# Traceability Transformed: Generating more Accurate Links with Pre-Trained BERT Models

Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, Jane Cleland-Huang
*Computer Science And Engineering*
*University Of Notre Dame*
Notre Dame, IN, USA
jlin6, yliu26, qzeng, mjiang2, janehuang@nd.edu

***Abstract*—Software traceability establishes and leverages associations between diverse development artifacts. Researchers have proposed the use of deep learning trace models to link natural language artifacts, such as requirements and issue descriptions, to source code; however, their effectiveness has been restricted by availability of labeled data and efficiency at runtime. In this study, we propose a novel framework called Trace BERT (T-BERT) to generate trace links between source code and natural language artifacts. To address data sparsity, we leverage a three-step training strategy to enable trace models to transfer knowledge from a closely related Software Engineering challenge, which has a rich dataset, to produce trace links with much higher accuracy than has previously been achieved. We then apply the T-BERT framework to recover links between issues and commits in Open Source Projects. We comparatively evaluated accuracy and efficiency of three BERT architectures. Results show that a Single-BERT architecture generated the most accurate links, while a Siamese-BERT architecture produced comparable results with significantly less execution time. Furthermore, by learning and transferring knowledge, all three models in the framework outperform classical IR trace models. On the three evaluated real-word OSS projects, the best T-BERT stably outperformed the VSM model with average improvements of 60.31% measured using Mean Average Precision (MAP). RNN severely underperformed on these projects due to insufficient training data, while T-BERT overcame this problem by using pretrained language models and transfer learning.**

***Index Terms*—Software traceability, deep learning, language models**

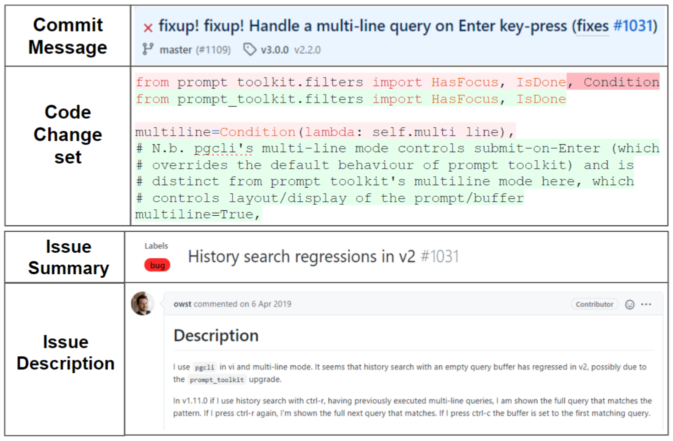
## I. Introduction

Software and systems traceability, is the ability to create and maintain relations between software artifacts and to leverage the resulting network of links to support queries about the product and its development process. Traceability is deemed essential in safety-critical systems where it is prescribed by certifying bodies such as the USA Federal Aviation Administration (FAA), USA Food and Drug Administration (FAA) [1]. When present, trace links support diverse software engineering activities such as impact analysis, compliance validation, and safety assurance. Unfortunately, in practice, the cost and effort of manually creating and maintaining trace links can be inhibitive, and therefore trace links are typically incomplete and inaccurate [2]. As a result, traceability data is often not trusted by developers and is often greatly underutilized.

Software artifacts, such as requirements, design definitions, code, and test cases all include natural language text, and therefore over the past decades, researchers have explored a wide variety of automated approaches for generating and evolving links automatically. Techniques have included probabilistic techniques [3], the Vector Space Model (VSM), [4], Latent Semantic Indexing [5], [6], Latent Dirichlet Allocation (LDA) [7], [8], AI swarm techniques [9], recurrent neural networks [10] to integrate semantics, heuristic approaches [11]–[13], combinations of techniques [7], [14], [15], and the use of decision trees and support vector machines [16] to integrate temporal dependencies and other process-related information into the tracing task. Despite all of these efforts, the accuracy of generated trace links has been unacceptably low, and therefore industry has been reticent to integrate automated tracing solutions into their development life-cycles. The primary impedance is a semantic one as most existing techniques rely upon word matching – either direct matches (e.g., VSM), topic-based matches (e.g., using LSI or LDA), or indirect matches based on building a domain-specific ontology to bridge the terminology gap [17]. Results have been mixed, especially when applied to industrial-sized datasets, where acceptable recall levels above 90% can often only be achieved at extremely low levels of precision [18].

