Which Variables Should I Log?

Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E. Hassan, Shanping Li

Abstract—Developers usually depend on inserting logging statements into the source code to collect system runtime information. Such logged information is valuable for software maintenance. A logging statement usually prints one or more variables to record vital system status. However, due to the lack of rigorous logging guidance and the requirement of domain-specific knowledge, it is not easy for developers to make proper decisions about which variables to log. To address this need, in this work, we propose an approach to recommend logging variables for developers during development by learning from existing logging statements. Different from other prediction tasks in software engineering, this task has two challenges: 1) Dynamic labels – different logging statements have different sets of accessible variables, which means in this task, the set of possible labels of each sample is not the same. 2) Out-of-vocabulary words – identifiers' names are not limited to natural language words and the test set usually contains a number of program tokens which are out of the vocabulary built from the training set and cannot be appropriately mapped to word embeddings. To deal with the first challenge, we convert this task into a representation learning problem instead of a multi-label classification problem. Given a code snippet which lacks a logging statement, our approach first leverages a neural network with an RNN (recurrent neural network) layer and a self-attention layer to learn the proper representation of each program token, and then predicts whether each token should be logged through a unified binary classifier based on the learned representation. To handle the second challenge, we propose a novel method to map program tokens into word embeddings by making use of the pre-trained word embeddings of natural language tokens. We evaluate our approach on 9 large and high-quality Java projects. Our evaluation results show that the average MAP of our approach is over 0.84, outperforming random guess and an information-retrieval-based method by large margins.

Index Terms—Log, Logging Variable, Word Embedding, Representation Learning

1 Introduction

In software development, logging is pervasively used to record runtime information of software systems [1]. Logs are important and valuable for various of software maintenance tasks, such as execution anomaly detection [2], [3], problem diagnosis [4]–[7], deployment verification [8], etc. Since logs are mainly produced by the logging statements inserted by developers in the source code, writing appropriate and high-quality logging statements are of great importance to facilitate these maintenance tasks.

As shown in the example below, a logging statement typically specifies a log level, a static text and one or more variables [9]–[11]. The log level (e.g., *info* in our example) specifies the verbosity level of a log entry. Meanwhile, the static text (e.g., *"Client run completed. Result="* in our example) describes some contextual information. Additionally, the variables (e.g., *result* in our example, which is the result of an IO operation), which are the focus of this work, contain important system status that needs to be logged for postmortem analysis.

```
logger.info("Client run completed. Result=" + result);
```

It is not practical for developers to log too many variables in logging statements, since this may incur performance overhead and prevent developers from zooming in on real problems [1]. In addition, missing logging vital variables can increase developers' burden in performing many corrective software maintenance tasks. For example, developers may not be able to understand the root cause of a problem affecting a deployed system well since important variables are not logged. Therefore, to write high-quality logging statements, developers need to decide which variables storing important values to be logged. However, it is not easy to make such decisions due to the following reasons. First, there is a lack of rigorous specification to guide developers' logging practices. For example, Fu et al. [10] have shown that even in Microsoft, a leading software company, it is hard to find thorough and complete logging specifications to guide developers' logging practices. Second, choosing appropriate variables in most cases requires domain-specific knowledge. High-level guidelines of logging variables or logging "rules" learned from one project may not be practical or appropriate when developers switch to a new project. Moreover, some empirical studies have shown that in their studied open source systems, over 25% of the changes to logging statements are related to variable logging [9], [11]. This highlights the need for a tool that can recommend logging variables for developers to help them write high-quality logging statements.

To address this need, in this work, we propose a

- *Zhongxin Liu and Shanping Li are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China.*
  E-mail: {liu_zx, shan}@zju.edu.cn
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, Australia.*
  E-mail: xin.xia@monash.edu
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *Zhenchang Xing is with the Research School of Computer Science, Australian National University, Australia.*
  E-mail: zhenchang.xing@anu.edu.au
- *Ahmed E. Hassan is with School of Computing, Queen's University, Canada*
  E-mail: ahmed@cs.queensu.ca
- *Xin Xia is the corresponding author.*

learning-based approach to help developers decide which variables to log during development. Our approach first learns logging "rules" about variables from existing logging statements and their contextual source code through a neural network. Then, given a new code snippet that needs to be logged, our approach can automatically suggest which variables should be logged.

The usage scenarios of our proposed tool are as follows:

Without Tool. Bob is a junior developer in a company. He always adds logging statements in his code according to the high-level logging guidelines of the company and his limited development experience. In his workflow, after he deploys his system in the production environment, he needs to monitor the system's status through the produced logs. If his system performs abnormally or even encounters a failure, the logs are precious information source for Bob to find the root cause of the anomaly or the failure. However, Bob sometimes finds that he did not log some important system status in the logging statements. So, he needs to modify his code, add the related variables into the logging statements, and re-deploy his system. While Bob only makes some trivial changes to his code, it takes a non-trivial amount of time to wait for his code passing all the tests and deploying in the production environment. Moreover, sometimes Bob can not easily modify the production environment in order to guarantee the stability of the system. Thus, missing recording important variables reduces Bob's productivity and makes it difficult for Bob to conduct maintenance tasks.

With Tool. Bob adopts our tool. Each time he adds a logging statement, our tool will recommend a ranked list of variables which may need to be logged. Bob can quickly review the list to check whether he misses any important variables to log. With the help of our tool, Bob can reduce the likelihood of him to miss recording important variables in the logging statements at the first time. Therefore, Bob spends less time making changes to variable logging and his maintenance work becomes less painful.

Some approaches have been proposed to help developers make informed logging decisions during development by learning from existing logging statements, i.e., "learning to log" [1], [12]–[14]. Some of them aimed to predict whether to instrument a code snippet with logging statements, i.e., addressing "where to log" issue [1], [12]. As for "what to log", Li et al. proposed a technique to learn to suggest appropriate log levels for newly-added logging statements [13]. He et al. characterized the static texts in logging statements and designed an IR-based method to generate static descriptions automatically [14]. Different from them, this work focuses on logging variables, and our approach can be used to complement them. For example, after a "where to log" tool predicts that a code snippet should be logged, our approach can be further used to infer which variables should be logged, and such variables may help developers understand why a logging statement should be inserted here.

Different from other software engineering prediction tasks (e.g., defect prediction), automatically suggesting logging variables requires us to overcome several challenges:

Dynamic labels. A logging statement will only print out the values of variables that can be accessed. Since different code snippets usually contain different sets of accessible variables, in this task, for different "samples" (i.e., code snippets), the sets of possible "labels" (i.e., variables) are also different. It is not suitable to simply regard this task as a multi-label classification task, and traditional approaches for solving multi-label classification tasks, e.g., building a binary classifier for each label, can not be easily adapted to solve this problem due to the non-uniform set of possible labels. Moreover, considering the union of all variables appearing in all samples as the set of labels will result in poor performance, as this increases the difficulty of the multi-label classification task. With these observations, we propose a novel approach to effectively solve this problem. Our framework takes a sequence of program tokens as input, i.e., a tokenized code snippet. To predict the logging variables, our framework first learns to represent each program token as a vector of real numbers through a neural network with an RNN (recurrent neural network) layer and a self-attention layer. We expect such token representation can contain sufficient information to predict whether this token should be logged or not. Then, our framework predicts the probability of each identifier being logged through a unified binary classifier. In this way, the dynamic-labels problem can be properly solved.

Out-of-vocabulary words. To leverage neural networks to process program tokens, we first need to build a vocabulary from the training set and represent each program token in the vocabulary in the vector form. Different from natural language, code snippets contain numerous tokens that can be constructed by concatenating multiple simple tokens following various coding conventions such as camel casing, snake casing, etc. We refer to such tokens as *compound words*. It is common that a new code snippet contains new *compound words*, e.g., this code snippet declares some new variables. Such new *compound words* can not be found in the training vocabulary, i.e., the out-of-vocabulary issue happens, and can not be properly represented as word vectors. To solve this problem, we propose a novel method to map program tokens into word vectors. First, we split all *compound words* into simple tokens according to camel casing and snake casing coding conventions. These simple tokens are usually English words or their abbreviations. Then, we leverage the GloVe [15] pre-trained word embeddings to map split simple tokens and *non-compound words* to word vectors. For *non-compound words*, such word vectors are input to neural networks directly. For a *compound word*, we treat the average vector of the word vectors of its split tokens as its vector representation.

We evaluate our approach on 9 large, high-quality Java open source projects. Evaluation results show that the average MAP (Mean Average Precision) of our approach is over 0.84, outperforming all the baseline methods by large margins. We also apply our approach to make cross-project predictions and our experiment results show that the average MAP of our approach for this setting is over 0.77. This means our approach is also effective for cross-project logging variable predictions.

The contributions of this work are as follows:

- We propose a novel method to map program tokens to word vectors, which alleviates the out-of-

vocabulary problem incurred by *compound words* in programs.
- We propose an approach to suggest proper logging variables to developers during development.
- We evaluate our approach on 9 large and high-quality Java open source projects. We also evaluate the effectiveness of our approach for cross-project logging variable predictions. The evaluation results show that the performance of our approach is promising.

The remainder of this paper is organized as follows: Section 2 provides some background knowledge of RNN, word embedding, and the self-attention mechanism. Section 3 formalizes the problem and describes our approach in detail. Section 4 presents our experimental settings, including our studied software systems and the procedures of data extraction and data preprocessing. In Section 5, we describe our research questions and corresponding evaluation results. Section 6 discusses our case studies on real-world logging bugs, the rationales of some design decisions, how to apply our approach to low-logging-quality projects and threats to the validity of our approach. Section 7 surveys the related work. We conclude this work and point out some future research directions in Section 8.

2 Background

Our approach leverages a novel word embedding method and a neural network with an RNN layer and a self-attention layer to recommend proper variables. We first introduce some background knowledge of these techniques. Since the input of our approach is a code snippet, i.e., a sequence of program tokens, to be convenient, we let $\boldsymbol{x} = (x_1, x_2, ..., x_{|\boldsymbol{x}|})$ represent a sequence of program tokens, where $|\cdot|$ denotes the length of a sequence.

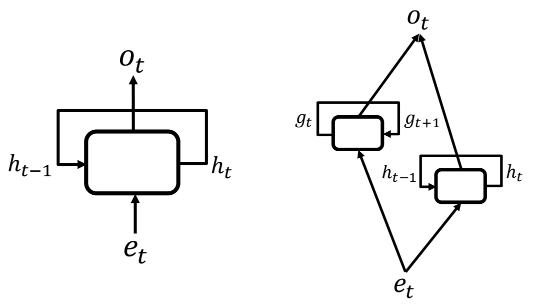
2.1 Bidirectional RNN

An RNN is a neural network which is specialized for processing sequence data [16]. Unlike feedforward neural networks and CNN (convolutional neural networks), RNNs can handle sequences of variable length, thus are widely used in the field of natural language processing (NLP). Figure 1a shows the basic structure of an RNN. This neural network takes vectors as input. To process a sequence of words $\boldsymbol{x}$ using the RNN, we need first map each word into a vector. Then, the RNN reads such vectors one by one and outputs a vector to represent the word at each time step.

Specifically, at step t, the RNN takes the word vector $\boldsymbol{e_t}$ as input, and computes the hidden state $\boldsymbol{h_t}$ of this step according to the previous hidden state $\boldsymbol{h_{t-1}}$ and the input $\boldsymbol{e_t}$:

$$\boldsymbol{h_t} = f(\boldsymbol{h_{t-1}}, \boldsymbol{e_t}) \tag{1}$$

Long short-term memory (LSTM) [17] and gated recurrent unit (GRU) [18] are two popular implementations of f. Both of them are capable of learning long-term dependencies. For most implementations of the RNN, the hidden state $\boldsymbol{h_t}$ is also the output $\boldsymbol{o_t}$ at step t. $\boldsymbol{o_t}$ can be regarded as the learned representation of the word x_t, which is able to adaptively capture the information of the first $t-1$ words (i.e., x_1, x_2, ..., x_{t-1}).



(a) RNN (b) Bidirectional RNN

Fig. 1: The structures of RNN and bidirectional RNN

However, $\boldsymbol{o_t}$ can not capture the information of the words after x_t which is also important for many tasks, e.g., named entity recognition [19] and part-of-speech tagging [20]. Bidirectional RNNs were invented to meet this need [21]. Figure 1b presents the structure of a typical bidirectional RNN. At each time step, the bidirectional RNN computes both the hidden state $\boldsymbol{h_t}$ of the forward sub-RNN and the hidden state $\boldsymbol{g_t}$ of the backward sub-RNN. $\boldsymbol{g_t}$ is computed as follows:

$$\boldsymbol{g_t} = f'(\boldsymbol{g_{t+1}}, \boldsymbol{e_t}) \tag{2}$$

In most cases, the output $\boldsymbol{o_t}$ of the bidirectional RNN is constructed by concatenating $\boldsymbol{h_t}$ and $\boldsymbol{g_t}$.

