, L., 1998. The anatomy of a large-scale hypertextual web search engine. Comput. Netw. 30 (1–7), 107–117. http://dx.doi.org/10.1016/s0169-7552(98)00110-x.

Brown, P.F., Pietra, S.D., Pietra, V.J.D., Lai, J.C., Mercer, R.L., 1992. An estimate of an upper bound for the entropy of english. Comput. Linguist. 18 (1), 31–40.

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP, pp. 1724–1734. http://dx.doi.org/10.3115/v1/d14-1179.

Chung, J., Gülçehre, Ç., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

Collobert, R., Weston, J., 2008. A unified architecture for natural language processing: deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning, vol. 307. ACM, pp. 160–167. http://dx.doi.org/10.1145/1390156.1390177.

Deng, L., Offutt, J., Samudio, D., 2017. Is mutation analysis effective at testing Android Apps?. In: Proceedings of the 17th International Conference on Software Quality, Reliability and Security. IEEE, pp. 86–93. http://dx.doi.org/10.1109/qrs.2017.19.

Eddy, B.P., Robinson, J.A., Kraft, N.A., Carver, J.C., 2013. Evaluating source code summarization techniques: Replication and expansion. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. IEEE, pp. 13–22. http://dx.doi.org/10.1109/icpc.2013.6613829.

Feldt, R., Magazinius, A., 2010. Validity threats in empirical software engineering research - an initial survey. In: Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering. Knowledge Systems Institute Graduate School, pp. 374–379.

Fowkes, J.M., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., Sutton, C.A., 2017. Autofolding for source code summarization. IEEE Trans. Softw. Eng. 43 (12), 1095–1109. http://dx.doi.org/10.1109/tse.2017.2664836.

Friedman, M., 1940. A comparison of alternative tests of significance for the problem of *m* rankings. Ann. Math. Stat. 11 (1), 86–92. http://dx.doi.org/10.1214/aoms/1177731944.

Goldberg, Y., Levy, O., 2014. word2vec Explained: Deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv preprint arXiv:1402.3722.

Graves, A., 2012. Sequence transduction with recurrent neural networks. arXiv preprint arXiv:1211.3711.

Graves, A., Mohamed, A., Hinton, G.E., 2013. Speech recognition with deep recurrent neural networks. In: Proceedings of the 38th IEEE International Conference on Acoustics, Speech and Signal Processing. IEEE, pp. 6645–6649. http://dx.doi.org/10.1109/icassp.2013.6638947.

Graves, A., Schmidhuber, J., 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Netw. 18 (5–6), 602–610. http://dx.doi.org/10.1016/j.neunet.2005.06.042.

Gu, X., Zhang, H., Kim, S., 2018. Deep code search. In: Proceedings of the 40th International Conference on Software Engineering. ICSE, pp. 933–944. http://dx.doi.org/10.1145/3180155.3180167.

Guerrouj, L., Bourque, D., Rigby, P.C., 2015. Leveraging informal documentation to summarize classes and methods in context. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. IEEE, pp. 639–642. http://dx.doi.org/10.1109/icse.2015.212.

Haiduc, S., Aponte, J., Marcus, A., 2010a. Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2. ACM, pp. 223–226. http://dx.doi.org/10.1145/1810295.1810335.

Haiduc, S., Aponte, J., Moreno, L., Marcus, A., 2010b. On the use of automated text summarization techniques for summarizing source code. In: Proceedings of the 17th Working Conference on Reverse Engineering,. IEEE, pp. 35–44. http://dx.doi.org/10.1109/wcre.2010.13.

Haque, S., LeClair, A., Wu, L., McMillan, C., 2020. Improved automatic summarization of subroutines via attention to file context. CoRR abs/2004.04881.

Herbold, S., 2020. Autorank: A Python package for automated ranking of classifiers. J. Open Source Softw. 5 (48), 2173. http://dx.doi.org/10.21105/joss.02173.