One of the primary reasons that automated approaches have underperformed is the semantic gap that often exists between related artifacts [10]. Techniques that are unable to reason about semantic associations and bridge this gap fail to establish accurate and relatively complete trace links. Recent work has proposed deep learning (DL) techniques [19], [20] for traceability, but without providing effective solutions. For example, Guo et al. [10] proposed an architecture based on a Recurrent Neural Network (RNN) , and evaluated two types of RNN tracing models (LSTM [21] and GRU [22]) for generating links between subsystem requirements and design definitions against a small dataset from an industrial project. While their results showed that accuracy improved as the size of the training set increased, their approach was not trained on large training sets, and therefore was not shown to generalize across larger or more diverse projects. We include both LSTM and GRU approaches for comparison purposes and refer to them collectively as TraceNN (TNN) in this paper.

Two primary factors impede the advancement of DL traceability solutions. The first is the sparsity of training data, given that DL techniques require large volumes of training data. Manually created trace links (i.e., golden answer sets) available in individual software projects are usually not sufficient for training a DL model. The second impedance is the practicality of applying multi-layer neural networks in a large industrial project as training and utilizing deep neural networks is significantly slower than more traditional information retrieval or machine learning techniques.



**Commit Message**

fixup! fixup! Handle a multi-line query on Enter key-press (fixes #1031)
master (#1109) v3.0.0 v2.2.0

**Code Change set**

```
from prompt toolkit.filters import HasFocus, IsDone, Condition
from prompt_toolkit.filters import HasFocus, IsDone

multiline=Condition(lambda: self.multi line),
# N.b. pgcli's multi-line mode controls submit-on-Enter (which
# overrides the default behaviour of prompt toolkit) and is
# distinct from prompt toolkit's multiline mode here, which
# controls layout/display of the prompt/buffer
multiline=True,
```

**Issue Summary**

Labels bug History search regressions in v2 #1031

**Issue Description**

owst commented on 6 Apr 2019 Contributor

Description

I use `pgcli` in vi and multi-line mode. It seems that history search with an empty query buffer has regressed in v2, possibly due to the `prompt_toolkit` upgrade.

In v1.11.0 if I use history search with ctrl-r, having previously executed multi-line queries, I am shown the full query that matches the pattern. If I press ctrl-r again, I'm shown the full next query that matches. If I press ctrl-c the buffer is set to the first matching query.

Fig. 1: An example commit message and code change set, where green lines have been added and red ones removed. The commit was tagged by the committer to the depicted issue.

The work reported in this paper addresses these two critical impedances in order to deliver fast and accurate automated traceability solutions for solving industrial problems. More specifically, our proposed Language Model (LM) approach to traceability is designed to (1) deliver accurate and therefore more trustworthy trace links, (2) be applicable for projects with limited training data, and (3) scale-up to support large industrial projects with low time complexity. Our approach leverages BERT (**B**idirectional **E**ncoder **R**epresentations from **T**ransformers) as its underlying language models. BERT, which was introduced by Google in 2018 [23], has delivered marked improvements in diverse NLP tasks, primarily because its bidirectional approach provides deeper contextual information than single-direction language models. In this paper, we explore the use of BERT in the traceability domain – introducing what we refer to as *Trace BERT* (T-BERT).

T-BERT is a framework for training a BERT-based relationship classifier to generate trace links. Three types of relation classification architectures are particularly well suited for traceability. These are the single, twin and, siamese architectures, which we describe in more depth later in the paper. We compare the effectiveness of these three architectures for generating trace links between Natural Language Artifacts (NLAs) and Programming Language Artifacts (PLAs). NLAs are artifacts such as feature requests, bug reports, requirements, and design definitions, which are written primarily using natural language but may also include code snippets. In contrast, PLAs are primarily programming language artifacts, such as code files, code snippets, function definitions, and code change sets, which also contain natural language comments and descriptors. We evaluated T-BERT by generating trace links from issues to code (represented by change sets), which we refer to as a ***NLA-PLA traceability challenge***.

The remainder of this paper is laid out as follows. Sec. II outlines the concrete research questions we address in this paper. Sec. III and Sec. IV provide a detailed description of our approach for achieving NLA-PLA traceability, while Sec. V describes the experiments we conducted to evaluate the effectiveness of our approach. Based on the results obtained from these experiments, we derive answers for our research questions in Sec. VI. Finally, Sec. VII to Sec. IX discuss related work, threats to validity, and conclusions.