2.2 Word Embedding

To process words using neural networks, we need first map words into vectors of real numbers. A simple way is to represent words as one-hot vectors. But one-hot vectors do not encode the similarities between words, and usually suffer from the curse of dimensionality. The second way is jointly learning word vectors with a target task, e.g., language modeling [22], machine translation [18] and etc. However, when dealing with the tasks without abundant annotated data, it may be more useful to map words to vectors through the word embedding pre-trained on large corpora.

Plenty of approaches are proposed to represent words as high-dimensional vectors that encode syntactic and semantic regularities between words as the linear relationships between word representations [15], [23]–[25]. Such high-dimensional vectors are called "word embeddings". Skip-gram [25], [26] and GloVe [15] are the most well-known unsupervised methods, both of which aim to learn accurate high-dimensional word embeddings from large unlabeled corpora. Skip-gram learns word embeddings by training a prediction model with the objective of predicting the context of a word given the word itself. GloVe combines global statistics of the training corpus with local context window methods, like skip-gram, to learn accurate word representations. The website of GloVe [27] also publishes some sets of pre-trained word embeddings to accelerate related work.

2.3 Self-Attention

Recently, attention mechanisms are widely used for many tasks, from image classification [28], [29] to neural machine

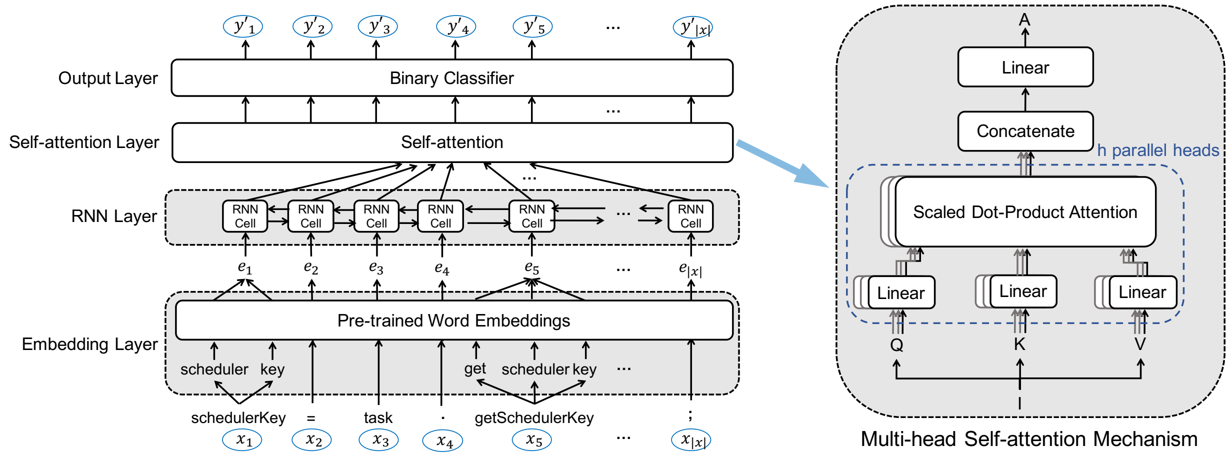

Output Layer
Self-attention Layer
RNN Layer
Embedding Layer
Binary Classifier
Self-attention
RNN Cell
Pre-trained Word Embeddings
scheduler key
get scheduler key
schedulerKey = task . getSchedulerKey ;
y'_1 y'_2 y'_3 y'_4 y'_5 $y'_{|x|}$
e_1 e_2 e_3 e_4 e_5 $e_{|x|}$
x_1 x_2 x_3 x_4 x_5 $x_{|x|}$
A
Linear
Concatenate
h parallel heads
Scaled Dot-Product Attention
Linear
Q K V
I
Multi-head Self-attention Mechanism

Fig. 2: The overall framework of our approach.

translation [30], [31], to help models focus on the important parts of inputs. The inputs of an attention function are a query and a set of key-value pairs, which are all vectors. The output is computed as the weighted sum of the values, and the weight assigned to each value is calculated by a compatibility function of the query and the value's corresponding key. According to how the weights are calculated, there are three popular attention mechanisms, i.e., the additive attention [30], the dot-product attention [32] and the scaled dot-product attention [31]. In this work, we use the scaled dot-product attention, since it is of good performance and computationally cheaper [31].

Given a query q, a set of keys K of dimension d_k and corresponding values V, the scaled dot-product attention first computes the dot products of q with all keys, divides each dot product by $\sqrt{d_k}$ for scaling and obtains the weights on the values through a softmax function. Then, it uses the calculated weights to calculate the weighted sum of the values as the output. In practice, we usually pack a sequence of queries together into a matrix Q, and compute their outputs simultaneously. These outputs are called attention vectors, and are computed as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3}$$

Self-attention is a special case of attention mechanism, of which the queries, keys and values are the same, i.e., $Q = K = V$. When applied in natural language processing, self-attention can model dependencies between words without considering their distances in the input or output sequences, and can capture the syntactic and semantic structure of sentences [31].

3 Approach

In this section, we describe our approach in detail. We first define and formalize the problem. Then we present the overall framework and all the components of our approach. Finally, we describe the details of our implementation and training process.

3.1 Problem Formulation

We want to recommend logging variables to developers during development. This problem can be formulated as follows: given a sequence of program tokens $\boldsymbol{x} = (x_1, x_2, ..., x_{|\boldsymbol{x}|})$ and the set of variables $\boldsymbol{v} = (v_1, v_2, ..., v_{|\boldsymbol{v}|})$ which should be logged, find a function f so that $f(\boldsymbol{x}) = \boldsymbol{v}$. $|\cdot|$ denotes the length of a sequence or size of a set.

We can regard this problem as a multi-label classification problem, i.e., treat each possible logging variable as a label and generate multiple labels for each code snippet. A common practice for solving multi-label classification problem is to first collect possible labels (i.e., variables) from the training set and build a binary classifier for each possible label; then, given a sample (i.e., a code snippet), predict the probability of each label being generated through the corresponding classifier; finally, output the top-k labels with highest probabilities. However, for the task considered in this work, the alternative labels of each sample are different, since a logging statement can only print the variables which can be accessed, e.g., global variables, class properties and local variables defined before it, and for different logging statements, the set of accessible variables are often different. Moreover, if we only build classifiers for the variables that appear in the training set, we can not suggest the logging variables which only appear in the test set. Therefore, it is not suitable to treat this problem as a traditional multi-label classification problem.

In this work, given a code snippet, we only try to answer which variables in the code snippet (i.e., a sequence of program tokens) should be logged. We simplify the original problem as an optimization problem: given a sequence of program tokens $\boldsymbol{x} = (x_1, x_2, ..., x_{|\boldsymbol{x}|})$ and the labels of whether each token should be logged $\boldsymbol{y} = (y_1, y_2, ..., y_{|\boldsymbol{x}|})$, find a function f so that $f(\boldsymbol{x}) = (y'_1, y'_2, ..., y'_{|\boldsymbol{x}|})$ and the loss between $f(\boldsymbol{x})$ and $\boldsymbol{y}$ is minimized. $y_i \in \{0, 1\}$. $y'_i \in [0, 1]$, and denotes the probability of the i_{th} token being logged.

3.2 Overall Framework

Figure 2 presents the overall framework of our approach. It contains four layers: the embedding layer, the RNN layer, the self-attention layer and the output layer. Given a sequence of program tokens, our approach first maps each token into a word vector through the embedding layer. The word vectors are expected to capture the semantic information of the corresponding tokens. Next, the RNN layer learns to represent each word vector in a new vector space. Then, for each token, the self-attention layer is used to refine its word representation output by the RNN layer by focusing on the contextual tokens that are important for representing this token. The RNN layer and the self-attention layer calculate and refine each token's representation by considering all the tokens in the sequence. They are expected to embed the syntactic and semantic information of the whole sequence in the learned word representation of each token. Finally, our approach leverages a unified binary classifier to predict the probability of each token being logged according to its representation learned by the previous layers. In a nutshell, our approach tries to learn the proper representation of each token through a multi-layer neural network so that a unified binary classifier can have sufficient syntactic and semantic information to make accurate decisions.

3.3 Embedding Layer

To process program tokens using a neural network, we first map each token into its distributed representation, which the embedding layer is responsible for. In the NLP community, a common practice is to first build a vocabulary from the training set, and then jointly learn the word embedding of each word in the vocabulary with the target task. But if a token only appears in the test set, this method will not be able to represent it properly. This phenomenon is referred to as the "out of vocabulary" issue.

When writing programs, developers often name new identifiers using *compound words*, which refers to the tokens that are constructed by concatenating multiple simple tokens following camel casing (e.g., *inputBuffer*) or snake casing (e.g., input_buffer). Compared to new natural language sentences, it is more possible for a new code snippet to contain out-of-vocabulary words. For example, it is common for a developer to declare some new variables in a new code snippet. Therefore, when processing code snippets, the out-of-vocabulary problem is worse.

Fortunately, we notice that in high-quality software projects, developers usually concatenate multiple English words and/or their abbreviations following camel or snake casing to name identifiers. So, given a *compound word*, we can construct its word embedding by combining the word embeddings of its constituent simple tokens. Following this observation, we propose a new method to map program tokens to word embeddings. Our method requires a set of word embeddings pre-trained on natural language corpora. In this work, we use the GloVe pre-trained word embeddings due to its good performance and ease of access [15]. The word embeddings of *non-compound words* are directly found from the pre-trained word embeddings. For each *compound word*, our method first splits it into several simple tokens, and then it finds the word embeddings of these simple tokens from the pre-trained word embeddings. Finally, the average vector of these simple tokens' word embeddings is regarded as the word embedding of the *compound word*.

```
1  public static FiCaSchedulerNode getMockNode(String host,
       String rack,int port,int memory,int vcores){
2    NodeId nodeId=NodeId.newInstance(host,port);
3    RMNode rmNode=mock(RMNode.class);
4    when(rmNode.getNodeID()).thenReturn(nodeId);
5    when(rmNode.getTotalCapability()).thenReturn(Resources.
         createResource(memory,vcores));
6    when(rmNode.getNodeAddress()).thenReturn(host + ":" +
         port);
7    when(rmNode.getHostName()).thenReturn(host);
8    when(rmNode.getRackName()).thenReturn(rack);
9    FiCaSchedulerNode node=spy(new FiCaSchedulerNode(rmNode,
         false));
10   LOG.info("node = " + host + " avail="+ node.
         getUnallocatedResource());
11   when(node.getNodeID()).thenReturn(nodeId);
12   return node;
13 }
```

Fig. 3: A method with a logging statement

While the GloVe pre-trained word embeddings include the word embeddings of numerous common abbreviations, e.g., "cnt" for "count" and "ldap" for "Lightweight Directory Access Protocol", there may also exist a few uncommon abbreviations which appear in the source code but are not included by the GloVe word embeddings. We simply regard such abbreviations as "unknown", and map them to the zero vector.

3.4 RNN Layer

The RNN layer is used to represent the output of the embedding layer in a new vector space for our task. Word embeddings can capture the syntactic and semantic relationships between tokens, while they do not contain the syntactic, semantic and structural information of a specific code snippet. Therefore, our approach leverage an RNN to encode such information into our word representations.

This layer adopts a bidirectional RNN. Given a program token, the bidirectional RNN can integrate the information of the whole code snippet, instead of only the tokens before it, to represent it. There are many variants of RNN, e.g., LSTM and GRU. In this work, we choose GRU since it performs similarly to LSTM but is computationally cheaper [18].

