RefBERT: A Two-Stage Pre-trained Framework for Automatic Rename Refactoring

Hao Liu
Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China
School of Informatics, Xiamen University
China
haoliu@stu.xmu.edu.cn

Yanlin Wang
School of Software Engineering
Sun Yat-sen University
China
wangylin36@mail.sysu.edu.cn

Zhao Wei
Yong Xu
Juhong Wang
Tencent
China
{zachwei,rogerxu,julietwang}@tencent.com

Hui Li*
Rongrong Ji
Key Laboratory of Multimedia Trusted Perception and Efficient Computing, Ministry of Education of China
School of Informatics, Xiamen University
China
{hui,rrji}@xmu.edu.cn

ABSTRACT

Refactoring is an indispensable practice of improving the quality and maintainability of source code in software evolution. Rename refactoring is the most frequently performed refactoring that suggests a new name for an identifier to enhance readability when the identifier is poorly named. However, most existing works only identify renaming activities between two versions of source code, while few works express concern about how to suggest a new name. In this paper, we study automatic rename refactoring on variable names, which is considered more challenging than other rename refactoring activities. We first point out the connections between rename refactoring and various prevalent learning paradigms and the difference between rename refactoring and general text generation in natural language processing. Based on our observations, we propose RefBERT, a two-stage pre-trained framework for rename refactoring on variable names. RefBERT first predicts the number of sub-tokens in the new name and then generates sub-tokens accordingly. Several techniques, including constrained masked language modeling, contrastive learning, and the bag-of-tokens loss, are incorporated into RefBERT to tailor it for automatic rename refactoring on variable names. Through extensive experiments on our constructed refactoring datasets, we show that the generated variable names of RefBERT are more accurate and meaningful than those produced by the existing method. Our implementation and data are available at https://github.com/KDEGroup/RefBERT.

*Corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598092

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; **Software notations and tools**; • **Computing methodologies** → **Natural language processing**.

KEYWORDS

rename refactoring, language modeling, contrastive learning, bag-of-tokens loss

ACM Reference Format:
Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. RefBERT: A Two-Stage Pre-trained Framework for Automatic Rename Refactoring. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598092

1 INTRODUCTION

Refactoring improves the internal code structure without altering external behavior [18]. It is a crucial activity often involved in software evolution, aiming at improving the quality of software projects and accelerating the process of development. Renaming refactoring identifies the inappropriate use of an identifier (e.g., variable, type, method and class names) and provides a meaningful name to replace. In practice, a large portion of source tokens are identifiers. For example, Deißenböck and Pizka [12] analyze the code base of Eclipse and find that identifiers account for 32.8% of the tokens and 71.7% of the characters in the source code. Developers often apply renaming operations for better readability and maintainability of source code, which helps avoid software fault-proneness [6], and renaming refactoring is the most frequently performed refactoring type that occupies developers' working time [20, 46, 47]. Choosing a meaningful name for an identifier is non-trivial and demands both professional and contextual knowledge of programs.

In the literature, various works study refactoring, but most of them focus on automatic refactoring detection [15, 30, 51, 56, 59, 60], i.e., detect the refactoring activities between two versions of source

code and identify the refactoring type. For example, Malpohl et al. [42] design a method to detect rename refactoring and Arnaoudova et al. [6] propose REPENT to detect and classify identifier renaming. But these efforts aim at identifying whether refactoring operations exist between two versions of source code, rather than providing automatic renaming. Only a few works study automatic rename refactoring [1, 33, 58]. A representative work is Naturalize [1] which leverages the n-gram language model to estimate the probability that a specific name should be used in a given context to rename an identifier.

To accelerate rename refactoring and reduce the intellectual burden of developers, in this paper, we propose a two-stage pre-trained framework for automatic Rename reFactoring based on the BERT architecture [14] (RefBERT). Particularly, **we study rename refactoring on variable names**, which is considered much more challenging than refactoring other types of identifiers such as method names and type names [1]. **Hence, throughout this paper, rename refactoring refers to rename refactoring on variable names.** Note that we refactor all the references of a variable name in one function. There are other works for rename refactoring on other identifiers. For example, rename refactoring on method names [35, 36, 50]. But they differ from our task. One variable name can appear in multiple positions in a function, and the positions of variables vary in different functions. Differently, method names typically appear in the beginning of the function.

The design of RefBERT is based on the following observations:

- Rename refactoring is essentially similar to masked language modelling (MLM), which is a pretext task commonly used in pre-training BERT [14, 39]. MLM fills the masked part of a text according to its context. Similarly, rename refactoring aims to suggest a meaningful variable name according to the context. Therefore, we believe MLM can be adopted for training an automatic rename refactoring model.
- Unlike the variable name prediction task [4, 53, 61], both the context of the target variable and the variable name before refactoring are known in rename refactoring. Contrastive learning [38], which contrasts positive and negative samples for improving representation learning, is an ideal learning paradigm for automatic rename refactoring: we expect the generated name to be close to the variable name after refactoring but far away from the variable name before refactoring.
- Although rename refactoring can be viewed as a generation task in natural language processing (NLP), the standard cross-entropy loss for text generation is not suitable for rename refactoring. Unlike natural language text where words should follow a strict order to ensure grammatical correctness, sub-tokens in a variable name do not have such a restriction. Different orders of sub-tokens for a variable name do not significantly affect our understanding of the variable. Thus, the standard cross-entropy loss that emphasizes the strict alignment between the prediction and the target is suboptimal for automatic rename refactoring.

Based on the above observations, we propose various techniques to endow RefBERT with the ability of renaming variables. In summary, our contributions are:

- We point out the connections between rename refactoring and various prevalent learning paradigms (MLM and contrastive learning) and the difference between rename refactoring and general text generation in NLP (the inappropriateness of using standard cross-entropy loss), which, as far as we know, has not been mentioned in the literature.
- We construct a refactoring dataset JavaRef, and it can be used to evaluate automatic rename refactoring and other refactoring tasks. We further adapt an existing code corpus TL-CodeSum as a supplement to JavaRef.
- We design a two-stage framework RefBERT for automatic rename refactoring. RefBERT first predicts the number of sub-tokens in the new name and then generates sub-tokens accordingly. Various techniques, including constrained masked language modeling, contrastive learning, and the bag-of-tokens loss, are incorporated into RefBERT to tailor it for automatic rename refactoring.
- We conduct extensive experiments to demonstrate the effectiveness of RefBERT. The experimental results show that RefBERT provides more accurate and meaningful suggestions for rename refactoring than the existing method.

The rest of this paper is organized as follows. In Sec. 2, we introduce the related work of RefBERT. In Sec. 3, we describe how we collect and construct two refactoring datasets JavaRef and TL-CodeSum. Our framework RefBERT is illustrated in Sec. 4. We then demonstrate experimental settings in Sec. 5. The experimental results are reported and analyzed in Sec. 6. In Sec. 7, we discuss possible threats to the validity of our work. Finally, we conclude our work in Sec. 8.

2 RELATED WORK

In this section, we discuss several areas that are related to our work.

2.1 Automatic Refactoring Detection

Existing studies of refactoring mostly focus on detecting refactoring activities between two versions of source code and identifying the refactoring type, i.e., automatic refactoring detection. Automatic refactoring detection helps update applications to use the new version of its components that have changed the interfaces [15], and enhance the understanding of software evolution [52, 55]. Weißgerber and Diehl [65] identify refactorings based on clone detection. Demeyer et al. [13] propose a set of heuristics based on object-oriented metrics for automatic refactoring detection. Negara et al. [48] design Refactoring Inference Algorithm that can detect 10 refactorings according to refactoring properties. Dig et al. [15] develop RefactoringCrawler that applies syntactic analysis and semantic analysis to detect 7 types of refactorings in Java projects. Prete et al. [51] and Kim et al. [30] design Ref-Finder which expresses each refactoring type in terms of template logic rules and uses a logic programming engine to detect refactorings. Tsantalis et al. [59, 60] propose RefactoringMiner that uses an abstract syntax tree (AST) based statement matching algorithm to detect refactorings. Silva and Valente [56] propose RefDiff, which employs both heuristics based on static analysis and code similarity to detect 13 refactoring types. There are also a few works specially designed for detecting the rename refactoring [6, 42].

Unlike the above works that take two versions of source code as inputs for detecting refactorings, our work requires a method containing a target variable as input and provides suggestions for rename refactoring, i.e., variable rename refactoring.