## II. Problem Statement

Researchers have addressed the data sparsity problem and the performance issues of training large models through the use of pre-trained DL models for various NLP problems . This approach divides the training stage into pre-training and fine-tuning phases. In the pre-training phase, DL models are constructed using a huge amount of unlabeled data and self-supervised training tasks. Then in the fine-tuning phase, the models are trained on smaller, labeled datasets in order to perform more specialized 'downstream' tasks. The underlying notion is that knowledge learned from pre-training a model on a larger and more generalized dataset can be effectively transferred to the downstream tasks which have limited labels for supervised training. Furthermore, a pre-trained model provides a better starting point for model optimization than a randomly seeded one. It therefore reduces the likelihood of local optimization traps and improves overall performance. Fine-tuning a pre-trained model on a smaller dataset takes significantly less time than training a deep learning model from scratch. While pre-training a general model is extremely expensive, the pre-training phase only needs to be performed once and can then be reused for various downstream tasks.

BERT-based language models make use of transformers [24] to learn contextual information from corpora in the pre-training stage and then transfer learned knowledge to downstream NLP tasks, such as question answering, document classification, and sentiment recognition [23], [25]. To our knowledge, this is the first study that has applied BERT or other transformer-based methods to the software traceability task. We pose a series of research questions to evaluate whether T-BERT can effectively address the traceability problem. Our first question is defined as follows:

**RQ1**: Given three variants of T-BERT models, based on single, twin, and siamese BERT relation classifiers, which is the best architecture for addressing NLA-PLA traceability with respect to both accuracy and efficiency?

In addition to investigating the DL model architecture, we also explore different training techniques for improving model accuracy. As discussed by Guo et al. [10], the DL trace model may hit a 'performance glass ceiling' and converge at relatively low accuracy. We therefore define our second research question as:

**RQ2**: Which training technique improves accuracy without suffering from the previously observed glass ceiling ?

Gururangan et al., in their study of Domain-Adaptive Pre-Training (DAPT), claim that a second phase of pre-training using a domain corpus leads to performance gains. This finding motivated us to explore the third and most important research question:

**RQ3**: Can T-BERT transfer knowledge from a resource-rich retrieval task to enhance the accuracy and performance of the downstream NLA-PLA tracing challenge?

Feng et al. [26] demonstrated that a BERT Language Model, pre-trained using large numbers of function definitions, can effectively address the downstream code search problem. In that study, researchers provided doc-strings (i.e., python comments) as user queries, and leveraged a BERT model to retrieve related functions. Since doc-strings and functions are always paired in the code base, ample training data for the code search problem is available. Our RQ3 explores whether the code search problem can be leveraged as a training task to improve T-BERT for the software traceability challenge. Because this step occurs between pre-training and fine-tuning, we refer to it as *intermediate-training*

## III. Approach

Trace retrieval algorithms dynamically generate trace links between artifacts [27], for example, by linking a source artifact (e.g., a python file) to a target artifact (e.g., an issue or requirement). The traceability algorithm computes the relevance between pairs of source and target artifacts and proposes the most related pairs as trace links. In this section, we first introduce the fundamental architecture of the BERT-based model and its variances, and then introduce T-BERT with three specific relation classifiers that are well suited for addressing this traceability problem.

### A. Introduction to BERT and Language Models

A language model represents a probability distribution over a word sequence [28], which with proper training can effectively capture the semantics of individual words based on their surrounding context. Given the importance of general context, DL models built upon pre-trained language models usually achieve better results than those trained on task-specific datasets directly. The architecture of the BERT-based model is based on transformers, in which each layer in the model is a transformer layer. The transformer layers allow the BERT model to focus on terms at any position in a sentence, and training a BERT model is accomplished through a novel technique called Masked Language Modeling (MLM). In a MLM training task, BERT randomly masks the words in the input text and then optimizes itself to predict the masked terms based on the contextual information. In this pre-training step, a massive amount of corpora are fed to the BERT-based model, and the resulting model is leveraged to address different downstream tasks by fine-tuning on task-specific datasets. A distinctive feature of BERT is its unified architecture across different tasks [23], as the architecture for LM pre-training and task-specific fine-tuning are almost identical, with only the last layer of the model customized according to the targeted downstream tasks. This layer is usually referred to as a ***task header*** in a BERT-based model.