As described in Section 2.1, given a word embedding $\boldsymbol{e_t}$ of a program token $\boldsymbol{w_t}$, the bidirectional GRU will compute its hidden states of the forward sub-GRU and backward sub-GRU, i.e., $\boldsymbol{h_t}$ and $\boldsymbol{g_t}$ respectively, using Equation 1 and 2. The output $\boldsymbol{o_t}$ at step t is then constructed by concatenating $\boldsymbol{h_t}$ and $\boldsymbol{g_t}$:

$$\boldsymbol{o_t} = (\boldsymbol{h_t}, \boldsymbol{g_t})$$

3.5 Self-attention Layer

Given a code snippet, it is usually the case that a logging variable is related to several but not all lines of code preceding its corresponding logging statement. For example, Figure 3 is a method extracted from the Hadoop project. Line 10 is a logging method with two logging variables, i.e., *host* and *node*. We can see that *host* only appears in line 1, 2, 6 and 7; *node* only appears in line 9. Learning to pay more attention to the related code may be helpful for accurate variable suggestions. Following this observation, our approach leverages a self-attention layer to learn to focus on important parts of the input and refine the word representations learned by the RNN layer.

The self-attention layer adopts the multi-head attention mechanism [31] to calculate the attention vectors, which is depicted in Figure 2. The input, $I \in \mathbb{R}^{|\boldsymbol{x}| \times d}$, is a sequence of vectors, which are also regarded as queries Q, keys K and values V in self-attention. The multi-head attention mechanism employs h heads to process the inputs in parallel in order to focus on different channels of the input vectors. For each head, queries, keys and values are first linearly projected to d_k, d_k and d_v dimensional vectors with different, learned linear projections, respectively. Then, we perform the scaled dot-product attention function described in Section 2.3 with these projected vectors as input, and obtain a sequence of d_v dimensional vectors or a matrix output by this head. Formally, the output matrix M_i of the i_{th} head is calculated as follows:

$$M_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

where the $Attention$ function is defined in Equation 3, the $W_i^Q \in \mathbb{R}^{d \times d_k}$, $W_i^K \in \mathbb{R}^{d \times d_k}$ and $W_i^V \in \mathbb{R}^{d \times d_v}$ are the parameter matrices of the linear projections. Finally, the output matrices of all the heads are concatenated into one matrix and are linearly projected once again. The corresponding formulas are shown below:

$$M = Concat(M_1, M_2, ..., M_h)$$

$$A = MW^A$$

where the $W^A \in \mathbb{R}^{d_v h \times d_a}$ is the parameter matrix of the last projection.

The self-attention layer learns to capture the syntactic and semantic information of a code snippet based on its ability to learn dependencies between code tokens. A key factor affecting such ability is the length of the path that the signal of one token has to traverse forward or backward to meet the signal of the other token in the neural network given any pair of tokens [33]. To learn the dependency between two tokens with k tokens between them, the path length in the RNN layer is k since a token is only connected to its previous token and following token and the signal is passed token by token. However, for the self-attention layer, each token is directly connected to all tokens in the code snippet through dot product operations, i.e., QK^T in Equation 3, hence the path length is always 1. This makes the self-attention layer more efficient than the RNN layer for learning dependencies, especially long-range dependencies, between words [31].

3.6 Output Layer

The previous layers learn to properly represent each program token into a feature vector. With such vectors as input, the output layer leverages a unified binary classifier to predict whether each token should be logged or not. In this work, we use the sigmoid classifier, which is commonly used in neural network architectures. Given the input matrix A, this layer calculates the probability vector $\boldsymbol{y'}$ as follows:

$$\boldsymbol{y'} = sigmoid(AW + \boldsymbol{b})$$

where $W \in \mathbb{R}^{d_a}$ and $\boldsymbol{b} \in \mathbb{R}^{|\boldsymbol{x}|}$ are the weights and biases that need to be learned. The i_{th} element of $\boldsymbol{y'}$, i.e., y'_i is the probability of the i_{th} token x_i being logged.

3.7 Implementation and Training

We implement our framework using PyTorch [34]. The set of word embeddings used by the embedding layer is pre-trained using GloVe on the Wikipedia 2014 and Gigaword 5 corpora [27], and their dimension is 100. This set of word embeddings is powerful enough for our framework to perform well. We do not update the embedding layer while training.

The other three layers are jointly trained and updated. The dimension of the hidden states in the RNN layer is set to 128. In the self-attention layer, we employ $h = 4$ parallel heads, d_k and d_v are set to 64, and d_a is set to $64 * 4 = 256$. We use the Binary Cross Entropy (BCE) as the loss function. Given the label vector $\boldsymbol{y}$ of the input, the BCE function first calculates the cross entropy of each program token, as shown in Equation 4. Then the average of the cross entropy of all program tokens is regarded as the loss, as shown in Equation 5.

$$l_i = -[y_i \cdot log(y'_i) + (1 - y_i) \cdot log(1 - y'_i)] \tag{4}$$

$$loss = \frac{1}{|\boldsymbol{x}|} \sum_{i=1}^{|\boldsymbol{x}|} l_i \tag{5}$$

We use Adam [35] as the optimizer, and the mini-batch size is set to be 80. We train our model for a maximum of 200 epochs, and measure the quality of the trained model after each epoch by calculating the MAP (Mean Average Precision) score of the model on a held out validation set. After training, the model that achieves the highest MAP score is picked as the final model and is tested on the test set.

4 EXPERIMENT SETUP

This section describes our studied projects and the procedures that we followed to extract and preprocess the data.

4.1 Studied Projects

We evaluate our framework on 9 open source Java projects of Apache Foundations, i.e., ActiveMQ, Camel, Cassandra, CloudStack, DirectoryServer, Hadoop, HBase, Hive and Zookeeper. The reasons for choosing these projects are as follows: First, they are all large, mature and successful projects that have been developed for years. Second, they are from different domains, which ensures that our approach is not limited to a specific domain. ActiveMQ is a powerful messaging and integration patterns server, Camel is a rule-based routing and mediation framework, Cassandra is a distributed NoSQL database management system, CloudStack is an Infrastructure as a Service (IaaS) cloud computing platform, DirectoryServer is an embeddable directory server, Hadoop is a scalable, distributed computing framework, HBase is a distributed, non-relational database, which provides big data store for Hadoop, Hive is a data warehouse software, Zookeeper is a centralized service

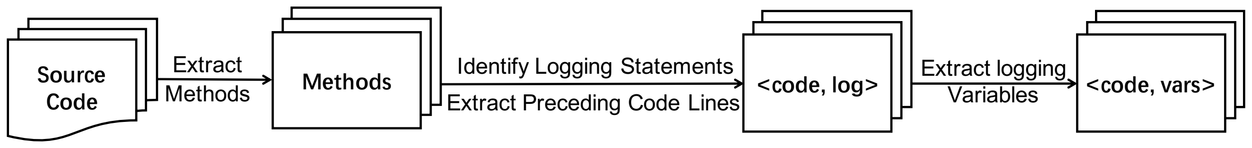

Source Code
Extract Methods
Methods
Identify Logging Statements
Extract Preceding Code Lines
<code, log>
Extract logging Variables
<code, vars>

Fig. 4: The overview of our data extraction process.

TABLE 1: Statistics of the studied systems

Projects	Release	LOC	#Methods	$\#\langle code, log\rangle$	$\#\langle C, V\rangle$	$\#\langle C, V\rangle_{len}$	$\#\langle C, V\rangle_{var}$	$\#\langle C, L, V\rangle$	Avg. #vars
ActiveMQ	*5.15.5*	658K	42.2K	7.0K	5.4K	62	517	4.9K	1.4
Camel	*2.22.0*	1936K	128.1K	10.9K	8.7K	179	569	7.9K	1.4
Cassandra	*3.11.3*	572K	34.5K	1.6K	1.3K	58	84	1.1K	1.5
CloudStack	*4.11.1.0*	895K	55.9K	12.5K	10.4K	762	248	9.4K	1.4
DirectoryServer	*2.0.0-M24*	406K	12.7K	2.1K	1.6K	27	163	1.4K	1.3
Hadoop	*3.0.3*	2101K	115.2K	14.0K	11.2K	321	649	10.2K	1.5
HBase	*2.1.0*	969K	55.2K	8.0K	6.4K	246	348	5.8K	1.5
Hive	*3.1.0*	1728K	104.8K	7.7K	6.5K	265	320	5.9K	1.4
Zookeeper	*3.4.13*	105K	6.2K	1.6K	1.3K	34	89	1.1K	1.3

*$\#\langle C, V\rangle$ refers to the number of $\langle code\ tokens, vars\rangle$ pairs, $\#\langle C, V\rangle_{len}$ is the number of the $\langle code\ tokens, vars\rangle$ pairs filtered by the *length filter* and $\#\langle C, V\rangle_{var}$ is the number of the $\langle code\ tokens, vars\rangle$ pairs filtered by the *var filter*. $\#\langle C, L, V\rangle$ refers to the number of $\langle code\ tokens, labels, vars\rangle$ triples, and the number of filtered $\langle code\ tokens, vars\rangle$ pairs is equal to it. **Avg. #vars** refers to the average number of logging variables in $\langle code\ tokens, labels, vars\rangle$ triples.

which can provide distributed configuration and synchronization services for large distributed systems. In addition, these projects extensively use standard Java logging libraries, e.g., Log4j [36], SLF4J [37] and Apache Commons Logging [38], to perform logging. The uniform formats of these libraries' APIs (e.g., *logger.info(message)*) can help us extract logging statements accurately. Table 1 presents some statistics of our studied projects.

4.2 Data Extraction

This work aims to suggest logging variables according to the contextual code snippet when a developer is adding a logging statement. In this scenario, the code succeeding such logging statement may not exist. Therefore, our approach takes the k lines of code preceding the logging statement as input. In this work, we set k to 15 by default. In addition, we also need to extract the logging variables of each logging statement for training and evaluation.

Figure 4 presents the overview of our data extraction process. This process aims to extract the $\langle code, vars\rangle$ pairs from the source code of each project to build its dataset, where $code$ refers to the k lines of code preceding a logging statement, and $vars$ denotes the variables printed by such logging statement. For each project, first, we download its source code from its official website or git repository and extract all the Java methods from the source code using the Eclipse Java development tools (JDT) [39]. We also delete the comments and the Javadoc in each method. Then, since each project extensively follows the uniform formats defined by the standard Java logging libraries to write logging statements, we construct a set of regular expressions to automatically identify the logging statements in each extracted method. For each identified logging statement, we record it with the k lines of code preceding it to form a $\langle code, log\rangle$ pair where log refers to the logging statement. A code line with only one or more right curly brackets, i.e., $\}$, is not counted as a line. If there are less than k lines of code from the method declaration (inclusive) to the logging statement (exclusive), we record all of them as the $code$. The $\langle code, log\rangle$ pairs which contain non-ASCII characters are also ignored. In addition, since developers may write multiple logging statements in one Java method, the $code$ in a $\langle code, log\rangle$ pair may contain some logging statements preceding the log. We keep such logging statements in the $code$ for they also provide contextual information. Finally, we leverage JDT again with a sequence of heuristic rules to extract variables from each logging statement, delete the $\langle code, log\rangle$ pairs with no logging variable, and convert the remaining $\langle code, log\rangle$ pairs to $\langle code, vars\rangle$ pairs. The number of extracted $\langle code, log\rangle$ pairs and $\langle code, vars\rangle$ pairs are shown in Table 1.

4.3 Variable Extraction

Given a logging statement, we first pick up all of its arguments through JDT. Then for each argument, we apply the following 5 rules to record the logging variables:

1) **RULE 1**: If the argument is a constant value (e.g., *"End position is"*), we ignore it.
2) **RULE 2**: If the argument is a simple identifier, e.g., *requestId*, we directly record it as a variable.
3) **RULE 3**: If the argument is a method invocation without arguments (e.g., *request.toString()*), or a field access (e.g., *this.result*), we record the first identifier that is not *this* (i.e., ThisExpression in JDT) as a variable.
4) **RULE 4**: If the argument is a method invocation with arguments (e.g., *Integer.toString(position)*), we iteratively extract variables from its arguments using the 5 rules. If we do not record any variable from its arguments (e.g., all the arguments are constant values), we will treat it as a method invocation without arguments, and apply RULE 3 to it. Specially, if the argument is a *get* method invocation (e.g., *list.get(i)*), we will extract variables from both its

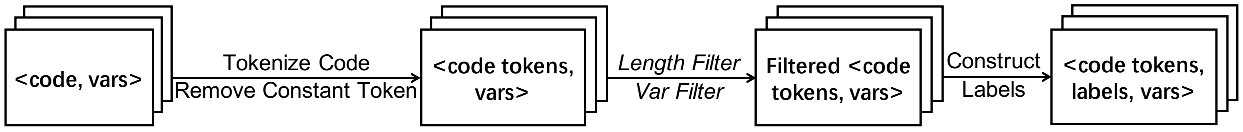


Fig. 5: Our data preprocessing procedure

```
log.info("End position is: ", Integer.toString(start +
     request.offset + request.length()));
```

Fig. 6: A logging statement

expression (e.g., *list*) and its arguments according to RULE 1-4.