2.2 Automatic Rename Refactoring

Developers spent much of their time in software refactoring [20] and rename refactoring is the most frequently performed refactoring operations, as reported by Murphy-Hill et al. [47].

There are a few works on automatic rename refactoring[1]. Caprile and Tonella [8] first map each word in the identifier to a standard lexicon and then the sequence of words for the identifier is required to be compliant with a grammar. Thies and Roth [58] present a tool to support identifier renaming based on information extracted via static code analysis. Feldthaus and Møller [16] propose a semi-automatic refactoring method that combines static analysis and interaction with the programmer for rename refactoring in JavaScript. Mayer and Schroeder [44] apply multiple existing language-specific refactoring routines for multi-language artifact binding and rename refactoring. Naturalize, developed by Allamanis et al. [1], is the prior work that adopts NLP techniques for refactoring. Naturalize leverages a n-gram language model to suggest identifier names to replace those that break coding conventions. It first retrieves a set of candidate names that have appeared in the similar context from other code snippets, and then ranks the candidates by measuring naturalness through a learned n-gram model. The idea of Naturalize has inspired other researchers to explore using language modeling for rename refactoring [33].

Most of the above works are rule-based or semi-automatic while RefBERT is a learning-based rename refactoring method that does not require complex features and rules. Naturalize that adopts language modeling for automatic rename refactoring is mostly related to our work. However, RefBERT differs from Naturalize in that we employ a pre-trained framework that benefits rename refactoring by leveraging a large volume of public code data. Besides, RefBERT conducts token-level rename refactoring that better captures the semantics of variables.

2.3 Variable Name Prediction

Variable name prediction is a related but different task to rename refactoring. The goal of variable name prediction is to generate a variable name that is as close as possible to the missing variable name. Differently, rename refactoring generates variable names from the perspective of code refactoring and the resulting variable name may be quite different but more readable compared to the original one.

There are some studies on variable name prediction. Raychev et al. [53] leverage conditional random fields to predict variable names by modeling relations among variables and program elements. Vasilescu et al. [61] recover variable names through statistical machine translation. Alon et al. [4] represent a program using paths in its AST to predict method names, variable names and variable types.

[1]Note that, in Sec. 2.2, we introduce related work of general rename refactoring on different types of identifiers. But this paper focus on rename refactoring on variable names.

2.4 Code Representation Learning

Recently, learning code representations has attracted great attention since it can assist various code-related tasks, including but not limited to code summarization [34], code completion [64] and code search [32]. A great number of works on code representation learning have sprung up. Earlier approaches apply the idea of word2vec [45] to obtain code representations [7, 9]. More recent works [17, 28, 29] mostly adopt pre-training techniques which can fully leverage the large volume of public source code in platforms like GitHub. Most of them are built based on the BERT architecture [14], which has achieved dramatic empirical improvements in multiple AI tasks. Some representative works include CodeBERT [17], GraphCodeBert [23], T5 [43], UniXcoder [22] and SPT-code [49], and they have benefited various downstream code-related tasks.

3 DATA CONSTRUCTION

We can utilize existing large code corpora, even those not related to rename refactoring, to pre-trained RefBERT to enhance its general understanding of programming language. When fine-tuning RefBERT on automatic rename refactoring, RefBERT requires rename refactoring data corpora that can be smaller than pre-training data but contain ground-truth rename refactoring. To prepare datasets for training and evaluating automatic rename refactoring approaches, we use rules to modify an existing code corpus TL-CodeSum[2] [25] originally designed for the code summarization task so that it can be used in the automatic rename refactoring task. Additionally, we collect and construct a Java refactoring corpus named JavaRef through manual inspection and using automated tools.

3.1 Adapt Existing Code Corpus for Rename Refactoring

Since the cost of manually labeling a large refactoring corpus is exorbitant, we first adapt the existing code corpus for the automatic rename refactoring task. We choose TL-CodeSum from the literature as the target dataset. TL-CodeSum is originally used in the code summarization task. We use AstParser[3] to parse a function into an AST and retrieve variable names from AST nodes. Since the original TL-CodeSum does not have code before/after rename refactoring, we assume that variable names in TL-CodeSum are already meaningfully and reasonably designed, and they do not require rename refactoring. We perform the following steps to adapt TL-CodeSum:

(1) We randomly pick one variable name from each function in TL-CodeSum to construct a variable name set.
(2) For each function, the picked variable name v is treated as the name after rename refactoring (i.e., ground-truth refactored variable names). We further sample another variable name v' from the variable name set as the name of v before rename refactoring.
(3) We perform subword tokenization on the modified TL-CodeSum. We adopt the subword tokenization approach

[2]https://github.com/xing-hu/TL-CodeSum
[3]https://github.com/Ragnaroek/eclipse-astparser

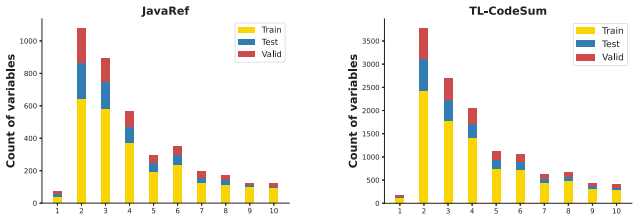

Figure 1: Distributions of the frequency of variable names.

of Byte-Pair Encoding [54] used by RoBERTa-base [39] to split code into subwords and variable names are divided into tokens with a finer granularity. This step is essential for rename refactoring. As the data of variable names is extremely sparse (i.e., developers can name a variable freely and most variable names only appear a few times in a large code corpus as shown in Fig. 1), variable names' sub-tokens that are more likely to occur before are a better encoding unit. Encoding sub-tokens, which occur more frequently, can better capture the semantics of variable names and alleviate the data sparsity issue of variable names.

Finally, we have a large dataset for automatic rename refactoring and it contains 65,324 rename refactoring data records.

3.2 Construction of JavaRef

The construction of the Java refactoring corpus JavaRef involves two parts:

Data Collection. This part contains several data collection steps:

(1) Tsantalis et al. [60] and Claes et al. [11] provide two datasets containing commit data in many open-source Java projects. Based on the project popularity (i.e., numbers of stars and forks), we select 114 java open-source GitHub projects containing 8,451 commits from their datasets.
(2) Then, to extract the refactoring history from commit data, we employ a popular refactoring type detector RefactoringMiner [59, 60] which shows high refactoring detection accuracy. The detection result contains the hash value of a commit. It also includes refactoring types and corresponding descriptions between two versions of the project before and after the commit. But the result does not have the information of changed source code files and code diff data.
(3) Based on the retrieved refactoring information, we further adopt a Python library PyDriller[4] to extract the url of each changed source code file and line numbers of the changed code. We crawl changed source code files before/after refactoring from GitHub according to these urls.

Data Preprocessing. We perform the following preprocessing steps in sequence:

(1) To improve data quality and eliminate the noise brought by the incorrect results from RefactoringMiner and PyDriller, we invite five students in Computer Science major to manually verify the collected data in a period of three months. They are asked to link a changed code snippet (i.e., code diff) to one of the refactoring types detected by using RefactoringMiner to compare the two commits where the code snippet is changed.
(2) Then, we keep verified, function-level refactoring data and filter other refactorings. We further remove duplicated code and functions shorter than three lines.
(3) We further extract the variable names in the data record. We use AstParser to parse the function into an AST and retrieve variable names from AST nodes.
(4) Finally, we perform subword tokenization on JavaRef in a similar way as preprocessing TL-CodeSum.

In total, JavaRef contains 17,910 refactorings, covering 25 refactoring types. Each data record in JavaRef consists of refactoring type, refactoring description, original code (before refactoring) and current code (after refactoring). If a code sample in JavaRef does not correspond to a rename refactoring operation, we use the same method for preprocessing TL-CodeSum to prepare variable names before and after rename refactoring.

4 OUR FRAMEWORK REFBERT

4.1 Overview

Compared to general-purpose code corpora like CodeSearchNet [26], the volume of rename refactoring data corpora is relatively small. Inspired by the success of large code pre-trained models [17, 22, 23, 43, 49] on various downstream code-related tasks like code search and code summarization, we opt to adopt a pre-trained architecture for RefBERT. In short, RefBERT is firstly pre-trained over large-scale code corpora and then fine-tuned over relatively smaller refactoring data.