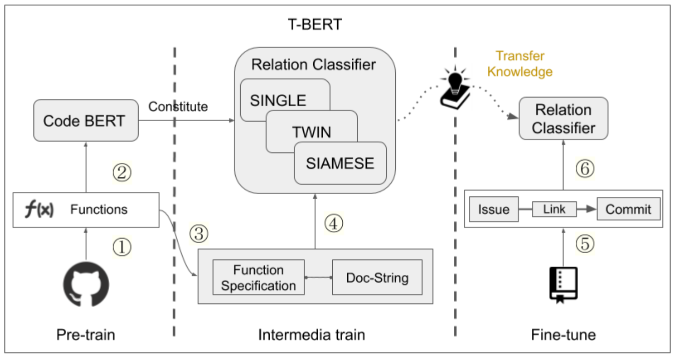


Fig. 2: A three step workflow applies T-BERT to NLA-PLA traceability. 1) Pre-training data are functions collected from Github projects 2) A BERT is trained as a language model for code with these functions and composed with a relation classifier as the T-BERT model 3) Functions are split as specifications and doc-strings and used as intermediate-training data 4) T-BERT model is intermediate-trained using code search data 5) OSS datasets are collected from Github repo 6) T-BERT model is fine-tuned as a trace model using transferred knowledge

### B. BERT For Software Traceability

The solution we propose represents a three-fold procedure of pre-training, intermediate-training and fine-tuning, as summarized in Fig. 2. In the pre-training phase, a dedicated language model is trained on source code and then utilized to construct the T-BERT models. In the intermediate-training phase, T-BERT is then trained to address the code search problem. In this phase, we provide adequate labeled training examples to T-BERT and expect it to learn general NL-PL classification knowledge that can ultimately be transferred to the traceability challenge. Finally, in the fine-tuning phase, the intermediate-trained T-BERT model is applied to the issue-commit tracing challenge in real-world open-source projects.

### C. T-BERT Architectures

The three variants of the T-BERT architecture that we investigate for software traceability have previously been applied to similar text-based problems. These variants are:

• **TWIN:** The Twin BERT architecture is shown in Fig. 3a. It leverages two BERT models to encode the NL and PL artifacts separately. The two artifacts are then transformed into two independent hidden state matrices, in which tokens are represented by fixed length vectors. We applied a pooling technique on these hidden state matrices to formulate feature vectors representing the artifacts. Finally, we concatenated these two feature vectors for classification tasks.

• **SIAMESE:** The siamese BERT architecture is shown in Fig. 3b. It is a hybrid of the single and twin architecture. It only uses one BERT model; however, instead of creating a concatenated token sequence for an NL-PL pair like single BERT, it passes each artifact sequentially to the BERT model and creates separate hidden state matrices for each of the two artifact types (i.e., NL and PL). The generated two hidden state matrices are then pooled and concatenated to produce a joint feature vector as in the Twin BERT architecture. This joint feature vector is then sent to classification headers to accomplish the prediction task.

Both Siamese and Twin T-BERT architectures concatenate artifact feature vectors to create a joint feature vector. Nils et al. [29] explored the impact of different concatenation approaches for the siamese BERT architecture and showed that given two pooled feature vectors $u$ and $v$, siamese BERT with a joint feature vector $(u, v, |u - v|)$ achieved the best performance on a sentence classification task. We therefore apply this type of concatenation method to fuse the NL and PL feature vectors to create a joint feature vector.

• **SINGLE:** The single BERT architecture is shown in Fig. 3c. NL and PL text are annotated with special tokens and then concatenated into a single sequence. For example, token [CLS]/[SEP] is used to annotate the start/end of a sentence. A sentence S with tokens $s_1, s_2, s_3, ..s_N$, and a piece of code C with tokens $c_1, c_2, c_3, ..c_N$ will be transformed into an input format of $[CLS]s_1, s_2, s_3, ..s_N, [SEP]c_1, c_2, c_3, ..c_N[SEP]$. The annotated and concatenated sequence is fed to the single BERT to generate a single hidden state matrix. A subsequent pooling layer then reduces the dimension of the matrix to create a fused feature vector, which is a counterpart of the joint feature vector in SIAMESE and TWIN. This feature vector is used by the classification header, to predict whether the input NL-PL pair is related or not.

## IV. Model Training

In this section, we describe the training strategies used for pre-training, intermediate-training , and fine-tuning phases. The dataset supporting pre-training and intermediate-training phases is provided by Hamel et al. [30] from their study of the code search problem. It includes function definitions and their associated doc-strings scraped from numerous Github projects, and includes Go, Java, JavaScript, PHP, Python, and Ruby programming languages.

The dataset used in the fine-tuning phase was retrieved from OSS by our team. We extracted issues and commits through Github's APIs and mined ground-truth trace links from the commit messages. We show the data format in Fig. 1, and explain details of the data collection process in Sec. V-A. In this study we selected Python as our target language for both training and evaluation due to the large number of active projects; however, our approach is not language dependent. Given sufficient time, the same post-training process could be applied to other programming languages.