5) **RULE 5**: If the argument fails to meet RULE 1-4 (e.g., it is an infix expression (*"start at" + length*)), we iteratively apply the 5 rules to each of its AST children to find variables. If this argument is a leaf node, e.g., a this expression (i.e., *this*), we ignore it.

Finally, all the recorded variables are merged together as the logging variables of this logging statement, and duplicate variables are removed.

For example, if we want to extract the logging variables of the logging statement shown in Figure 6, we first pick up its two arguments and then process each argument using the 5 rules. The first argument is *"End position is"*, which is a string literal, so we simply apply RULE 1 and ignore it. The second argument is *Integer.toString(start + request.offset + request.length())* which is a method invocation with an argument. Therefore, we apply RULE 4 which guides us to recursively process its argument. Since its argument is an infix expression, RULE 5 is used, which asks us to iteratively process the three AST children, i.e., *start*, *request.offset* and *request.length()*. According to RULE 2 and RULE 3 respectively, we extract *start* and *request* from the first two children as logging variables. The last children is a method invocation without arguments, thus we also extract *request* from it based on RULE 3. At last, we obtain a list of three variables, which is $\langle start, request, request \rangle$. After removing duplicate variables, the final set of variables, i.e., $vars$, is $\langle start, request \rangle$.

The intuitions of RULE 1 and 2 are obvious. We design RULE 3-5 based on the following reasons: For compound identifiers described in RULE 3, the first simple identifier is usually the key object, while the remaining parts are a method or a property of the key object. For example, for *request.offset*, *request* is the main object of this expression, and *offset* is just a property of *request*. If we record *offset* as a variable, developers may be confused since there may be a local identifier named *offset* too. It is also not practical to record the whole compound identifier as a variable, because the identifier may be very complicated or not able to be inferred from the contextual code snippet. For example, for the statement in Figure 6, *request* must be an accessible object, but we may not know that *request* has a property named *offset* according to the contextual code snippet. As for RULE 4-5, an argument of a logging statement can be a complicated expression, it is difficult to design rules for all possible situations. Therefore, in RULE 4 and RULE 5, we

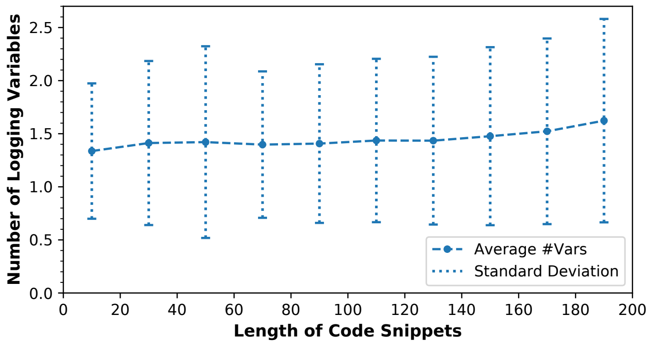


Fig. 7: The mean and the standard deviation of the number of logging variables in $\langle code\ tokens, labels, vars \rangle$ triples with various code length. For example, the leftmost point and error bar present that the average number of logging variables in $\langle code\ tokens, labels, vars \rangle$ triples with code length of 0 to 20 is about 1.3, and the corresponding standard deviation is about 0.64.

recursively process the descendants of those complicated expressions, and extract variables from the descendants which meet RULE 1-3.

4.4 Data Preprocessing

After data extraction, we obtain a set of $\langle code, vars \rangle$ pairs from each project. To meet the requirements of our approach, we need to further preprocess these pairs. Figure 5 presents the procedure of our data preprocessing. For each pair, we tokenize its $code$, delete all string and number literals from the $code\ tokens$, and mark all the identifiers (i.e., TokenNameIdentifier in JDT) in the $code\ tokens$ using JDT. Then, we remove the pairs with code lengths of more than 200 (i.e., *length filter*) and the pairs of which none of the $vars$ appears in corresponding $code\ tokens$ (i.e., *var filter*). The numbers of the removed pairs are presented in Table 1. We limit the code length to 200 for better training the RNN layer in our framework. Finally, given a pair, by comparing each $code\ token$ with each variable in the $vars$, we can obtain the label sequence, whose length is equal to the length of the $code\ tokens$. Each label is either 1 or 0, representing whether the corresponding $code\ token$ is logged or not. We keep the $vars$ in our preprocessed dataset for computations of our evaluation metrics (see Section 5.1).

For each project, we obtain a set of $\langle code\ tokens, labels, vars \rangle$ triples. The number of the triples and the average number of $vars$ for the triples in each project are presented in Table 1. The average number of $vars$ is between 1.3 and 1.5. Figure 7 shows the mean and the standard deviation of the number of $vars$ for the triples in all projects with various code length. We can see that for the triples with different code length, the average number of $vars$ is between 1.3 to 1.6, and the corresponding standard deviation is between 0.64 and 0.96.

To build our dataset, for each project, we randomly select 10% of them for testing, 10% of them for validation and the remaining 80% for training. As described in Section 3.7, our approach uses the Binary Cross Entropy (BCE) as the loss function, which regards the average of the cross entropy of all $code\ tokens$, including tokens that are not identifiers, as the loss of each iteration. However, since only identifiers can be logged, our approach filters out tokens that are not identifiers and only outputs a ranked list of identifiers while inferring.

5 Evaluation

In this section, we evaluate the performance of our approach. We first describe the evaluation metrics and the baselines, and then present our research questions and corresponding experiment results.

5.1 Evaluation Metrics

With a code snippet as input, our approach predicts the probability of each identifier being logged in the corresponding logging statement. It will output multiple probabilities for an identifier if the identifier appears at multiple positions in the code snippet. Our approach ranks all distinct identifiers according to their highest probabilities. The higher an identifier's probability is, the higher its rank is. To evaluate our approach, we calculate top-k accuracy, Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) scores based on such ranked lists.

5.1.1 *Top-k Accuracy*

Top-k accuracy is the percentage of code snippets of which at least one logging variable is in the set of the top-k variables returned by an approach. Given a code snippet, if at least one of the top-k variables recommended by an approach is actually logged in the corresponding logging statement, we consider the recommendation to be successful and set $success(code, top\text{-}k)$ to 1; else we regard the recommendation to be unsuccessful and set $success(code, top\text{-}k)$ to 0. Given a set of code snippets C, its top-k accuracy Accuracy@k is calculated as:

$$Accuracy@k = \frac{\sum_{code \in C} success(code, top\text{-}k)}{|C|}$$

where $|C|$ refers to the number of code snippets in C.

The higher the top-k accuracy is, the better a logging variable recommendation approach performs. As shown in Table 1, for each studied project, the average number of logging variables in each sample is between 1.3 and 1.5. So in this work, we set k to 1 and 2.

5.1.2 *Mean Reciprocal Rank (MRR)*

MRR is a popular metric used to evaluate an information retrieval technique [40]. Given a query (in our case: a code snippet), its reciprocal rank is the multiplicative inverse of the rank of the first correct document (in our case: a variable) in a rank list produced by a ranking technique (in our case: a logging variable recommendation approach). Given a set of code snippets C, its MRR score is the average of the reciprocal ranks of all code snippets in C:

$$MRR(C) = \frac{1}{|C|} \sum_{code \in C} \frac{1}{rank(code)}$$

where $rank(code)$ is the rank of the first correct logging variable returned by an approach for $code$.

5.1.3 *Mean Average Precision (MAP)*

MAP is a single-figure measure of quality, which has been shown to have good discrimination and stability to evaluate ranking techniques [41]. Different from MRR which only considers the first correct result, MAP considers all correct results in each ranked list. For a code snippet, its average precision is defined as the mean of the precision values obtained for different sets of top-k variables that were retrieved before every logged variable is retrieved, which is computed as:

$$AP(code) = \frac{\sum_{k=1}^{n} P(k) \times Rel(k)}{|vars|}$$

where n is the number of variables in the ranked list, $Rel(k)$ indicates whether the variable at position k is actually logged or not, and $P(k)$ is the precision at the given cut-off position k and is computed as:

$$P(k) = \frac{|vars\ in\ top\text{-}k|}{k}$$

where $top\text{-}k$ denotes the set of top-k variables returned by our approach, and $|vars\ in\ top\text{-}k|$ refers to the number of the logging variables that appear in the top-k variables.

The MAP of a set of code snippets C is then the mean of the average precision scores for all code snippets in C:

$$MAP(C) = \frac{\sum_{code \in C} AP(code)}{|C|}$$

Developers may log several variables in a logging statement. The MAP value can measure the average performance of our approach to suggest all logged variables.

5.2 Baselines

We use Random Guess (RG) and an information-retrieval (IR)-based method as baselines.

5.2.1 *Random Guess*

Given a sequence of $code\ tokens$, the random guess method collects all of its distinct identifiers, shuffles these identifiers and outputs the shuffled identifiers as a ranked list.

5.2.2 *IR-based method*

Given a sequence of $code\ tokens$ in the test set, the IR-based method first finds the most similar $code\ token$ sequence, i.e., its nearest neighbor, from the training set, and then treats the $vars$ of the nearest neighbor as output. According to how to find the nearest neighbors, we consider four variants of the IR-based method, which we refer to as IR-comp, IR-flat, IR-mix and IR-WE.

The first three variants all represent each sequence of $code\ tokens$ in the form of "bag-of-words", and measure the similarity between two sequences by calculating the cosine

TABLE 2: The Effectiveness Level of Cliff's Delta

Cliff's delta ($\|\delta\|$)	Effectiveness Level
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\|$	Large

similarity between their "bag-of-words" vectors. They differ from each other in the way they cope with the *compound words* in sequences. IR-comp treats the *compound words* as they are. In contrast, IR-flat splits each *compound word* into several simple tokens before mapping a sequence to a "bag-of-words" vector. IR-mix keeps both the *compound words* and their constituent simple tokens to form the "bag-of-words" vectors.

The last variant IR-WE leverages word embeddings to calculate the similarities between *code token* sequences and find the nearest neighbors. For each *code token*, IR-WE leverages our word embedding method to construct its word vector. Then, IR-WE uses the method proposed in [42] to calculate the similarity between two *code token* sequences based on the word vectors. The nearest neighbors are then selected according to such similarities.

5.3 RQ1: The Effectiveness of Our Approach

Motivation. We want to investigate how effective our approach is and how much performance improvement our approach can achieve over the baselines.

Approach. We apply our approach and the baseline methods (i.e., random guess and the IR-based methods) on the dataset of each project, and compare their performance in terms of Accuracy@1, Accuracy@2, MRR and MAP. We also conduct Wilcoxon signed-rank tests [43] at the confidence level of 95% to check whether the performance differences between our approach and the baseline approaches are significant. For each approach, we collect 9 scores (one for each project) considering every evaluation metric. For each pair of competing approaches, the Wilcoxon signed-rank test is conducted considering the 9 scores for each of the approaches. Moreover, we use an effect size measure named Cliff's delta (δ) to analyze the magnitude of the observed differences. According to the guidelines in [44], the Cliff's Delta values are interpreted based on Table 2.

Results. The Accuracy@1, Accuracy@2, MRR and MAP for our approach and the baseline methods are shown in Table 3 and Table 4. For each project, our approach outperforms all the baselines in terms of the four metrics by large margins. On average, the performance improvements of our approach over the baselines are all no less than 43% in terms of all the metrics.

Table 5 presents the p-values and Cliff's delta (δ) when we compare our approach with the baseline approaches in terms of each metric. We consider our approach to be statistically significantly better than the baseline approaches at the confidence level of 95% if the corresponding p-value is less than 0.05. We can see that all the p-values are less than 0.05 and all the δ values are greater than 0.474, which means our approach significantly improves over the baseline approaches with large effect size on all the metrics.

Compared to random guess and the IR-based methods, our approach tries to learn what information is important for an identifier to be logged, and encodes such information into a vector of real numbers to represent the identifier. The good performance of our approach indicates that our approach is effective in representing identifiers to suggest proper variables to be logged.

5.4 RQ2: The Effects of Main Components

Motivation. To model code snippets, we first propose a novel method to map *code tokens* into word vectors. Then, a bidirectional RNN is used to learn to incorporate the information of a code snippet into the representations of its *code tokens*. We also leverage the self-attention mechanism to focus on important parts of a code snippet and refine the representations learned by the RNN layer. We want to investigate the impacts of these components on the performance of our approach.