Our design is based on not only the conclusion from previous studies [17, 49] that pre-training can endow the model with a better generalization ability via training over large related data but also the following observations: (1) the prevalent pretext task in pre-training is essentially similar to rename refactoring; (2) the recently popular self-supervised learning paradigm, contrastive learning [38], fits the nature of rename refactoring and can be adopted in the fine-tuning phase; (3) sub-tokens in a variable do not need to follow a strict order.

Fig. 2 provides an overview of RefBERT. RefBERT is trained to generate refactored variable names in two steps:

Length Prediction (LP): In the LP task, RefBERT predicts the number of tokens in the refactored variable name.

Token Generation (TG): Given the predicted number of tokens, RefBERT generates tokens in the refactored variable name in the TG task.

4.2 Architecture of RefBERT

Prevalent pre-trained models in NLP typically adopt the architecture of BERT [14] which contains a multi-layer bidirectional Transformer [62]. We follow this design and use 12 RoBERTa layers in RefBERT (i.e., RoBERTa-base in [39]), which is a replication of the original BERT model with improved performance. Given the widespread adoption of Transformer in software engineering field [67], we do not introduce its background in detail.

[4] https://github.com/ishepard/pydriller

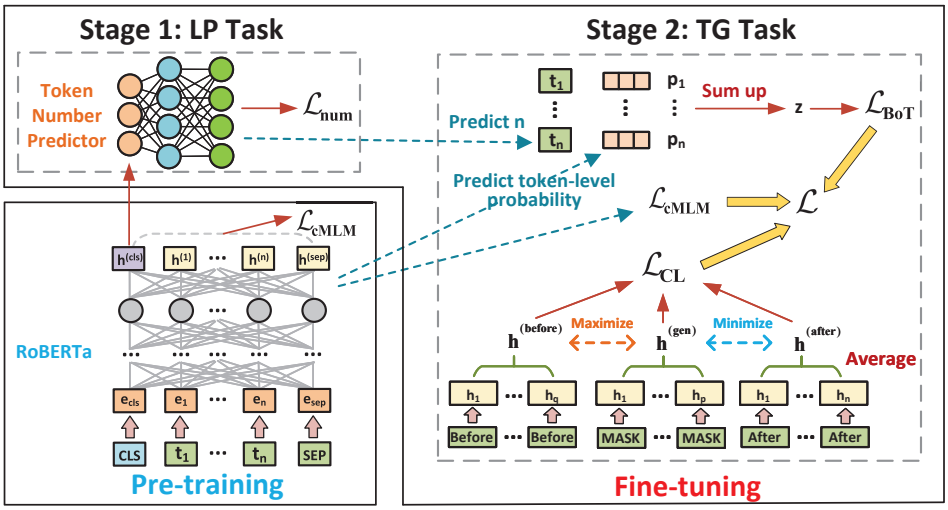

Figure 2: Overview of RefBERT.

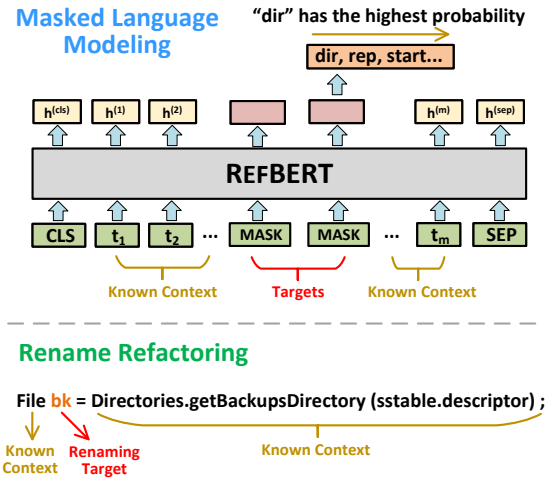

Figure 3: A comparison of MLM and rename refactoring.

Model Input. RefBERT takes tokens of a code snippet as input: $\{[\text{CLS}], t_1, \cdots, t_g, [\text{SEP}]\}$, where t denotes a token and g is the number of tokens in the code snippet. [CLS] and [SEP] are two special tokens indicating the start and end of a sequence.

Model Output. RefBERT produces two types of output: the contextual representation vector for each token and the representation of [CLS]. The latter serves as the aggregated sequence representation and is employed to predict the number of tokens in a variable name (see Sec. 4.4.1).

4.3 Pre-train RefBERT

During pre-training, RefBERT can benefit from training over large code corpora and we adopt two datasets CodeSearchNet [26] and Java-small [3] in pre-training. Their details are described in Sec. 5.1.

We find that rename refactoring is essentially similar to *masked language modeling* (MLM), which is the primary pretext task used in various pre-trained models [14, 17, 27, 57]. As shown in Fig. 3, MLM masks some tokens in the input and trains the model to precisely predict masked tokens based on unmasked tokens. MLM does not require additional labels and can leverage intrinsic information of the input data. Similarly, in rename refactoring, the goal is to predict the unknown tokens constituting the variable name based on other known tokens in the code snippet. If we treat unknown and known tokens in rename refactoring as masked tokens and unmasked tokens in MLM, we can find that the two tasks are similar as depicted in Fig. 3. MLM has been proven effective for learning rich token representations in various applications [14, 40]. The similarity between MLM and rename refactoring and the success of MLM in other domains have motivated us to adopt the idea of MLM in pre-training RefBERT for rename refactoring.

To be specific, the objective of MLM is to predict the original token, which is replaced by a [MASK] token during training, based only on its context. Let $T_s^{(m)} = \{t_{s,1}^{(m)}, \cdots, t_{s,g_s^{(m)}}^{(m)}\}$ and $T_s^{(u)} = \{t_{s,1}^{(u)}, \cdots, t_{s,g_s^{(u)}}^{(u)}\}$ denote the set of masked tokens and unmasked tokens in a code snippet s, respectively. $g_s^{(m)}$ and $g_s^{(u)}$ are the numbers of masked tokens and unmasked tokens, respectively. The tokens in $T_s^{(m)}$ and $T_s^{(u)}$ are sorted according to their order in s. Then, the objective of MLM can be defined as:

$$\mathcal{L}_{\text{MLM}}(\theta) = -\sum_{s \in \mathcal{S}} \sum_{i=1}^{g_s^{(m)}} \log p\left(t_{s,i}^{(m)} | T_s^{(u)}; \theta\right), \tag{1}$$

where θ indicates the model parameters and $\mathcal{S}$ is the code corpus.

There exist various masking strategies for MLM. The most common one is to randomly mask the tokens in sentences with a fixed ratio, e.g., 15% in BERT [14]. Since rename refactoring focuses on variable names in a code snippet, we propose to use *constrained*

masked language modeling (cMLM) in RefBERT. cMLM masks chosen tokens instead of randomly picked tokens. Given a refactored code snippet s, RefBERT always masks all tokens in the target variable name with $g_s^{(m)}$ [MASK] tokens.

The representation of the i-th [MASK] token ($1 \le i \le g_s^{(m)}$) from RefBERT is fed into an output softmax over the token vocabulary $\mathcal{V}$ to produce the probability distribution $\mathbf{p}_{s,i} \in \mathbb{R}^{|\mathcal{V}|}$ of all tokens for being the corresponding masked token, where $|\mathcal{V}|$ denotes the vocabulary size. The cMLM task can be trained with a standard cross entropy loss for classification:

$$\mathcal{L}_{\text{cMLM}} = -\sum_{s\in\mathcal{S}}\sum_{i=1}^{g_s^{(m)}}\sum_{j\in\mathcal{V}} y_{s,i,j}\log p_{s,i,j}, \tag{2}$$

where $y_{s,i,j}$ equals 1 if the i-th masked token in code snippet s is token j otherwise 0. $p_{s,i,j}$ is the predicted probability of the i-th masked token being j (i.e., i-th dimension in $\mathbf{p}_{s,i}$) and it is produced by the output softmax.

It is worth pointing out that we adopt cMLM in both pre-training phase and fine-tuning phase since it is critical for training a rename refactoring model.

4.4 Fine-Tune RefBERT

There are two stages in the fine-tuning phase. RefBERT is firstly trained to predict token number. Then it is guided to generate variable name tokens precisely over refactoring data (JavaRef and TL-CodeSum).

4.4.1 Stage 1: Predict token number (LP task). Since code corpora have been preprocessed and a variable name is split into multiple tokens in order to better capture its semantics and alleviate the data sparsity issue, before generating tokens, RefBERT should first predict the number of tokens (i.e., masked token number $g_s^{(m)}$) in the variable name after refactoring.