Hochreiter, S., 1998. The vanishing gradient problem during learning recurrent neural nets and problem solutions. Int. J. Uncertain. Fuzziness Knowl.-Based Syst. 6 (02), 107–116. http://dx.doi.org/10.1142/S0218488598000094.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation. In: Proceedings of the 26th IEEE International Conference on Program Comprehension. http://dx.doi.org/10.1145/3196321.3196334.

Huang, Q., Xia, X., Lo, D., Murphy, G.C., 2018. Automating intention mining. IEEE Trans. Softw. 1–1. http://dx.doi.org/10.1109/tse.2018.2876340.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: ACL (1). The Association for Computer Linguistics, http://dx.doi.org/10.18653/v1/p16-1195.

Jiang, S., Armaly, A., McMillan, C., 2017. Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. pp. 135–146. http://dx.doi.org/10.1109/ase.2017.8115626.

Kalchbrenner, N., Blunsom, P., 2013. Recurrent continuous translation models. In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing. ACL, pp. 1700–1709.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

K.Landauer, T., Foltz, P.W., Laham, D., 1998. An introduction to latent semantic analysis. Discourse Process. 25 (2–3), 259–284. http://dx.doi.org/10.1080/01638539809545028.

LeClair, A., Jiang, S., McMillan, C., 2019. A neural model for generating natural language summaries of program subroutines. In: Atlee, J.M., Bultan, T., Whittle, J. (Eds.), Proceedings of the 41st International Conference on Software Engineering, ICSE 2019. IEEE / ACM, pp. 795–806. http://dx.doi.org/10.1109/ICSE.2019.00087.

Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z., Wang, X., 2018. Neural-machine-translation-based commit message generation: how far are we?. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 373–384. http://dx.doi.org/10.1145/3238147.3238190.

Luong, T., Pham, H., Manning, C.D., 2015. Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing. ACL, pp. 1412–1421. http://dx.doi.org/10.18653/v1/d15-1166.

McBurney, P.W., Liu, C., McMillan, C., Weninger, T., 2014. Improving topic model source code summarization. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 291–294. http://dx.doi.org/10.1145/2597008.2597793.

McBurney, P.W., McMillan, C., 2016. Automatic source code summarization of context for java methods. IEEE Trans. Softw. Eng. 42 (2), 103–119. http://dx.doi.org/10.1109/tse.2015.2465386.

McKinney, W., 2011. Pandas: A foundational python library for data analysis and statistics. In: Python for High Performance and Scientific Computing. pp. 1–9.

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013a. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Proceedings of the 26th Conference on Advances in Neural Information Processing Systems, pp. 3111–3119.

Mikolov, T., Yih, W., Zweig, G., 2013c. Linguistic regularities in continuous space word representations. In: Proceedings of the 11th Conference on Human Language Technologies. The Association for Computational Linguistics, pp. 746–751.

Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S., 2016. Reducing combinatorics in GUI testing of android applications. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 559–570. http://dx.doi.org/10.1145/2884781.2884853.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L.L., Vijay-Shanker, K., 2013a. Automatic generation of natural language summaries for java classes. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. IEEE, pp. 23–32. http://dx.doi.org/10.1109/icpc.2013.6613830.

Moreno, L., Marcus, A., Pollock, L.L., Vijay-Shanker, K., 2013b. Jsummarizer: An automatic generator of natural language summaries for Java classes. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. IEEE, pp. 230–232. http://dx.doi.org/10.1109/icpc.2013.6613855.

Nazar, N., Jiang, H., Gao, G., Zhang, T., Li, X., Ren, Z., 2016. Source code fragment summarization with small-scale crowdsourcing based features. Front. Comput. Sci. 10 (3), 504–517. http://dx.doi.org/10.1007/s11704-015-4409-2.

Nemenyi, P., 1962. Distribution-free multiple comparisons. In: Biometrics. p. 263.

Page, L., Brin, S., Motwani, R., Winograd, T., 1999. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab.