Approach. To understand the importance of these components, we compare our approach with four of its incomplete variants:

1) **NE+RNN+Attn** (for short, **NRA**) makes use of the RNN layer and the self-attention layer, but replaces our embedding (OE) layer with a normal embedding (NE) layer. The normal embedding layer first builds a vocabulary from the training set. The word embeddings of the tokens in the vocabulary are then jointly learned with the target task.
2) **OE+Attn** removes the RNN layer from our approach.
3) **OE+RNN** removes the self-attention layer from our approach.
4) **OE+Uni-RNN+Attn** (for short, **OURA**) uses a unidirectional instead of bidirectional RNN in the RNN layer.

By comparing our approach with NRA, we can understand the effect of our embedding method. Comparing our approach with OE+Attn and OE+RNN helps us measure the performance improvements achieved due to the incorporation of the RNN layer and the self-attention layer, respectively. The performance differences between our approach and OURA can demonstrate how important the information of the *code tokens* after a token is for learning this token's representation. Accuracy@1, Accuracy@2, MRR and MAP are used to evaluate our approach and the four variants. The Wilcoxon signed-rank test is conducted and the Cliff's delta (δ) is computed.

Results. Table 6 and Table 7 present the effectiveness of the four variants in terms of Accuracy@1, Accuracy@1, MRR and MAP. We can see that on average, our approach outperforms the four variants on each metric. For the nine studied projects, our approach performs better than the variants in terms of Accuracy@1 and MAP, is more or as effective in terms of MRR, and improves the variants in most cases in terms of Accuracy@2.

Specifically, compared to the variant which uses the normal embedding layer, i.e., NRA, the average performance

TABLE 3: Comparisons of our approach (Ours) with each baseline in terms of Accuracy@1 and Accuracy@2

Projects	**Accuracy@1**						**Accuracy@2**					
	RG	**IR-comp**	**IR-flat**	**IR-mix**	**IR-WE**	**Ours**	**RG**	**IR-comp**	**IR-flat**	**IR-mix**	**IR-WE**	**Ours**
ActiveMQ	0.165	0.557	0.565	0.561	0.561	**0.860**	0.287	0.581	0.584	0.581	0.592	**0.936**
Camel	0.132	0.634	0.626	0.623	0.621	**0.866**	0.263	0.678	0.666	0.666	0.670	**0.941**
Cassandra	0.140	0.553	0.561	0.561	0.570	**0.789**	0.281	0.596	0.605	0.605	0.623	**0.904**
CloudStack	0.100	0.595	0.571	0.577	0.617	**0.830**	0.198	0.639	0.617	0.624	0.658	**0.919**
DirectoryServer	0.103	0.566	0.596	0.581	0.625	**0.846**	0.228	0.640	0.654	0.640	0.706	**0.926**
Hadoop	0.121	0.471	0.463	0.462	0.464	**0.786**	0.234	0.516	0.502	0.501	0.512	**0.898**
HBase	0.133	0.515	0.494	0.496	0.527	**0.793**	0.247	0.572	0.547	0.553	0.585	**0.895**
Hive	0.135	0.497	0.496	0.501	0.520	**0.796**	0.213	0.547	0.554	0.557	0.574	**0.893**
Zookeeper	0.186	0.584	0.575	0.575	0.637	**0.788**	0.319	0.593	0.593	0.593	0.655	**0.903**
Average	*0.135*	*0.552*	*0.550*	*0.549*	*0.571*	***0.817***	*0.252*	*0.596*	*0.591*	*0.591*	*0.619*	***0.913***
Improved	*505.3%*	*47.9%*	*48.7%*	*49.0%*	*43.0%*	-	*261.9%*	*53.2%*	*54.4%*	*54.4%*	*47.4%*	-

*RG refers to the random guess method.

TABLE 4: Comparisons of our approach (Ours) with each baseline in terms of MRR and MAP

Projects	**MRR**						**MAP**					
	RG	**IR-comp**	**IR-flat**	**IR-mix**	**IR-WE**	**Ours**	**RG**	**IR-comp**	**IR-flat**	**IR-mix**	**IR-WE**	**Ours**
ActiveMQ	0.346	0.570	0.576	0.573	0.578	**0.912**	0.312	0.539	0.545	0.539	0.545	**0.864**
Camel	0.321	0.657	0.648	0.646	0.647	**0.919**	0.291	0.607	0.601	0.600	0.600	**0.879**
Cassandra	0.338	0.583	0.591	0.591	0.605	**0.870**	0.311	0.531	0.540	0.540	0.556	**0.829**
CloudStack	0.272	0.618	0.595	0.602	0.638	**0.895**	0.251	0.576	0.556	0.561	0.596	**0.862**
DirectoryServer	0.291	0.603	0.625	0.610	0.665	**0.906**	0.269	0.577	0.596	0.581	0.631	**0.867**
Hadoop	0.302	0.494	0.484	0.484	0.490	**0.867**	0.276	0.443	0.432	0.432	0.437	**0.827**
HBase	0.308	0.548	0.525	0.528	0.558	**0.866**	0.275	0.490	0.465	0.470	0.504	**0.821**
Hive	0.289	0.525	0.527	0.531	0.549	**0.870**	0.256	0.488	0.489	0.490	0.509	**0.830**
Zookeeper	0.360	0.594	0.590	0.590	0.649	**0.868**	0.339	0.562	0.555	0.555	0.611	**0.840**
Average	*0.314*	*0.577*	*0.573*	*0.573*	*0.598*	***0.886***	*0.287*	*0.535*	*0.531*	*0.530*	*0.554*	***0.847***
Improved	*182.0%*	*53.6%*	*54.5%*	*54.7%*	*48.2%*	-	*195.3%*	*58.3%*	*59.4%*	*59.8%*	*52.7%*	-

*RG refers to the random guess method.

TABLE 5: P-value and Cliff's delta (δ) for our approach compared with each baseline

Ours vs. Baseline	**Accuracy@1**		**Accuracy@2**		**MRR**		**MAP**	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ
Ours vs. RG	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)
Ours vs. IR-comp	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)
Ours vs. IR-flat	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)
Ours vs. IR-mix	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)
Ours vs. IR-WE	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)

*RG refers to the random guess method, and Lar refers to large effect size.

TABLE 6: Effectiveness of each incomplete variant of our approach in terms of Accuracy@1 and Accuracy@2

Projects	**Accuracy@1**					**Accuracy@2**				
	NRA	**OE+Attn**	**OE+RNN**	**OURA**	**Ours**	**NRA**	**OE+Attn**	**OE+RNN**	**OURA**	**Ours**
ActiveMQ	0.816	0.606	0.829	0.808	**0.860**	0.913	0.751	0.926	0.903	**0.936**
Camel	0.844	0.673	0.843	0.824	**0.866**	0.929	0.792	0.928	0.922	**0.941**
Cassandra	0.772	0.596	0.754	0.684	**0.789**	0.860	0.711	0.877	0.868	**0.904**
CloudStack	0.817	0.652	0.827	0.794	**0.830**	0.909	0.791	**0.920**	0.902	0.919
DirectoryServer	0.809	0.699	0.824	0.809	**0.846**	**0.934**	0.831	0.912	0.912	0.926
Hadoop	0.763	0.539	0.765	0.707	**0.786**	0.880	0.694	0.885	0.840	**0.898**
HBase	0.777	0.558	0.767	0.732	**0.793**	0.884	0.712	0.869	0.846	**0.895**
Hive	0.765	0.608	0.748	0.736	**0.796**	0.879	0.748	0.867	0.859	**0.893**
Zookeeper	0.699	0.611	0.726	0.743	**0.788**	0.823	0.752	0.858	0.841	**0.903**
Average	*0.785*	*0.616*	*0.787*	*0.760*	***0.817***	*0.890*	*0.754*	*0.894*	*0.877*	***0.913***
Improved	*4.1%*	*32.7%*	*3.8%*	*7.6%*	-	*2.5%*	*21.1%*	*2.2%*	*4.1%*	-

TABLE 7: Effectiveness of each incomplete variant of our approach in terms of MRR and MAP

Projects	MRR					MAP				
	NRA	OE+Attn	OE+RNN	OURA	Ours	NRA	OE+Attn	OE+RNN	OURA	Ours
ActiveMQ	0.885	0.728	0.896	0.878	**0.912**	0.838	0.676	0.849	0.833	**0.864**
Camel	0.903	0.773	0.904	0.892	**0.919**	0.860	0.722	0.863	0.848	**0.879**
Cassandra	0.847	0.715	0.847	0.812	**0.870**	0.807	0.685	0.810	0.784	**0.829**
CloudStack	0.884	0.766	**0.895**	0.873	**0.895**	0.850	0.724	0.861	0.838	**0.862**
DirectoryServer	0.887	0.806	0.893	0.885	**0.906**	0.843	0.759	0.849	0.839	**0.867**
Hadoop	0.849	0.674	0.852	0.810	**0.867**	0.809	0.621	0.811	0.767	**0.827**
HBase	0.857	0.685	0.852	0.824	**0.866**	0.813	0.632	0.806	0.781	**0.821**
Hive	0.849	0.725	0.841	0.830	**0.870**	0.810	0.670	0.798	0.788	**0.830**
Zookeeper	0.807	0.726	0.827	0.828	**0.868**	0.781	0.694	0.795	0.794	**0.840**
Average	*0.863*	*0.733*	*0.867*	*0.848*	***0.886***	*0.823*	*0.687*	*0.827*	*0.808*	***0.847***
Improved	*2.6%*	*20.8%*	*2.1%*	*4.5%*	-	*2.8%*	*23.2%*	*2.4%*	*4.8%*	-

TABLE 8: P-value and Cliff's Delta (δ) for our approach compared with the incomplete variants of our approach

Ours vs. Constructed	Accuracy@1		Accuracy@2		MRR		MAP	
	p-value	δ	p-value	δ	p-value	δ	p-value	δ
Ours vs. NRA	0.002	0.48 (Lar)	0.004	0.41 (Med)	0.002	0.48 (Lar)	0.002	0.53 (Lar)
Ours vs. OE+Attn	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)	0.002	1.00 (Lar)
Ours vs. OE+RNN	0.002	0.48 (Lar)	0.006	0.40 (Med)	0.007	0.44 (Med)	0.002	0.48 (Lar)
Ours vs. OURA	0.002	0.53 (Lar)	0.002	0.57 (Lar)	0.005	0.51 (Lar)	0.005	0.58 (Lar)

*Lar and Med refer to large and medium effect size, respectively.

improvement of our approach on each metric is over 2.5%. Among the 9 projects, the largest improvements (over 7.5%) of our approach in terms of the four metrics are all achieved on Zookeeper, which is the project with the minimal number of training samples. These results show that our embedding method can boost the performance of predicting which variables to log, and is helpful for projects with limited numbers of samples. The remarkable performance improvements (more than 20% on average) of our approach over OE+Attn on each metric highlight the importance of the RNN layer for learning the representations of $code\ tokens$. Our approach also outperforms OE+RNN in terms of each metric by over 2.1% on average, which highlights the value of the self-attention layer for refining $code\ token$ representation. In addition, our approach improves OURA by over 4.1% on average, which means the $code\ tokens$ after a token play an important role in learning the token's representation. This result is also intuitive. For example, given a statement "int status = answer.getStatus();", the meaning and the role of "status" should be related to both "int" and the tokens after it, i.e., "answer" and "getStatus".

The p-values and Cliff's delta values for our approach compared with the variants are presented in Table 8. We can see that our approach significantly performs better than the four variants with at least medium effect size in terms of the four metrics.

In summary, our embedding layer, the RNN layer, the self-attention layer and the bidirectional setting of the RNN layer are effective and helpful to boost the effectiveness of our approach.

5.5 RQ3: Cross-Project Prediction

Motivation. Different from mature projects, new projects often lack sufficient training data. Thus it is difficult to directly apply our approach to a new project. This problem may be overcome through the cross-project prediction, which uses the data collected from mature projects (a.k.a. *source projects*) to train a model, and applies the trained model to make suggestions on which variables to log for new projects (a.k.a. *target projects*). We want to investigate whether our approach is still effective for the cross-project setting.

Approach. We conduct a cross-project prediction experiment for each of our studied projects. Each time, our approach is first trained and validated on 8 projects, i.e., *source projects*, and then tested on the remaining project, i.e., the *target project*. Specifically, we merge the $\langle code\ tokens, labels, vars\rangle$ triples of all source projects, randomly select 10% of the triples as the validation set, and regard the rest of the tuples as the training set. The test set contains all the $\langle code\ tokens, labels, vars\rangle$ tuples of the target project. Accuracy@1, Accuracy@2, MRR and MAP are used to measure the effectiveness of cross-project predictions.