The representation of the first special token [CLS] in each input sequence to pre-trained models is commonly used for sentence classification or ranking tasks [14, 40]. Inspired by this, we use the representation $\mathbf{h}_s^{(\text{cls})}$ of the [CLS] token in a code snippet s to predict $g_s^{(m)}$. Note that, unlike cMLM which masks each token in a variable name with one [MASK] token, *we mask each variable name with only one [NUM] token in predicting the number of tokens* to avoid data leakage since the number of masked tokens reveals the ground truth. We feed $\mathbf{h}_s^{(\text{cls})}$ into a single-layer feedforward neural network (token number predictor) to predict the number of tokens in the masked variable name of s:

$$\mathbf{q}_s = \mathbf{W}_q\mathbf{h}_s^{(\text{cls})} + \mathbf{b}_q, \tag{3}$$

where $\mathbf{W} \in \mathbb{R}^{l_{\max}\times d}$ and $\mathbf{b}_l \in \mathbb{R}^{l_{\max}}$ are learnable parameters. $l_{\max}$ is the pre-defined maximum number of tokens in a variable name, which is derived from a statistical analysis of the different variable names present in the training dataset. d indicates the dimensionality of representations. $\mathbf{q}_s$ is the predicted probability distribution of the token number for the code snippet s.

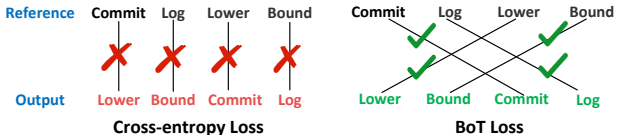

Figure 4: A comparison of cross-entropy loss and BoT loss.

We adopt the standard cross-entropy loss for predicting the number of tokens:

$$\mathcal{L}_{\text{num}} = -\sum_{s\in\mathcal{S}}\sum_{i=1}^{l_{\max}} r_{s,i}\log q_{s,i}, \tag{4}$$

where $r_{s,i}$ equals 1 if the masked variable name in s has i tokens otherwise 0. $q_{s,i}$ is the i-th dimension in $\mathbf{q}_{s,i}$, i.e., the predicted probability that the masked variable name has i tokens.

4.4.2 Stage 2: Generate tokens (TG task). In Stage 2, RefBERT generates tokens in refactored variable names.

Bag-of-token Loss for Guiding Token Generation. In many NLP applications, the generated sentence consisting of multiple words are expected to be similar to the target sentences in order to express the correct semantics. Word order deviations in sentences should yield excessively large losses since different word orders indicates different meanings or even violates the grammar rules of natural languages. Therefore, training a generative model with a cross-entropy loss is a standard scheme in NLP. The cross-entropy loss is computed strictly between aligned ground-truth tokens and predicted tokens.

However, in rename refactoring, the order of the generated tokens for the refactored variable name may vary. Due to different coding conventions, it is possible that different developers use different orders of tokens to express the same variable name. For example, for the ground-truth refactored variable name *defaultVersion*, *versionDefault* which contain same tokens *default* and *version* in a different order, expresses the same meaning. And the different token orders do not violate the rule of programming languages since *defaultVersion* and *versionDefault* are processed integrally as variable names instead of individual tokens by the compiler. However, if we solely adopt the cross-entropy loss to guide the generation of variable name tokens, *versionDefault* will be punished as the incorrect answer since ['version', 'default'] is not strictly aligned with ['default', 'version'], which potentially harms the training of the model.

Consequently, we consider loosening the strict restriction of token orders in the cross-entropy loss to guide the generation of variable name tokens by introducing a new loss. We adopt the idea of using bag-of-words (BoW) [68] in designing text generation losses [37, 41] and modify the cross-entropy loss to a *bag-of-tokens* (BoT) loss to reduce the punishment for token deviations in rename refactoring. BoW is a text representation where a text is represented as the bag (multiset) of its words, disregarding grammar, word order and word frequency. Similarly, when using BoT, we do not consider token order and token frequency in a variable name. For instance, variable names *CommitLogLowerBound* and *LowerBoundCommitLog* have the same BoT representation {*Commit*, *Log*, *Lower*, *Bound*}.

Fig. 4 illustrates the difference between the BoT loss and the cross-entropy loss.

The idea of the BoT loss is to compare the BoT representations of the generated variable names and the ground-truth, refactored variable name. This way, token order deviation will not incur punishment of large losses as long as the generated variable names have similar tokens as the ground-truth, refactored variable name. In the BoT loss, the probability distributions $\{\mathbf{p}_{s,1}, \cdots, \mathbf{p}_{s,i}\}$ $(1 \le i \le g_s^{(m)})$ of [MASK] tokens predicted by RefBERT for all positions are summed up to form the variable-name-level distribution $\mathbf{z}_s \in \mathbb{R}^{|V|}$ for the code snippet s:

$$\mathbf{z}_s = \sum_{i=1}^{g_s^{(m)}} sigmoid(\mathbf{p}_{s,i}), \tag{5}$$

where $sigmoid(\cdot)$ indicates the dimension-wise calculation using the sigmoid function. Each dimension in the variable-name-level distribution $\mathbf{z}_s$ represents how possible the corresponding token appears in the BoT for the refactored variable name in s regardless of the token order. Compared to the token-level probability distribution $\mathbf{p}_s$, the variable-name-level probability distribution $\mathbf{z}_s$ removes the restriction of the token order. Based on $\mathbf{z}_s$, we can calculate the BoT loss which guides token generation as follows:

$$\mathcal{L}_{\text{BoT}} = -\sum_{s \in \mathcal{S}} \sum_{i=1}^{g_s^{(m)}} \sum_{j \in \mathcal{V}} y_{s,i,j} \log z_{s,j}, \tag{6}$$

where $z_{s,j}$ is the predicted probability of the i-th masked token being j (i.e., i-th dimension in $\mathbf{z}_s$) and other notations have same meanings as Eq. 2.

Contrast Variable Names Before and After Refactoring. As discussed in Sec. 2.3, rename refactoring is different from the variable name prediction task since the variable name after refactoring is quite different compared to the name before refactoring. To fully utilize such a difference to improve rename refactoring, we opt for contrastive learning, a type of self-supervised learning that has recently gained significant momentum [38]. Conceptually, contrastive learning aims to learn a representation space by minimizing the distance between positive samples while maximizing the distance between negative samples. A suitable contrast task will facilitate capturing intrinsic features in the data without requiring additional labels [38].

For rename refactoring, in the context of contrastive learning, the representation of the variable name after refactoring should be as close as possible to the representation of the ground-truth refactored variable name (i.e., positive sample), and far away from the variable name before refactoring (i.e., negative sample), as depicted in the bottom-right part of Fig. 2. We perform the average pooling operation on the representations of all tokens in a variable name v to construct the representation $\hat{\mathbf{h}}_v$ of v:

$$\begin{aligned} \mathbf{h}_v &= \text{Avg}(\{\mathbf{h}^{(1)}, \cdots, \mathbf{h}^{(j)}\}),\ 1 \le j \le |v| \\ \hat{\mathbf{h}}_v &= \mathbf{h}_v / \|\mathbf{h}_v\|_2 \end{aligned} \tag{7}$$

where $|v|$ is the number of tokens in v, $\mathbf{h}^{(1)}$ is the representation of the j-th token in v, $\text{Avg}(\cdot)$ is the average pooling operation, and $\|\cdot\|_2$ denotes L2 normalization.

We adopt the non-parametric instance discrimination loss [66] in RefBERT to achieve the goal of contrastive learning:

$$\mathcal{L}_{CL} = -\sum_{i \in \mathcal{I}} \log \frac{e^{sim(\hat{\mathbf{h}}_i^{(\text{gen})}, \hat{\mathbf{h}}_i^{(\text{after})})/\tau}}{e^{sim(\hat{\mathbf{h}}_i^{(\text{gen})}, \hat{\mathbf{h}}_i^{(\text{after})})/\tau} + e^{sim(\hat{\mathbf{h}}_i^{(\text{gen})}, \hat{\mathbf{h}}_i^{(\text{before})})/\tau}}, \tag{8}$$

where $\mathcal{I}$ is the refactored variable name set, $sim(\cdot)$ denotes the cosine similarity and τ is the temperature hyper-parameter that affects distribution concentration [66]. $\hat{\mathbf{h}}^{(\text{gen})}$, $\hat{\mathbf{h}}^{(\text{before})}$ and $\hat{\mathbf{h}}^{(\text{after})}$ are representations of the predicted variable name, the variable name before refactoring and the variable name after refactoring, respectively.