Papineni, K., Roukos, S., Ward, T., Zhu, W., 2002. Bleu: A method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. ACL, pp. 311–318. http://dx.doi.org/10.3115/1073083.1073135.

Pascanu, R., Mikolov, T., Bengio, Y., 2013. On the difficulty of training recurrent neural networks. In: Proceedings of the 30th International Conference on Machine Learning, vol. 28. JMLR.org, pp. 1310–1318.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Moschitti, A., Pang, B., Daelemans, W. (Eds.), Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. ACL, pp. 1532–1543. http://dx.doi.org/10.3115/v1/d14-1162.

Rahman, M.M., Roy, C.K., Keivanloo, I., 2015. Recommending insightful comments for source code using crowdsourced knowledge. In: Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 81–90. http://dx.doi.org/10.1109/scam.2015.7335404.

Rodeghero, P., McMillan, C., McBurney, P.W., Bosch, N., D'Mello, S., 2014. Improving automated source code summarization via an eye-tracking study of programmers. In: Proceedings of the 36th International Conference on Software Engineering. pp. 390–401. http://dx.doi.org/10.1145/2568225.2568247.

Schuster, M., Paliwal, K.K., 1997. Bidirectional recurrent neural networks. IEEE Trans. Signal Process. 45 (11), 2673–2681. http://dx.doi.org/10.1109/78.650093.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L.L., Vijay-Shanker, K., 2010. Towards automatically generating summary comments for Java methods. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. pp. 43–52. http://dx.doi.org/10.1145/1858996.1859006.

Srivastava, N., Hinton, G.E., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting. Mach. Learn. Res. 15 (1), 1929–1958.

Sutskever, I., Vinyals, O., Le, Q.V., 2014. Sequence to sequence learning with neural networks. In: Proceedings of the 27th Conference on Advances in Neural Information Processing Systems, pp. 3104–3112.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. Curran Associates Inc., pp. 6000–6010.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S., 2018. Improving automatic source code summarization via deep reinforcement learning. In: Huchard, M., Kästner, C., Fraser, G. (Eds.), Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE. ACM, pp. 397–407. http://dx.doi.org/10.1145/3238147.3238206.

Werbos, P., 1990. Backpropagation through time: what it does and how to do it. Proc. IEEE 78 (10), 1550–1560. http://dx.doi.org/10.1109/5.58337.

Wong, E., Liu, T., Tan, L., 2015. CloCom: Mining existing source code for automatic comment generation. In: Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering. IEEE, pp. 380–389. http://dx.doi.org/10.1109/saner.2015.7081848.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2018. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. Softw. Eng. 44 (10), 951–976. http://dx.doi.org/10.1109/tse.2017.2734091.

Ying, A.T.T., Robillard, M.P., 2014. Selection and presentation practices for code example summarization. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 460–471. http://dx.doi.org/10.1145/2635868.2635877.

Yuan, Y., Xu, L., Xiao, X., Podgurski, A., Zhu, H., 2017. RunDroid: Recovering execution call graphs for android applications. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp. 949–953. http://dx.doi.org/10.1145/3106237.3122821.

Alireza Aghamohammadi is a Ph.D. candidate at the Sharif University of Technology. He works on a wide range of recommendation systems that exploit state-of-the-art machine learning techniques. He is also a software developer for more than seven years. Contact him at aaghamohammadi@ce.sharif.edu.

Maliheh Izadi is a Ph.D. candidate of software engineering at the Sharif University of Technology. She is also currently a visiting Ph.D. student to the Software Engineering Research Group at the Delft University of Technology, Netherlands. Her research interests include data science for software engineering, natural language processing, and mining software repositories. Contact her at maliheh.izadi@sharif.edu.

Abbas Heydarnoori is an assistant professor at the Sharif University of Technology. Before, he was a post-doctoral fellow at the University of Lugano, Switzerland. Abbas holds a Ph.D. from the University of Waterloo, Canada. His research interests focus on software evolution, mining software repositories, and software analytics. Contact him at heydarnoori@sharif.edu.