Results. Table 9 shows the results of our cross-project predictions. The projects listed in the first column are the target projects of our experiments, and the **Ratio** columns present the ratios of the performance of cross-project predictions to the performance of corresponding within-project predictions on each metric. We can see that the cross-project predictions can achieve 78%-97% of the performance of corresponding within-project predictions in terms of the four metrics. Moreover, in most cases, the **Ratios** are greater than 85%. These results show that our approach can also be

TABLE 9: Comparisons of cross-project predictions with within-project predictions in terms of Accuracy@1, Accuracy@2, MRR and MAP

Target Projects	Accuracy@1			Accuracy@2			MRR			MAP		
	Within	Cross	Ratio	Within	Cross	Ratio	Within	Cross	Ratio	Within	Cross	Ratio
ActiveMQ	0.860	0.687	*79.9%*	0.936	0.826	*88.2%*	0.912	0.799	*87.6%*	0.864	0.750	*86.8%*
Camel	0.866	0.677	*78.2%*	0.941	0.818	*86.9%*	0.919	0.793	*86.3%*	0.879	0.750	*85.3%*
Cassandra	0.789	0.731	*92.6%*	0.904	0.869	*96.1%*	0.870	0.832	*95.6%*	0.829	0.786	*94.8%*
CloudStack	0.830	0.677	*81.6%*	0.919	0.827	*90.0%*	0.895	0.793	*88.6%*	0.862	0.756	*87.7%*
DirectoryServer	0.846	0.677	*80.0%*	0.926	0.859	*92.8%*	0.906	0.803	*88.6%*	0.867	0.759	*87.5%*
Hadoop	0.786	0.713	*90.7%*	0.898	0.855	*95.2%*	0.867	0.819	*94.5%*	0.827	0.768	*92.9%*
HBase	0.793	0.712	*89.8%*	0.895	0.847	*94.6%*	0.866	0.815	*94.1%*	0.821	0.760	*92.6%*
Hive	0.796	0.742	*93.2%*	0.893	0.862	*96.5%*	0.870	0.835	*96.0%*	0.830	0.789	*95.1%*
Zookeeper	0.788	0.755	*95.8%*	0.903	0.867	*96.0%*	0.868	0.843	*97.1%*	0.840	0.806	*96.0%*
Average	*0.817*	*0.708*	*86.9%*	*0.913*	*0.848*	*92.9%*	*0.886*	*0.815*	*92.0%*	*0.847*	*0.769*	*91.0%*

TABLE 10: A test sample in cross-project predictions

Code Snippet:

```
1 protected int doPoll() throws Exception {
2     class ExcludePathFilter implements PathFilter {
3         public boolean accept(Path path) {
4             return !(path.toString().endsWith(config.
                    getOpenedSuffix()) || path.toString().
                    endsWith(config.getReadSuffix()));}}
5     int numMessages = 0;
6     HdfsInfo info = setupHdfs(false);
7     FileStatus fileStatuses[];
8     if (info.getFileSystem().isFile(info.getPath())) {
9         fileStatuses = info.getFileSystem().globStatus(
                info.getPath());
10    } else {
11        Path pattern = info.getPath().suffix("/" + this.
                config.getPattern());
12        fileStatuses = info.getFileSystem().globStatus(
                pattern, new ExcludePathFilter());}
13    for (FileStatus status : fileStatuses) {
14        if (normalFileIsDirectoryNoSuccessFile(status,
                info)) {
15            continue;}
16        if (config.getOwner() != null) {
17            if (!config.getOwner().equals(status.getOwner
                    ())) {
18                if (log.isDebugEnabled()) {
```

Logging Statement:

```
log.debug("Skipping file: as not matching owner: ",
status.getPath().toString(), config.getOwner());
```

Top-3 Variables Predicted by Our Approach:
config, status, path

* This code snippet is extracted from Camel, and our approach is trained using the other eight projects. We present all the code preceding the logging statement in the corresponding Java method, but please note our approach only takes as input 15 lines of code (i.e., line 4 to line 18).

useful when it is applied to new projects by leveraging data from other projects.

Our studied projects are from different domains. Intuitively, different domains have different logging practices, hence it is a little surprising that our approach can obtain such good performance in cross-project predictions. To figure out the reason, we manually inspect the test results of the cross-project predictions. We find that projects in various domains share some common and domain-independent logging patterns, and our approach can learn such patterns and transfer them over different project domains. The example in Table 10 presents some of such patterns. One pattern is that if a logging statement is placed in a *for* loop, it usually needs to record the loop variable, e.g., line 13 and "status" in the code snippet. Another pattern is that the variables used in adjacent conditional statements are more likely to be logged, e.g., line 16, line 17 and "config". Also, the semantics of variable names affects the probability of them being logged. For example, in Table 10, the top-3 variables recommended by our approach, i.e., "config", "status" and "path", are often logged in different contexts. We can see that these logging patterns are common and domain-independent; hence it is possible for our approach to learn and transfer them over various projects and application domains.

5.6 RQ4: The Effects of Different Fitness Measures

Motivation. While training, we use a fitness measure to assess the quality of the trained model after each epoch. After training, the model with the greatest fitness score is chosen as the best trained model. By default, we set the fitness measure as MAP. In this research question, we want to investigate the impact of varying the fitness measure on the effectiveness of our approach.

Approach. For each project, we calculate four different fitness measures, i.e., Accuracy@1, Accuracy@2, MRR and MAP, after each training epoch. After training, we pick the four models with the best Accuracy@1, Accuracy@2, MRR and MAP on the validation set, and refer to them as Ours^{ACC1}, Ours^{ACC2}, Ours^{MRR} and Ours^{MAP}, respectively. The four models are evaluated on the test set in terms of Accuracy@1, Accuracy@2, MRR and MAP.

Results. Table 11 and Table 12 present the Accuracy@1, Accuracy@2, MRR and MAP for our approach using different fitness measures on the studied projects. We can see that the performance differences between the four models are small. On average, Ours^{MRR} and Ours^{MAP} perform slightly better than Ours^{ACC1} and Ours^{ACC2} on each metric. However, for different projects, the best variants in terms of a metric are also different. In summary, the variants of our approach using different fitness measures are almost equally effective on average. In practice we recommend users to choose the fitness measure according to their own dataset.

5.7 RQ5: Time Costs of Our Approach

Motivation. Our approach needs to be trained before being adapted to recommend logging variables to developers. The

TABLE 11: Effectiveness of our approach using different fitness measures on Accuracy@1 and Accuracy@2

Projects	Accuracy@1				Accuracy@2			
	Ours^{ACC1}	Ours^{ACC2}	Ours^{MRR}	Ours^{MAP}	Ours^{ACC1}	Ours^{ACC2}	Ours^{MRR}	Ours^{MAP}
ActiveMQ	0.852	0.852	0.860	0.860	0.940	0.936	0.936	0.936
Camel	0.851	0.866	0.866	0.866	0.938	0.941	0.941	0.941
Cassandra	0.781	0.781	0.781	0.789	0.912	0.912	0.912	0.904
CloudStack	0.835	0.830	0.835	0.830	0.936	0.919	0.936	0.919
DirectoryServer	0.838	0.853	0.838	0.846	0.912	0.941	0.912	0.926
HBase	0.788	0.786	0.801	0.793	0.879	0.898	0.900	0.895
Hadoop	0.782	0.781	0.786	0.786	0.888	0.902	0.898	0.898
Hive	0.785	0.811	0.794	0.796	0.886	0.891	0.896	0.893
Zookeeper	0.788	0.770	0.788	0.788	0.903	0.850	0.903	0.903
Average	*0.811*	*0.814*	*0.817*	*0.817*	*0.910*	*0.910*	*0.915*	*0.913*

TABLE 12: Effectiveness of our approach using different fitness measures on MRR and MAP

Projects	MRR				MAP			
	Ours^{ACC1}	Ours^{ACC2}	Ours^{MRR}	Ours^{MAP}	Ours^{ACC1}	Ours^{ACC2}	Ours^{MRR}	Ours^{MAP}
ActiveMQ	0.909	0.909	0.912	0.912	0.863	0.860	0.864	0.864
Camel	0.911	0.919	0.919	0.919	0.868	0.879	0.879	0.879
Cassandra	0.867	0.867	0.867	0.870	0.829	0.829	0.829	0.829
CloudStack	0.902	0.895	0.902	0.895	0.871	0.862	0.871	0.862
DirectoryServer	0.902	0.912	0.902	0.906	0.861	0.876	0.861	0.867
HBase	0.863	0.866	0.873	0.866	0.816	0.816	0.825	0.821
Hadoop	0.863	0.863	0.867	0.867	0.820	0.826	0.827	0.827
Hive	0.862	0.877	0.870	0.870	0.817	0.836	0.828	0.830
Zookeeper	0.868	0.848	0.868	0.868	0.840	0.817	0.840	0.840
Average	*0.883*	*0.884*	*0.887*	*0.886*	*0.843*	*0.845*	*0.847*	*0.847*

TABLE 13: Time costs of our approach

Projects	Train	Test	Test One
ActiveMQ	0.4 h	7.7 s	0.02 s
Camel	1.0 h	16.7 s	0.02 s
Cassandra	0.2 h	2.7 s	0.02 s
CloudStack	1.3 h	11.5 s	0.01 s
DirectoryServer	0.2 h	3.4 s	0.03 s
HBase	0.8 h	10.4 s	0.02 s
Hadoop	1.3 h	17.0 s	0.02 s
Hive	1.0 h	10.0 s	0.02 s
Zookeeper	0.2 h	2.2 s	0.02 s

training process is conducted offline only once, but recommendations are made online many times. In this research question, we want to investigate the training time cost and the test time cost of our approach to better understand its practicality.

Approach. For each studied project, we record the start time and the end time of the training process and the test process, calculate the time cost of each project and the average test time of each test sample. We conduct our experiments on a computer with a 3.6 GHz Intel Core i7 CPU, 64 GB RAM and 4 NVIDIA GeForce GTX 1080 GPUs with 8GB memory. For each project, we only use one GPU to train and test our approach.

Results. The time costs of our approach on each project are presented in Table 13. We can see that the training time of our approach ranges from 0.2 hours to 1.3 hours. The larger a project's training set is, the more time it takes to train the model. It takes less than 20 seconds to test our approach on each project. Moreover, it only takes less than 0.05 second for our approach to recommend variables to log for one code snippet.

6 Discussion

In this section, we discuss our case studies on two real-world logging bugs, the rationales of some design decisions, how to apply our approach to low-logging-quality projects and threats to the validity of our approach.

6.1 Case Studies on Real Logging Bugs

To investigate the usefulness of our approach in practice, we manually find two issues related to logging variables from Hadoop's issue tracking system[1]: HADOOP-7159[2] and HADOOP-10702[3]. The Java method related to HADOOP-7159 reads and processes an RPC (Remote Procedure Call) message from a client. HADOOP-7159 describes that developers missed logging "the client hostname when read exception happened", which made it hard to find "mismatched clients". Developers fixed this issue by adding the client object "c" in the corresponding logging statement. The Java method related to HADOOP-10702 initializes an "authentication handler instance" using "the principal and keytab specified in the configuration". A previous change modified the behavior of this method and made it possible

1. https://issues.apache.org/jira/projects/HADOOP/issues
2. https://issues.apache.org/jira/browse/HADOOP-7159
3. https://issues.apache.org/jira/browse/HADOOP-10702

TABLE 14: The recommendations of our approach for two Hadoop issues

	HADOOP-7159	HADOOP-10702
Before Fixing	[getName, e, count]	[keytab, principal]
After Fixing	[getName, e, ***c***, count]	[keytab, ***spnegoPrincipal***]
Cross-Project Model	[**e**, **getName**, ieo, key, **c**, **count**]	[**spnegoPrincipal**, principal, **keytab**, nameRules, spnegoPrincipals, trim]
Within-Project Model	[**c**, **getName**, **e**, **count**, ieo, key]	[**spnegoPrincipal**, principal, **keytab**, spnegoPrincipals, nameRules, config]

*For each issue, the last four rows refer to the variables used in its logging statement before and after it was fixed and the logging variables predicted by our cross-project prediction model and within-project prediction model. We only present the top-6 recommendations of our approach.

to "load multiple principal names or all HTTP principals in the key tab". But HADOOP-10702 reports that developers forgot to revise the related logging statement and did not "log the principal names correctly". Developers fixed this issue by replacing the wrong logging variable "principal" with the correct one "spnegoPrincipal" in the corresponding logging statement.