Complete Loss in Stage 2. The complete training loss in Stage 2 can be defined as follows:

$$\mathcal{L}_{\text{fine-tune}} = \lambda_{\text{cMLM}} \cdot \mathcal{L}_{\text{cMLM}} + \lambda_{\text{BoT}} \cdot \mathcal{L}_{\text{BoT}} + \lambda_{\text{CL}} \cdot \mathcal{L}_{\text{CL}}, \tag{9}$$

where λ_{cMLM}, λ_{BoT} and λ_{CL} are hyper-parameters for loss weights. To balance the strict order consistency brought by the original cross-entropy loss for cMLM (Eq. 2) and the token order flexibility brought by the BoT loss (Eq. 6), during fine-tuning, $\mathcal{L}_{\text{cMLM}}$ is still kept. Note that, since calculating the BoT loss brings additional overhead and the pre-training corpora are much larger than the fine-tuning corpora, we do not adopt $\mathcal{L}_{\text{BoT}}$ in the pre-training phase.

4.5 Predict with RefBERT

At predication, for a variable name in a code snippet s to be refactored, we first use the token number predictor in RefBERT to predict the number of tokens in the new variable name after refactoring (i.e., $g_s^{(m)}$). After that, for each of the $g_s^{(m)}$ tokens in the refactored variable name, RefBERT predicts the token probability distribution $\mathbf{p}$, and it select the token from the token vocabulary $\mathcal{V}$ with the largest probability as the generated token. Then, all the generated tokens are concatenated as the predicted variable name. This way, RefBERT will not produce the same token multiple times for a variable name. Since a variable name typically does not contain many tokens and these tokens are usually different, our method can generate more readable refactored variable names, as shown in our experiments.

5 EXPERIMENT SETTINGS

This section illustrates the details of the experiment settings.

5.1 Data

We choose two public datasets that are commonly used in code representation learning to pre-train RefBERT:

- **CodeSearchNet**[5] [26]: It contains about 6 million functions from open-source GitHub repositories spanning 6 programming languages and is widely used in various code relevant tasks including but not limited to code retrieval [21], code completion [10] and code summarization [34]. We use the Java dataset in CodeSearchNet. We use the data split in the dataset for training, validation and test.

[5] https://github.com/github/CodeSearchNet

Table 1: Statistics of datasets.

Stage	Dataset	Training	Test	Validation
Pre-training	CodeSearchNet	410,050	24,607	11,004
	Code2Seq	418,438	50,480	50,993
Fine-tuning	JavaRef	14,319	1,801	1,790
	TL-CodeSum	50,908	7,243	7,173

- **code2seq**[6] [3]: It contains 11 large Java projects originally provided by Allamanis et al. [2]. We adopt its preprocessed version, Java-small [3], in our experiments. It contains about 700K examples. We use the data split in the dataset for training, validation and test.

For the fine-tuning phase, our constructed rename refactoring datasets JavaRef and TL-CodeSum illustrated in Sec. 3 are used on the rename refactoring task.

Tab. 1 provides the statistics of all the datasets in the experiment.

5.2 Environment

We implement RefBERT[7] using PyTorch and run the experiments on a machine with two Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz, 256 GB main memory and one NVIDIA GeForce RTX 3090.

5.3 Evaluation

RefBERT first predicts the number of tokens in the refactored variable name (the LP task), then it generates tokens in the refactored variable name accordingly (the TG task). Hence, our evaluations involve two parts:

5.3.1 Length Prediction (LP). The LP task corresponds to the first stage illustrated in Sec. 4.4.1.

Baseline: We adopt a heuristic-based method as the baseline for the LP task. It selected the predicted number with the largest average probability from RefBERT, i.e., maximum $\frac{1}{g_s^{(m)}} \sum_{i=1}^{g_s^{(m)}} \log p(t_{s,i}^{(m)} | T_s^{(u)}; \theta)$.

Metrics: We choose Hit@K as the evaluation metric. We evaluate whether the model predicts the ground-truth number as the most possible number (Hit@1) or among the top-3 most possible numbers (Hit@3) [19].

5.3.2 Token Generation (TG). The TG task corresponds to the second stage illustrated in Sec. 4.4.2.

Baseline: Naturalize [1] is the baseline used in the TG task. It leverages a n-gram language model to suggest identifier names, including variable names, to replace those that break coding conventions. It first retrieves a set of candidate names that have appeared in the similar context from other code snippets, and then ranks the candidates by measuring naturalness through a learned n-gram model.

[6] https://github.com/tech-srl/code2seq

[7] Our implementation and data are available at https://github.com/KDEGroup/RefBERT.

Table 2: Performance of the LP task.

Method	Dataset	JavaRef		TL-CodeSum	
		Hit@1	Hit@3	Hit@1	Hit@3
Heuristic-based Method	CodeSearchNet	0.527	0.817	0.543	0.828
	Code2Seq	0.506	0.782	0.529	0.792
RefBERT	CodeSearchNet	**0.655**	**0.946**	**0.702**	**0.963**
	Code2Seq	**0.675**	**0.946**	**0.727**	**0.960**

Metrics: We adopt various metrics for the TG task.

- **Accuracy**: It measures the percentage of tokens, which are contained in the ground-truth variable name, can be generated by the model in the prediction. Large accuracy indicates good results.
- **Exact Match (EM)** [5]**:** It measures the percentage of refactored variable names that can be exactly generated by the model. It is a variable-name-level metric and does not count the token-level similarity. For instance, the EM between "ImageFolder" and "ImgFolder" is 0. Large ED shows good results.
- **Edit Distance (ED)**: It compares the similarity of two strings by measuring the minimum number of editing operations w.r.t. tokens required to convert one string into another. For instance, the ED between "ImageFolder" and "ImgFolder" is 1. Small ED indicates good results.
- **Character Error Rate (CER)** [63]**:** It calculates the ED between the ground-truth variable name and the generated variable name. Then the result is normalized by the number of characters in the ground-truth variable name. Small CER indicates good results.

5.4 Hyper-parameters

We preserve the first 512 tokens in each code snippet at most. Following RoBERTa [40], for RefBERT, we sample initial weights from $\mathcal{N}(0, 0.02)$, initialize biases to zero, set dropout ratio to 0.1 for all layers, and use GELU [24] as activation function in all layers. For predicting the token number, $l_{\max}$ is set to 5 by default. We conduct grid search for λ and τ on the validation set. The default λ_{CL}, λ_{cMLM} and λ_{BoT} are 1, 1 and 0.1, respectively. The default τ is set to 0.05. We optimize RefBERT using Adam [31]. We terminate training when the method converges.

6 EXPERIMENTAL RESULTS

In this section, we report and analyze the experimental results to answer the following research questions (RQ):

- **RQ1:** Is RefBERT capable of predicting the correct number of tokens in the refactored variable name?
- **RQ2:** Can RefBERT generate high-quality tokens in refactored variable names for rename refactoring?
- **RQ3:** Does each component of RefBERT contribute to its performance?
- **RQ4:** Do different settings affect the performance of RefBERT?

Table 3: Performance of the TG task. The results of RefBERT are reported separately for pre-training on CodeSearchNet or Code2Seq.

Model	JavaRef				TL-CodeSum			
	Accuracy	EM	CER	ED	Accuracy	EM	CER	ED
Naturalize	0.035	0.029	336.356	10.793	0.067	0.059	242.757	10.762
RefBERT (CodeSearchNet)	0.581	0.524	57.897	3.185	0.576	0.530	58.521	2.991
RefBERT (Code2Seq)	0.584	0.537	55.122	3.168	0.572	0.531	54.851	2.938

Table 4: Results of the ablation study on two datasets for the TG task. All methods are pre-trained on CodeSearchNet. Best results are shown in bold.

Model	JavaRef				TL-CodeSum			
	Accuracy	EM	CER	ED	Accuracy	EM	CER	ED
RefBERT w/o BoT & CL	0.712	0.668	36.930	2.013	0.673	0.633	41.866	2.117
RefBERT w/o BoT	0.739	0.699	33.472	1.856	0.674	0.634	41.479	2.120
RefBERT w/o CL	0.740	0.697	31.817	1.804	0.692	0.653	39.836	2.015
RefBERT w/o cMLM	0.040	0.024	110.750	7.829	0.230	0.198	104.027	5.610
RefBERT	**0.750**	**0.711**	**31.164**	**1.775**	**0.700**	**0.662**	**37.834**	**1.975**

6.1 Performance of the LP Task (RQ1)

Tab. 2 shows the performance of the heuristic-based method and RefBERT on the LP task. From Tab. 2, we have several observations:

(1) On both JavaRef and TL-CodeSum, RefBERT outperforms the heuristic-based method by 0.13-0.20 on Hit@1. Overall, RefBERT can correctly predict the number of tokens in the refactored variable name with a high probability (0.65-0.72).
(2) The performance of RefBERT is around 0.94-0.96 on Hit@3 and it consistently exceeds the heuristic-based method by a large margin. The high Hit@3 of RefBERT shows that it is very likely for RefBERT to predict the appropriate token number with the near-the-top rank.

Based on the above observation, we can conclude that RefBERT is capable of predicting the correct number of tokens in the refactored variable name.

6.2 Performance of the TG Task (RQ2)

RefBERT uses the top-1 predicted number in the LP task to guide itself to generate the corresponding quantity of tokens in the TG task. Tab. 3 presents the performance of RefBERT and Naturalize on the TG task.

We can see that RefBERT significantly outperforms Naturalize on both JavaRef and TL-CodeSum. There are two possible reasons:

(1) Firstly, Naturalize only generates variable names that exist in the training data. Due to the sparsity issue of variable names shown in Fig. 1, it is very likely that the refactored variable name does not exist in the vocabulary of Naturalize.
(2) Besides, Naturalize offers variable-level rename refactoring and it does not capture the semantics of sub-tokens, which significantly affects its performance w.r.t. token-level metrics Accuracy, CER and ED. Differently, RefBERT is a token-level rename refactoring method and does not only generate existing variable names for rename refactoring. Hence, it shows much better results on the TG task.

In summary, we can conclude that RefBERT can generate high-quality tokens in refactored variable names for rename refactoring.

6.3 Ablation Study (RQ3)

Since the token number predictor (Sec. 4.4.1) used for the LP task is a simple single-layer feedforward neural network without additional components, we only conduct ablation study for the TG task. To investigate whether each component in RefBERT takes effect for the TG task, we investigate the performance of the following variants of RefBERT:

- **RefBERT w/o BoT & CL:** It only keeps $\mathcal{L}_{\text{cMLM}}$ in Eq. 9 during fine-tuning.
- **RefBERT w/o BoT:** It removes $\mathcal{L}_{\text{BoT}}$ in Eq. 9 during fine-tuning.
- **RefBERT w/o CL:** It removes $\mathcal{L}_{\text{CL}}$ in Eq. 9 during fine-tuning.
- **RefBERT w/o cMLM:** It removes $\mathcal{L}_{\text{CE}}$ in Eq. 9 during fine-tuning.

Tab. 4 reports the results of the ablation study. All methods are pre-trained on CodeSearchNet. All methods generate tokens according to the ground-truth token number in the variable name. From the results, we have the following observations:

- We can see that RefBERT w/o cMLM shows the worst performance among all variations of RefBERT, meaning that completely removing the cross-entropy loss will significantly downgrade the performance on the TG task. Hence, we keep $\mathcal{L}_{\text{cMLM}}$ in Eq. 9 to balance the strict order consistency

brought by $\mathcal{L}_{\text{cMLM}}$ and the token order flexibility brought by $\mathcal{L}_{\text{BoT}}$.
- All four variations of RefBERT show worse performance than RefBERT, showing the effectiveness of each component in RefBERT. Moreover, RefBERT w/o BoT & CL performs worse than RefBERT w/o BoT and RefBERT w/o CL, which further demonstrates that both the BoT loss and contrastive learning play important roles in RefBERT.

Case Study. We provide 6 cases of rename refactoring on JavaRef in Fig. 5 as case study to illustrate the superiority of using the BoT loss over solely using the standard cross-entropy loss $\mathcal{L}_{\text{cMLM}}$ in Eq. 9 on the rename refactoring task. We observe that, if results using BoT are of low quality, results using $\mathcal{L}_{\text{cMLM}}$ only are also of inferior quality. Hence, we selected six cases where RefBERT produces consistent results as ground truth, in three categories (more meaningful, abbreviated, and non-repetitive). From Fig. 5, we can see that RefBERT using the BoT loss can generate more accurate variable names:

- RefBERT can produce more meaningful tokens according to the context of the variable name. For instance, in Example 2, *count* generated by using the BoT loss is a more meaningful variable name than *i* generated by using the cross-entropy loss.
- Moreover, using the BoT loss helps generate abbreviated variable names (e.g., *Dir* for *Directory* in Example 1 and *repr* for *Representation* in Example 4). The reason is that the BoT loss cares the global probability of a token when generating tokens. Assume *backupsDir* is split into *backups* and *Dir*. At step 1, the probability of *backups* is highest followed by *Dir*, while the probability of *Directory* is quite low. At step 2, the probability of *Directory* is highest followed by *Dir* and *backups*. Using the cross-entropy loss, the model is trained to generate the token at each step with the highest probability and hence it yields *backupsDirectory*. Differently, using the BoT loss, the model finds the global probability (i.e., consider probabilities for all steps) of *Dir* and *backups* are higher than *Directory*. As the predicted token number is 2, the model generates *backupsDir* as the output. This may be a desirable merit for the case where developers prefer abbreviations.
- Using the BoT loss, RefBERT can avoid generating duplicate tokens. For example, solely using cross-entropy loss, RefBERT produces *fieldFieldValue* in Example 6. As a comparison, using the BoT loss, the generated variable name is *patternFieldValue*. The reason is that no duplicated tokens are recorded in the bag-of-tokens representation, and RefBERT generates $g_s^{(m)}$ unique tokens from the token vocabulary, as illustrated in Sec. 4.5.

Example 1 Ground-truth refactored name : backupsDir
BoT: backupsDir cMLM: backupsDirectory

```
public void maybeIncrementallyBackup (final Iterable<SSTableReader> sstables) {
  if (!DatabaseDescriptor.isIncrementalBackupsEnabled ()) return;
  for (SSTableReader sstable : sstables) {
    File ? = Directories.getBackupsDirectory (sstable.descriptor) ;
    sstable.createLinks(FileUtils.getCanonicalPath (?)) ;
  }
}
```

Example 2 Ground-truth refactored name : count
BoT: count cMLM: i

```
public Void apply (Iterable<Row> input) {
  int ? = 0;
  for (Row row : input) { ?++ ; }
  Assert.assertEquals(size, ?);
}
```

Example 3 Ground-truth refactored name : pendingElement
BoT: pendingElement cMLM: windPElement

```
private synchronized void addPendingElements(Iterable<? extends WindowedValue
<?>> newPending) {
  for(WindowedValue <?> ? : newPending) {
  pendingElements.add(?);
  }
}
```

Example 4 Ground-truth refactored name : repr
BoT: repr cMLM: reation

```
public void selfIsFollowable() throws Exception {
  CapabilitiesRepresentation ? = givenRepresentation();
  assertThat(?, isFollowableLinkToSelf(client));
}
```

Example 5 Ground-truth refactored name : embedsThumbnail
BoT: embedsThumbnail cMLM: embededsThumbnail

```
public CGImageDestinationProperties setEmbedsThumbnail(boolean ?){
  set(Keys.EmbedThumbnail(), CFBoolean.valueOf(?));
  return this;
}
```

Example 6 Ground-truth refactored name : patternFieldValue
BoT: patternFieldValue cMLM: fieldFieldValue

```
protected boolean matchesPattern (ObjectData patternData, ObjectData testData) {
  Enumeration fields = patternData.fields();
  while(fields.hasMoreElements()) {
  String field = (String)fields.nextElement();
  Object ? = patternData.get(field);
  Object testFieldValue = testData.get(field);
  if(testFieldValue instanceof ReferenceVector){
  ReferenceVector patternElements = (ReferenceVector) ? ;
    ...
  }else{
    if(!?.equals(testFieldValue)){
      return false;
    }
  ...
```

Figure 5: Case study for comparing using BoT loss and solely using the cross-entropy loss. Question marks indicate target variables of rename refactoring. Correct tokens are shown in yellow.