We investigate whether our approach can help avoid such issues by suggesting proper logging variables for the logging statement modified in each issue using both within-project prediction and cross-project prediction. For cross-project prediction, we use the prediction model mentioned in RQ3 (Section 5.5) with Hadoop as the target project. For within-project prediction, since the two issues are fixed in different releases, we train two new models for them. For each issue, we first build a dataset based on the source code before the issue is fixed. The Java method that was modified to fix this issue is removed from the dataset for avoiding noise. Next, we randomly select 90% of the samples for training and 10% for validation. Then, we train our approach on the new dataset and recommend logging variables for the logging statement in the issue using the trained model. Please note that our approach does not take the logging statement as input, so it does not know which variables were logged by developers before and after the issue was fixed. Finally, we post-process the ranked list of identifiers output by our approach. The identifiers that are class names are removed.

Table 14 shows the experimental results. We can see that in HADOOP-7159, developers missed logging the variable "c". Our cross-project prediction model recommends "c" as the fifth logging variable, and all four logged variables are covered in the top-6 recommendations. Our within-project prediction model performs better. It suggests "c" as the first logging variable, and its top-4 suggested variables are exactly the four logged variables. Developers fixed HADOOP-10702 by replacing "principal" with "spnegoPrincipal". Both our cross-project model and within-project model predict "spnegoPrincipal" as the variable with the highest probability to be logged. All logged variables are covered in the top-3 recommendations of the two models.

We can see from the two abovementioned examples that developers may miss logging important variables or log wrong variables. Our approach aims to learn common logging "rules" from existing logging statements, and such learned "rules" can help developers avoid mistakes and improve the logging quality. Specifically, for HADOOP-7159, developers may not realize that the client information is necessary for problem diagnosis when writing the corresponding logging statement. The logging statement is placed in a *catch* clause. "c" appears in the corresponding *try* clause and is usually used to name a client object in Hadoop. After manual inspection, we think the reason that our approach can successfully recommend "c" may be that our prediction model, especially the within-project prediction model, learns the patterns that the variables appearing in a *try* clause are likely to be logged in the corresponding *catch* clauses and that developers often log client such information in Hadoop. For HADOOP-10702, developers forgot to revise the logging statement after the behavior of the related Java method is changed, which resulted in a wrong logging variable. In the changed method, the logging statement is placed in a *for* loop and "spnegoPrincipal" is the loop variable. According to our manual inspection, our prediction model learns the pattern that a logging statement in a *for* loop usually logs the loop variable. Therefore, our approach is able to successfully recommend "spnegoPrincipal" as the first logging variable.

In summary, the two real-world examples show that our approach can help developers avoid issues related to logging variables and make us believe that our approach can be useful in practice.

TABLE 15: Number of exception instances in studied systems

Projects	#Exceptions	#Logged	Ratio
ActiveMQ	4926	1064	21.6%
Camel	6538	838	12.8%
Cassandra	2220	394	17.7%
CloudStack	6408	3046	47.5%
DirectoryServer	2040	252	12.4%
Hadoop	12258	2585	21.1%
HBase	5619	1856	33.0%
Hive	9391	2212	23.6%
Zookeeper	1166	483	41.4%
Average	*5618*	*1414*	*25.7%*

***#Exceptions** refers to the number of exception instances declared in *catch* clauses, **#Logged** refers to the number of such exception instances which are logged, and **Ratio** is the ratio of the **#Logged** to the corresponding **#Exceptions**.

6.2 Exception Instances

After an exception occurs, developers will sometimes record the corresponding exception instance in logs for post-mortem analysis. Exception instances are usually named following similar patterns. For example, an *Exception* instance is often named as *e*, and a *Throwable* instance is usually named as *t*. Thus, compared to other variables, learning to suggest exception instances is relatively easy. However, not every exception instances should be logged. Table 15 presents the number of exception instances (e.g., *catch (Exception e)*) and the number of logged exception instances in *catch* clauses in each project. We can see that there are on average only 25.7% of exception instances are logged. Zhu et al. also presented that in their dataset, the majority of exceptions are not logged [1]. According to developers' feedback, they also pointed out that logging every exception is not practical [1]. These facts show that deciding whether to log an exception instance is non-trivial, and it is worthwhile to learn logging practices from exception instances. Therefore, we do not remove exception instances from our dataset.

6.3 The Effects of Different Settings of Parameter K

Our approach takes as input k lines of code preceding a logging statement. By default, k is set to 15. We conduct a sensitivity study to investigate the effects of different settings of parameter k on the effectiveness of our approach.

We first set k to 1-20 and build corresponding datasets for each studied project. When extracting logging statements from source code, we assign a unique ID to each logging statement. The ID of a sample (i.e., a $\langle code\ tokens, labels, vars\rangle$ triple) is set to the ID of its corresponding logging statement. In the datasets of a project with different settings of k, we regard the samples with the same ID as the same sample regardless of whether they have the same $code\ tokens$ and $labels$. As described in Section 4.4, we use two filters, i.e., the *length filter* and the *var filter*, to preprocess our dataset. However, for the datasets of a project with different settings of k, the *length filter* and the *var filter* may cause them to contain different test sets. Because the longer k is, the more $code\ tokens$ a sample may have, and the smaller k is, the more likely it is that the $code\ tokens$ of a sample cover none of its $vars$. Different test sets result in unfair comparisons.

To fairly compare the performance of our approach with respect to different settings of k, we slightly modify our preprocessing procedures. Given a project, we first use the *length filter* to identify the samples that fail to meet the code length requirement in the dataset with the largest k (in our case, 20), and mark such samples as *long-code samples*. Then, we remove the *long-code samples* (identified by IDs) from all datasets to ensure the project's datasets contain the same set of samples after using the *length filter*. Finally, we divide each dataset into training, validation and test sets using the same random seed and only use the *var filter* to process the training and validation sets, keeping the test set unchanged. The reason is that the samples of which the $code\ tokens$ cover none of the $vars$ (referred to as *none-var samples*) are useless for training and validation due to their all-zero $labels$. However, we need to keep *none-var samples* in test sets to guarantee that the datasets with different settings of k contain the same set of samples as their test sets. Obviously, *none-var samples* will obtain a score of 0 for each metric during testing.

We evaluate our approach on the datasets of each project with different settings of k in terms of Accuracy@1, Accuracy@2, MRR as well as MAP, and calculate the average score of the nine studied projects in terms of every metric with each k. The corresponding average scores are plotted in Figure 8. We can see that the average scores of our

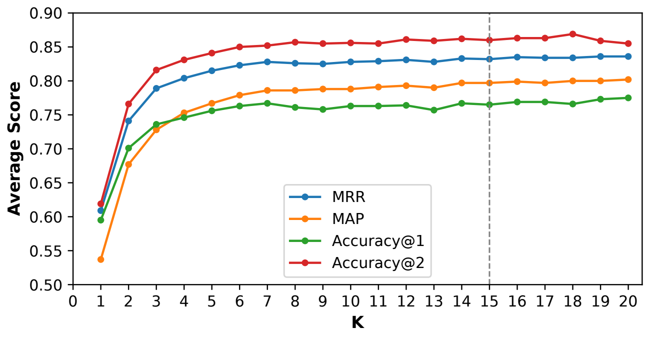


Fig. 8: The effects of different settings of parameter k

approach in terms of each metric increase with k when k is less than 8. When k is between 8 and 20, the scores are relatively stable. These results indicate that most logging variables can be inferred from the 8 lines of code preceding their corresponding logging statements, and k can be set to any number from 8 to 20 in general. However, due to the *length filter* and the *var filter*, both large k and small k may limit the number of samples in a dataset. Therefore, our approach sets k to 15 by default for balance. In addition, we also notice that different projects have different preferences of k in terms of diverse metrics. We suggest users choose the best k according to their own project characteristics and evaluation metrics that they deem to be more important.

6.4 The Effects of Different Word Embedding Techniques

There are three commonly used word embedding techniques, i.e., word2vec [26], [45], GloVe and `fastText` [46], [47]. To investigate the impacts of different word embedding techniques on our approach, we replace GloVe in our embedding layer with word2vec and `fastText` and construct two variants of our approach, respectively. Word2vec only provides one set of pre-trained word embeddings with 300 dimensions (for short, word2vec-300). For `fastText`, we use the set of word vectors trained with subword information on Wikipedia 2017, UMBC webbase corpus and statmt.org news dataset, of which the dimension is also 300 (for short, `fastText`-300). Our approach uses 100-dimensional GloVe word vectors (for short, GloVe-100) by default, but word2vec and fastText only provide 300-dimensional vectors. For fair comparisons, we construct another variant of our approach using the 300-dimensional word vectors pre-trained using GloVe (for short, GloVe-300).

We evaluate the four variants which use GloVe-100, GloVe-300, word2vec-300 and `fastText`-300 respectively on our dataset in terms of MAP. Table 16 presents our experimental results. We can see that on average, the variants using GloVe outperform those using Word2Vec and `fastText`, and the variant using word2vec performs worst. But the variants using GloVe do not perform best on all projects. Two projects prefer `fastText`-300. On the other hand, the performance differences between the variants using GloVe are small on most projects. On average, the variant using GloVe-100 (our default setting) obtains the equal MAP score to that using GloVe-300 with less compu-

TABLE 16: The effects of different types of pre-trained word embeddings

Embeddings	ActiveMQ	Camel	Cassandra	CloudStack	DirectoryServer	HBase	Hadoop	Hive	Zookeeper	Average
GloVe-100	0.864	0.879	0.829	0.862	0.867	0.821	0.827	**0.830**	**0.840**	**0.847**
GloVe-300	**0.869**	**0.880**	**0.841**	0.857	**0.871**	**0.824**	0.829	0.822	0.828	**0.847**
word2vec-300	0.847	0.865	0.794	0.862	0.859	0.810	0.810	0.807	0.800	0.828
`fastText`-300	0.850	0.876	0.800	**0.865**	0.855	0.821	**0.830**	0.825	0.831	0.839

tation cost. Therefore, our default choice to use GloVe-100 is reasonable.

6.5 The Effects of Different Pooling Operators in Our Embedding Layer

In the embedding layer of our approach, given a *compound word*, we use the average vector of its split tokens' word embeddings as its word vector. We also construct two variants of our approach by using two alternative pooling operators, i.e., sum and max, to build word vectors for *compound words*. We evaluate our default setting and the two variants, which are referred to as Ours$_{mean}$, Ours$_{sum}$ and Ours$_{max}$ respectively, on our dataset in terms of MAP. The evaluation results are shown in Table 17. We can see that on average, Ours$_{mean}$ performs best, while Ours$_{sum}$ performs worst. The reason may be that summing up word embeddings may magnify the bias in each word's embedding, and only keeping the max value in each dimension of word embeddings may lose some information about the relationship between words, while using the average vector equilibrates the information in each word's embedding. In addition, Ours$_{mean}$ obtains best MAP scores on 6 out of 9 projects. These results indicate that average vectors can better represent *compound words* than sum vectors and max vectors. Therefore, it is reasonable to use average vectors in our embedding layer.

6.6 How to Apply Our Approach to Projects With Low Logging Quality

Our approach aims to learn common logging practices about variables from existing logs. For low-logging-quality projects, it may not be appropriate to use our approach for within-project predictions. However, as we have shown in RQ3 (Section 5.5), our approach can be useful and effective for cross-project predictions. Therefore, if the logging quality of a target project is low, users can first select several projects with high logging quality to train our approach, and then use the trained model to make recommendations for the target project.

6.7 Threats to Validity

First, due to the use of pre-trained natural-language word embeddings, our word embedding method may not be suitable for projects which do not obey the common naming conventions, i.e., naming an identifier by using a meaningful word or by combining multiple common dictionary words and/or their abbreviations considering camel and snake casing. However, the framework of our approach is flexible, and users can replace our embedding method with a normal embedding layer to meet the requirements of such projects. Although using a normal embedding layer may reduce the performance of our approach to some extent, according to the evaluation results of RQ2 (Section 5.4), the reduced performance is still decent and our approach can still be helpful.

Second, in this work, given a code snippet, we only try to predict which variables in this code snippet should be logged, and can not suggest the accessible variables that are not in this code snippet, e.g., global variables. To avoid this constrain introducing bias into our evaluation results, we keep all logging variables in our preprocessed dataset (i.e., the $vars$ in $\langle code\ tokens, labels, vars\rangle$ triples) no matter whether they appear in corresponding code snippets or not for computations of our evaluation metrics. In future work, we plan to improve our approach to handle all accessible variables.