6.4 Effects of Different Settings (RQ4)

Impact of l_{max}. As explained in Sec. 4.4.1, we formulate the prediction of the number of tokens in the refactored variable name as a multi-label classification task and the number of the labels is decided by the pre-defined maximum token number. In Fig. 6, we provide the distributions of the number of tokens in variable names in JavaRef and TL-CodeSum. From Fig. 6, we can observe that more than 40% of variable names contain only one token, more than 70% of variable names contain no more than 2 tokens, and all variable names have no more than 10 tokens. To investigate the impact of using different l_{max} on the TG task, we test RefBERT with different l_{max} in {1, 3, 5, 10, 15, 20} and report the results on JavaRef in Fig. 7. From the results, we can see that the change trends of performance w.r.t. accuracy and exact match are mostly consistent on JavaRef. Pre-trained RefBERT on CodeSearchNet

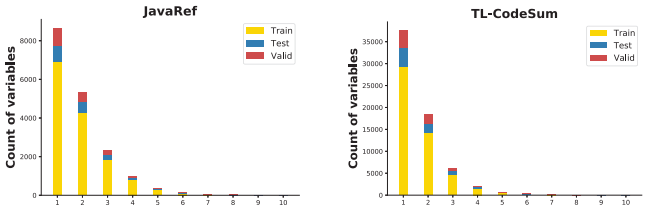

Figure 6: Distributions of the number of tokens in variable names.

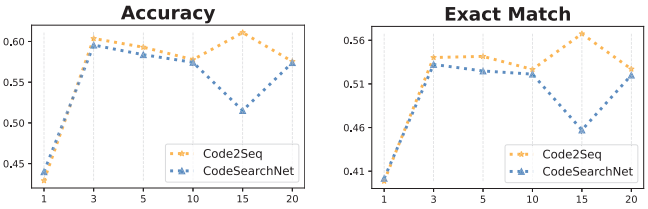

Figure 7: Effects of l_{max} on the performance of RefBERT on JavaRef. Orange dotted line and blue dotted line indicate pre-trained RefBERT on Code2Seq and CodeSearchNet, respectively.

Table 5: Impact of λ

λ_{CL}	λ_{cMLM}	λ_{BoT}	Accuracy
1	1	0	0.738
1	0	1	0.040
0.5	0.2	0.1	0.731
0.5	1	0.1	0.740
1	**1**	**0.1**	**0.750**
1	0.5	0.1	0.745

Table 6: Impact of τ

τ	Accuracy
0.02	0.741
0.05	**0.750**
0.10	0.743
0.15	0.737
0.20	0.741
0.25	0.736

shows the best result when $l_{max} = 3$ or $l_{max} = 5$. Although pre-trained RefBERT on Code2Seq shows best result when $l_{max} = 15$, most variable names do not contain many tokens as demonstrated in Fig. 6. Therefore, our default setting $l_{max} = 5$ is an appropriate choice in practice.

Impact of λ. Tab. 5 reports the results using different λ_{CL}, λ_{cMLM} and λ_{BoT} in Eq. 9 on JavaRef for the TG task and RefBERT is pre-trained on CodeSearchNet. The performance of the default setting is shown in bold. We can see that, completely removing the cross-entropy loss ($\lambda_{\text{cMLM}} = 0$) significantly affects RefBERT on the TG task, as already shown in the ablation study (Sec. 6.3). The result also provides proof for our previous idea of retaining $\mathcal{L}_{\text{cMLM}}$ while introducing $\mathcal{L}_{\text{BoT}}$ in Sec. 4.4.2. Moreover, from Tab. 5, we can conclude that RefBERT consistently shows ~0.7 for accuracy on the TG task and our default setting yield the best result.

Impact of τ. Tab. 6 provides the results using different τ in Eq. 8 on JavaRef for the TG task and RefBERT is pre-trained on CodeSearchNet. The performance of the default setting is shown in bold. We can observe that the accuracy of RefBERT is not sensitive to the choice of τ. Hence, it is easy to tune τ.

7 THREATS TO VALIDITY

Three threats may affect the validity of our study:

Oracle Bias. We construct JavaRef using RefactoringMiner, which may fail to identify some refactorings. Moreover, using the refactoring detection tool, oracles may be biased towards refactoring instances that are easy for the tool to discover. It means, if multiple refactoring operations are performed on a code snippet at the same time, some refactoring types may be neglected as the tool may not detect all the instances. To mitigate oracle bias, we ask students with professional background to manually check the detected data and correct errors.

False Oracles. We adapt an existing dataset TL-CodeSum and use it for rename refactoring. We assume existing variable names in TL-CodeSum are meaningful and do not require rename refactoring. This may be true for most existing variable names, but some variable names may not be meaningful and require refactoring. Thus, TL-CodeSum may contain false oracles. However, as the primary challenge for our task is the lack of data, we have to adapt TL-CodeSum in addition to JavaRef and hold such an assumption. As we are keeping enriching JavaRef, we expect JavaRef will be sufficient for the future study.

Metrics. To our best knowledge, there is no universally acknowledged metric for evaluating rename refactoring. We choose several prevalent metrics from information retrieval and natural language processing domains for evaluation. But there exists a threat that using one metric may not fairly evaluate the generated variable names. Hence, we suggest considering all the metrics together to get a full picture of the performance of rename refactoring.

8 CONCLUSION

Automatic rename refactoring reduces the intellectual burden of developers and improves the readability of programs. However, few existing works express concern on how to suggest a new name for rename refactoring on variable names. In this paper, we study automatic rename refactoring and particularly focus on refactoring variable names that are more difficult to refactor compared to other identifiers. Based on our observations on rename refactoring, we propose RefBERT, a two-stage pre-trained framework tailored for rename refactoring. Experimental results demonstrate the effectiveness of RefBERT. In the future, we will explore how we can better determine the order of the generated tokens for constructing the refactored name. Moreover, we will continue to construct and improve our refactoring dataset JavaRef to alleviate oracle bias and reduce false oracles.

To enhance reproducibility and replicability, we provide our implementation and data at https://github.com/KDEGroup/RefBERT.

ACKNOWLEDGMENTS

This work was partially supported by National Key R&D Program of China (No. 2022ZD0118201), National Natural Science Foundation of China (No. 62002303, 42171456), Natural Science Foundation of Fujian Province of China (No. 2020J05001) and CCF-Tencent Open Fund.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *SIGSOFT FSE*. 281–293. https://doi.org/10.1145/2635868.2635883

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *ICML*, Vol. 48. 2091–2100.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR (Poster)*.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *PLDI*. 404–419. https://doi.org/10.1145/3192366.3192412

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[6] Venera Arnaoudova, Laleh Mousavi Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2014. REPENT: Analyzing the Nature of Identifier Renamings. *IEEE Trans. Software Eng.* 40, 5 (2014), 502–532. https://doi.org/10.1109/TSE.2014.2312942

[7] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146. https://doi.org/10.1162/tacl_a_00051

[8] Bruno Caprile and Paolo Tonella. 2000. Restructuring Program Identifier Names. In *ICSM*. 97–107. https://doi.org/10.1109/ICSM.2000.883022

[9] Zimin Chen and Martin Monperrus. 2019. A Literature Study of Embeddings on Source Code. *arXiv Preprint* (2019). https://arxiv.org/abs/1904.03061

[10] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. In *MSR*. 108–119. https://doi.org/10.1109/MSR52588.2021.00024

[11] Maëlick Claes and Mika V. Mäntylä. 2020. 20-MAD: 20 Years of Issues and Commits of Mozilla and Apache Development. In *MSR*. 503–507. https://doi.org/10.1145/3379597.3387487

[12] Florian Deißenböck and Markus Pizka. 2005. Concise and Consistent Naming. In *IWPC*. 97–106. https://doi.org/10.1109/WPC.2005.14

[13] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2000. Finding refactorings via change metrics. In *OOPSLA*. 166–177. https://doi.org/10.1145/353171.353183

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[15] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph E. Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *ECOOP*, Vol. 4067. 404–428. https://doi.org/10.1007/11785477_24

[16] Asger Feldthaus and Anders Møller. 2013. Semi-automatic rename refactoring for JavaScript. In *OOPSLA*. 323–338. https://doi.org/10.1145/2509136.2509520

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP (Findings)*, Vol. EMNLP 2020. 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[18] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley. https://doi.org/10.1007/3-540-45672-4_31

[19] Marjan Ghazvininejad, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. Mask-Predict: Parallel Decoding of Conditional Masked Language Models. In *EMNLP/IJCNLP (1)*. 6111–6120. https://doi.org/10.18653/v1/D19-1633

[20] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One thousand and one stories: a large-scale survey of software refactoring. In *ESEC/SIGSOFT FSE*. 1303–1313. https://doi.org/10.1145/3468264.3473924

[21] Jian Gu, Zimin Chen, and Martin Monperrus. 2021. Multimodal Representation for Neural Code Search. In *ICSME*. 483–494. https://doi.org/10.1109/ICSME52107.2021.00049

[22] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *ACL (1)*. 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[23] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*.

[24] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv Preprint* (2016). https://arxiv.org/abs/1606.08415

[25] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.* 25, 3 (2020), 2179–2217. https://doi.org/10.1145/3368089.3417926

[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv Preprint* (2019). https://arxiv.org/abs/1909.09436

[27] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. 2020. SpanBERT: Improving Pre-training by Representing and Predicting Spans. *Trans. Assoc. Comput. Linguistics* 8 (2020), 64–77. https://doi.org/10.1162/tacl_a_00300

[28] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pre-trained Contextual Embedding of Source Code. *arXiv Preprint* (2020). https://arxiv.org/abs/2001.00059

[29] Rafael-Michael Karampatsis and Charles Sutton. 2020. SCELMo: Source Code Embeddings from Language Models. *arXiv Preprint* (2020). https://arxiv.org/abs/2004.13214

[30] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-Finder: a refactoring reconstruction tool based on logic query templates. In *SIGSOFT FSE*. 371–372. https://doi.org/10.1145/1882291.1882353

[31] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*. http://arxiv.org/abs/1412.6980

[32] Triet Huynh Minh Le, Hao Chen, and Muhammad Ali Babar. 2021. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3 (2021), 62:1–62:38. https://doi.org/10.1145/3383458

[33] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, Gabriele Bavota, and Michele Lanza. 2017. Investigating the Use of Code Analysis and NLP to Promote a Consistent Usage of Identifiers. In *SCAM*. 81–90. https://doi.org/10.1109/SCAM.2017.17

[34] Chen Lin, Zhichao Ouyang, Junqing Zhuang, Jianqiang Chen, Hui Li, and Rongxin Wu. 2021. Improving Code Summarization with Block-wise Abstract Syntax Tree Splitting. In *ICPC*. 184–195. https://doi.org/10.1109/ICPC52881.2021.00026

[35] Fang Liu, Ge Li, Zhiyi Fu, Shuai Lu, Yiyang Hao, and Zhi Jin. 2022. Learning to Recommend Method Names with Global Context. In *ICSE*. 1294–1306. https://doi.org/10.1145/3510003.3510154

[36] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *ICSE*. 1–12. https://doi.org/10.1109/ICSE.2019.00019

[37] Qing Liu, Lei Chen, Yuan Yuan, and Huarui Wu. 2021. History Reuse and Bag-of-Words Loss for Long Summary Generation. *IEEE ACM Trans. Audio Speech Lang. Process.* 29 (2021), 2551–2560. https://doi.org/10.1109/TASLP.2021.3100281

[38] Xiao Liu, Fanjin Zhang, Zhenyu Hou, Li Mian, Zhaoyu Wang, Jing Zhang, and Jie Tang. 2023. Self-Supervised Learning: Generative or Contrastive. *IEEE Trans. Knowl. Data Eng.* 35, 1 (2023), 857–876. https://doi.org/10.1109/TKDE.2021.3090866

[39] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv Preprint* (2019). https://arxiv.org/abs/1907.11692

[40] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv Preprint* (2019). https://arxiv.org/abs/1907.11692

[41] Shuming Ma, Xu Sun, Yizhong Wang, and Junyang Lin. 2018. Bag-of-Words as Target for Neural Machine Translation. In *ACL (2)*. 332–338. https://doi.org/10.18653/v1/P18-2053

[42] Guido Malpohl, James J. Hunt, and Walter F. Tichy. 2003. Renaming Detection. *Autom. Softw. Eng.* 10, 2 (2003), 183–202. https://doi.org/10.1109/MICRO56248.2022.00061

[43] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *ICSE*. 336–347. https://doi.org/10.1109/ICSE43902.2021.00041

[44] Philip Mayer and Andreas Schroeder. 2014. Automated Multi-Language Artifact Binding and Rename Refactoring between Java and DSLs Used by Java Frameworks. In *ECOOP*, Vol. 8586. 437–462. https://doi.org/10.1007/978-3-662-44202-9_18

[45] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*. 3111–3119.

[46] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How Are Java Software Developers Using the Eclipse IDE? *IEEE Softw.* 23, 4 (2006), 76–83. https://doi.org/10.1109/MS.2006.105

[47] Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Trans. Software Eng.* 38, 1, 5–18. https://doi.org/10.1109/TSE.2011.41

[48] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP*, Vol. 7920. 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[49] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. SPT-Code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations. In *ICSE*. 1–13. https://doi.org/10.1145/3510003.3510096

[50] Saeed Parsa, Morteza Zakeri Nasrabadi, Masoud Ekhtiarzadeh, and Mohammad Ramezani. 2023. Method name recommendation based on source code metrics. *J. Comput. Lang.* 74 (2023), 101177. https://doi.org/10.1016/j.cola.2022.101177
[51] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-based reconstruction of complex refactorings. In *ICSM*. 1–10. https://doi.org/10.1109/ICSM.2010.5609577
[52] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the relation of refactorings and software defect prediction. In *MSR*. 35–38. https://doi.org/10.1007/11785477_24
[53] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2019. Predicting program properties from 'big code'. *Commun. ACM* 62, 3 (2019), 99–107. https://doi.org/10.1145/3306204
[54] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *ACL (1)*. https://doi.org/10.18653/v1/p16-1162
[55] Danilo Silva, Nikolaos Tsantalis, and Marco Túlio Valente. 2016. Why we refactor? confessions of GitHub contributors. In *SIGSOFT FSE*. 858–870. https://doi.org/10.1145/2950290.2950305
[56] Danilo Silva and Marco Túlio Valente. 2017. RefDiff: detecting refactorings in version histories. In *MSR*. 269–279. https://doi.org/10.1109/MSR.2017.14
[57] Yu Sun, Shuohuan Wang, Yu-Kun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. 2019. ERNIE: Enhanced Representation through Knowledge Integration. *arXiv Preprint* (2019). https://arxiv.org/abs/1904.09223
[58] Andreas Thies and Christian Roth. 2010. Recommending rename refactorings. In *RSSE@ICSE*. 1–5. https://doi.org/10.1145/1808920.1808921
[59] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Trans. Software Eng.* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722
[60] Nikolaos Tsantalis, Matin Mansouri, Laleh Mousavi Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *ICSE*. 483–494. https://doi.org/10.1145/3180155.3180206
[61] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *ESEC/SIGSOFT FSE*. 683–693. https://doi.org/10.1145/3106237.3106289
[62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.
[63] Weiyue Wang, Jan-Thorsten Peter, Hendrik Rosendahl, and Hermann Ney. 2016. CharacTer: Translation Edit Rate on Character Level. In *WMT*. 505–510. https://doi.org/10.18653/v1/w16-2342
[64] Yanlin Wang and Hui Li. 2021. Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs. In *AAAI*. 14015–14023.
[65] Peter Weißgerber and Stephan Diehl. 2006. Identifying Refactorings from Source-Code Changes. In *ASE*. 231–240. https://doi.org/10.1109/ASE.2006.41
[66] Zhirong Wu, Yuanjun Xiong, Stella X. Yu, and Dahua Lin. 2018. Unsupervised Feature Learning via Non-Parametric Instance Discrimination. In *CVPR*. 3733–3742. https://doi.org/10.1109/CVPR.2018.00393
[67] Yanming Yang, Xin Xia, David Lo, and John C. Grundy. 2022. A Survey on Deep Learning for Software Engineering. *ACM Comput. Surv.* 54, 10s (2022), 206:1–206:73. 10.1145/3505243
[68] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. 2010. Understanding bag-of-words model: a statistical framework. *Int. J. Mach. Learn. Cybern.* 1, 1-4 (2010), 43–52. https://doi.org/10.1007/s13042-010-0001-0

Received 2023-02-16; accepted 2023-05-03