Another threat to validity is that our approach presumes our training data is representative of common and good logging practices, so that the trained model can represent high-quality logging specifications and generalize well for new code snippets. However, there is no "ground truth" for which variables to log given a code snippet. To mitigate this threat, we choose 9 mature, actively maintained Java projects that are of different domains and sizes as our subject projects. We believe the quality of the logging practice in these projects is high and the logging knowledge learned from them can guide the logging practices of other projects. To some extent, the good performance of our approach in the within-project prediction scenario and the cross-project prediction scenario supports our argument.

In addition, our variable extraction rules (described in Section 4.3) only regard the variables appearing in logging statements as logging variables. If developers concatenate variables using string concatenation operators or Java APIs like StringBuffer or StringBuilder and assign the concatenated string to a new variable for logging, our rules will only extract the new variable. However, in our dataset, such cases are much less than those where logging variables are directly concatenated in logging statements. Specifically, in the datasets of our studied projects, the percentages of the samples where the logging variables contain at least one instance of StringBuffer or StringBuilder are all less than 1%. Therefore, the threat is limited.

7 Related Work

This section describes the related studies on log analysis, logging practice and improving the logging quality.

7.1 Log Analysis

A great deal of runtime information of software systems is recorded in their execution logs. Prior research has leveraged log analysis to facilitate diverse software maintenance

TABLE 17: The effects of different pooling operators in our word embedding layer

Operations	ActiveMQ	Camel	Cassandra	CloudStack	DirectoryServer	HBase	Hadoop	Hive	Zookeeper	Average
Mean	0.864	**0.879**	**0.829**	0.862	**0.867**	**0.821**	**0.827**	0.830	**0.840**	**0.847**
Sum	0.859	0.875	0.809	0.870	0.859	0.808	0.816	0.828	0.813	0.837
Max	**0.866**	0.875	0.796	**0.874**	0.866	0.817	0.821	**0.831**	0.816	0.840

tasks, including execution anomaly detection [2], [3], problem diagnosis [4]–[7], deployment verification [8], etc. For instance, Xu et al. [3] proposed an approach to extract composite features from logs by combining source code analysis and information retrieval, and built a machine learning model which can detect runtime problems efficiently based on such features. To reduce the deployment verification efforts of Big Data Analytics Applications, Shang et al. [8] proposed a method which can recover execution sequences from logs and identify deployment failures by comparing such sequences between test deployment and cloud deployment. These tasks highlight the requirement of proper logging practices and motivate our work to help developers write high-quality logging statements by recommending which variables to log during development.

7.2 Logging Practice

Researchers have empirically studied diverse aspects of software logging practices [9]–[12], [48]–[53]. Fu et al. [10] analyzed two large industrial systems to explore where developers log, and summarized five categories of logged code snippets. Pecchia et al. [50] studied the logging practices in a company and highlighted the need for standardizing a logging policy in such company. Li et al. [12] empirically studied the logging practices in six open source systems, and found whether a code snippet contains logging statements is strongly related to its topic. The insightful information offered by these studies cast lights on the design of our approach. In addition, Shang et al. [49] have shown that the functionality of log processing applications is often affected by the changes of logging statements, and resources also need to be allocated to maintain such applications with the evolution of logs. Kabinna et al. [53] performed a study on the stability of logging statements, and found that over 20% logging statements in four open source projects were changed at least once. Yuan et al. [9] also found that more than 30% of logging statements in four open-source software were modified at least once, and they pointed out that one fourth of such modifications were to variable logging. Chen et al. [11] conducted an empirical study on 21 Java-based open source projects, and also found that 27% of after-thought logging changes (i.e., the changes to logging statements that are independent of other changes) were related to variable logging. Such prior findings motivate our work towards assisting developers in logging appropriate variables at the first time.

7.3 Improving Logging

Some prior work focuses on improving the logging quality, which can be divided into two categories: "where to log" and "what to log". "Where to log" studies aim to suggest the placement of logging instructions [1], [52], [54], [55]. For example, Cinque et al. [54] distilled a set of rules to guide the placement of logging statements based on system design artifacts. Zhu et al. [1] proposed a tool named *LogAdvisor*, which applies machine learning techniques to learn common logging practices from existing logging instances and suggest whether a code snippet should be logged. In contrast, the objective of "what to log" studies is the content of logging statements [13], [14], [48], [56], i.e., the log levels, the static text and the logging variables. For example, Li et al. [13] proposed an approach which leverages ordinal regression models to automatically recommend proper log levels for newly-added logging statements. He et al. [14] leveraged an information retrieval method to automatically generate natural language descriptions for logging instructions based on existing logging instances. Unlike them, our work focuses on logging variables.

Yuan et al. proposed a tool, named LogEnhancer, to ease the diagnosis of software failures by automatically inferring and inserting more causally-related variables, i.e., variables which may affect whether a logging statement will be executed, into existing logging statements [56]. It tries to add many variables (on average 14.6 variables per logging statement) before compiling to help recover the execution paths of the failures occurred in the future. Our approach is different from LogEnhancer in the following aspects: 1) Usage Scenarios. Our approach aims to help developers decide which variables to log during development, while LogEnhancer is expected to be used offline prior to software release to enhance existing logging messages [56]. 2) Input and output. LogEnhancer takes the source code of a program as input and enhances its existing logging statements by exhaustively adding causally-related variables. The added variables are not visible in the source code, often far more than what developers would record and not prioritized. Different from LogEnhancer, our approach takes as input code snippets (which may not be compilable) and prioritizes accessible variables to help developers write high-quality logging statements at the first time. 3) Target logs. LogEnhancer needs to add many variables and may introduce performance overhead, hence it is more suitable for error or fatal logs. However, our approach focuses on recommending and prioritizing logging variables and can be used for all kinds of logs. 4) Implementation. Currently, LogEnhancer is implemented for C programs, while our data extraction procedures are designed for Java programs and our approach is programming language agnostic.

Since the objective and/or the usage scenarios of our approach differ from those of existing tools, we believe our work is a complement, instead of a competitor, to them.

8 Conclusion and Future Work

This work aims to suggest proper variables to log for developers during development. We point out two challenges of solving this problem, i.e., dynamic labels and out-of-vocabulary words, and propose a neural-network-based approach to handle these challenges. To handle dynamic labels, we treat this problem as a representation learning problem instead of a multi-label classification problem. Our approach first leverages a multi-layer neural network to learn to represent program tokens, and then uses a unified binary classifier to predict whether an identifier should be logged based on such learned representations. To deal with out-of-vocabulary words, our approach does not jointly learn word embeddings with the target task from scratch. Instead, we propose a novel method to map program tokens into word embeddings with the help of pre-trained natural-language word embeddings. We evaluate our approach on 9 mature open source Java projects. Our experimental results show that our approach outperforms five baselines, i.e., random guess and four variants of an IR-based method, by large margins.

For now, our approach only takes as input the code preceding a logging statement and only considers the variables in the preceding code as candidates. However, the code succeeding a logging statement can also be helpful for logging variable recommendation and the variables not appearing in the preceding code may also be logged, e.g., global variables. In the future, we plan to investigate the effect of the succeeding code on recommending logging variables and improve our approach to consider all accessible variables for more accurate suggestions on which variables to log. The training of our proposed model relies on one or more projects with high logging quality. Nevertheless, to the best of our knowledge, there is no prior work focusing on quantifying and assessing the logging quality of an existing project, which can be an interesting direction for future work. Also, future work could integrate our approach and other approaches that address "where to log" and "what to log" issues. Such integrated approach can automatically provide comprehensive logging guidance for developers, and its components may benefit from one another and result in better performance.

References

[1] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 415–425.

[2] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 2009, pp. 149–158.

[3] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.

[4] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*. IEEE, 2008, pp. 117–126.

[5] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *ACM SIGARCH computer architecture news*, vol. 38, no. 1. ACM, 2010, pp. 143–154.

[6] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 26–26.

[7] D.-Q. Zou, H. Qin, and H. Jin, "Uilog: Improving log-based fault diagnosis by log analysis," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 1038–1052, 2016.

[8] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin, "Assisting developers of big data analytics applications when deploying on hadoop clouds," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 402–411.

[9] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.

[10] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 24–33.

[11] B. Chen and Z. M. J. Jiang, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[12] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, pp. 1–40, 2018.

[13] H. Li, W. Shang, and A. E. Hassan, "Which log level should developers choose for a new logging statement?" *Empirical Software Engineering*, vol. 22, no. 4, pp. 1684–1716, 2017.

[14] P. He, Z. Chen, S. He, and M. R. Lyu, "Characterizing the natural language descriptions in software logging statements," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 178–189.

[15] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[16] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

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

[18] 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.

[19] J. Chiu and E. Nichols, "Named entity recognition with bidirectional lstm-cnns," *Transactions of the Association of Computational Linguistics*, vol. 4, no. 1, pp. 357–370, 2016.

[20] P. Wang, Y. Qian, F. K. Soong, L. He, and H. Zhao, "Part-of-speech tagging with bidirectional long short-term memory recurrent neural network," *arXiv preprint arXiv:1510.06168*, 2015.

[21] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[22] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[23] T. K. Landauer and S. T. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge." *Psychological review*, vol. 104, no. 2, p. 211, 1997.

[24] P. D. Turney and P. Pantel, "From frequency to meaning: Vector space models of semantics," *Journal of artificial intelligence research*, vol. 37, pp. 141–188, 2010.

[25] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[27] "The website of glove," https://nlp.stanford.edu/projects/glove/, 2019.

[28] V. Mnih, N. Heess, A. Graves *et al.*, "Recurrent models of visual attention," in *Advances in neural information processing systems*, 2014, pp. 2204–2212.

[29] M. F. Stollenga, J. Masci, F. Gomez, and J. Schmidhuber, "Deep networks with internal selective attention through feedback connections," in *Advances in neural information processing systems*, 2014, pp. 3545–3553.

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

[31] 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.

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

[33] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber *et al.*, “Gradient flow in recurrent nets: the difficulty of learning long-term dependencies,” 2001.

[34] “Pytorch: An open source deep learning platform,” https://pytorch.org/, 2019.

[35] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[36] “Apache log4j 2,” https://logging.apache.org/log4j/2.x/, 2019.

[37] “Simple logging facade for java (slf4j),” https://www.slf4j.org/, 2019.

[38] “Apache commons logging,” https://commons.apache.org/proper/commons-logging/, 2019.

[39] “Eclipse java development tools (jdt),” https://www.eclipse.org/jdt/, 2019.

[40] R. Baeza-Yates, B. d. A. N. Ribeiro *et al.*, *Modern information retrieval*. New York: ACM Press; Harlow, England: Addison-Wesley,, 2011.

[41] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[42] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, “From word embeddings to document similarities for improved information retrieval in software engineering,” in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 404–415.

[43] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[44] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[45] “The webisite of word2vec,” https://code.google.com/archive/p/word2vec/, 2019.

[46] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, “Bag of tricks for efficient text classification,” in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*. Valencia, Spain: Association for Computational Linguistics, Apr. 2017, pp. 427–431. [Online]. Available: https://www.aclweb.org/anthology/E17-2068

[47] “The webisite of fasttext,” https://fasttext.cc/, 2019.

[48] D. Yuan, S. Park, P. Huang, Y. Liu, M. M.-J. Lee, X. Tang, Y. Zhou, and S. Savage, “Be conservative: Enhancing failure diagnosis with proactive logging.” in *OSDI*, vol. 12, 2012, pp. 293–306.

[49] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora, “An exploratory study of the evolution of communicated information about the execution of large software systems,” *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 3–26, 2014.

[50] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, “Industry practices and event logging: assessment of a critical software development process,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 169–178.

[51] W. Shang, M. Nagappan, and A. E. Hassan, “Studying the relationship between logging characteristics and the code quality of platform software,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, 2015.

[52] H. Li, W. Shang, Y. Zou, and A. E. Hassan, “Towards just-in-time suggestions for log changes,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[53] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, “Examining the stability of logging statements,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 290–333, 2018.

[54] M. Cinque, D. Cotroneo, and A. Pecchia, “Event logs for the analysis of software failures: A rule-based approach,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 806–821, 2013.

[55] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, “Log20: Fully automated optimal placement of log printing statements under specified overhead threshold,” in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 565–581.

[56] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, “Improving software diagnosability via log enhancement,” *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 4, 2012.