### *A. Three Step Training*

• **Pre-training Code Language Model:** In the pre-training step, we leveraged the BERT model to learn the word distribution among NL and PL documents, and refer to this BERT model as 'code BERT' to distinguish it from the plain BERT model that handles only NL text. In the plain BERT model, masked LM (MLM) tasks were used to pre-train BERT as a language model. As previously explained, in MLM tasks, 15% of tokens are selected and masked, and then BERT is trained to recover the masked tokens based on their surrounding context.

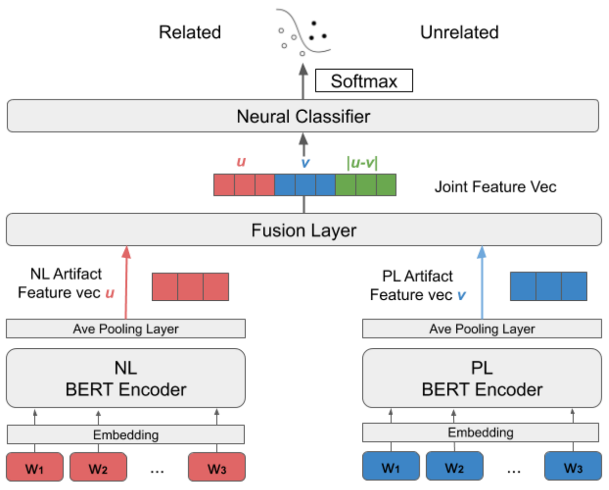
Given that pre-training a language model is very expensive, three commercial organizations have released their own pre-trained code BERT models (Hugging Face [31], CodistAI [32], and Microsoft [26]), all of which were trained on the CodeSearchNet dataset. Of these, we leverage Microsoft's model (referred to as MS-CodeBert) as our source code language model directly for T-BERT relation classification models depicted in Fig. 3, as it has been shown to deliver improved language comprehension for diverse downstream software engineering tasks. These improvements in MS-CodeBert can be attributed to its 'Replaced Token Detection' training tasks, which have been shown to be a more effective way of training LMs [33]. This training task replaces a small portion tokens in the corpus with random tokens and then requires the BERT model to identify which tokens have been replaced in the corpus.

• **Intermediate-training:** For intermediate-training we trained T-BERT models to perform the code search problem, as this problem is inherently similar to the NLA-PLA traceability challenge. In both cases, we used T-BERT to retrieve related source code based on a NL description of code functionality. The CodeSearchNet dataset [1] provides a benchmark for the code search problem, as each function in the dataset is paired with a doc-string. For Python, the dataset includes 824,342 functions for training, 46,213 functions for development and 22,176 functions for testing. This dataset is ideal for intermediate-training purposes because 1) it is large in size, therefore the T-BERT model has adequate labeled data to learn general rules for identifying NL-PL relevance, 2) the relationships between doc-string and function are definitive, meaning that there is minimal noise in the ground truth, and 3) the function definitions use only part of the python grammar which makes this task easier to handle than NLA-PLA traceability.

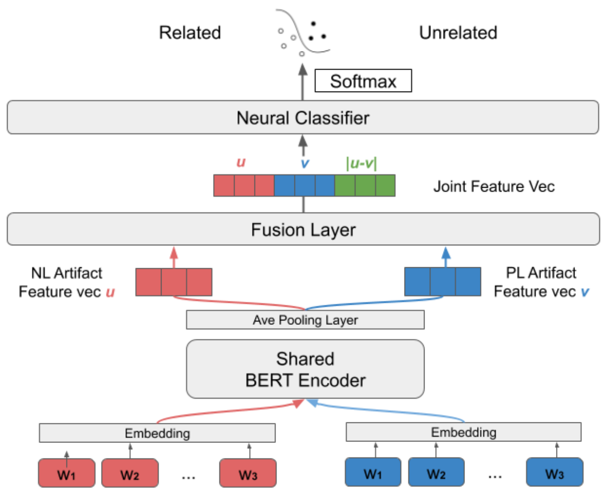
We formulated the intermediate-training as a binary classification task in which T-BERT was asked to identify whether a given doc string properly describes its paired function or not. The loss function used in this intermediate-training step was Cross Entropy Loss, and Adam Optimizer [34] was used to update the parameters and optimize for the loss function.

The code search problem creates an unbalanced distribution of positive and negative docstring-to-code pairs, we therefore created a balanced training dataset with an equal number of positive and negative samples. Guo et al. [10] adopted a dy-

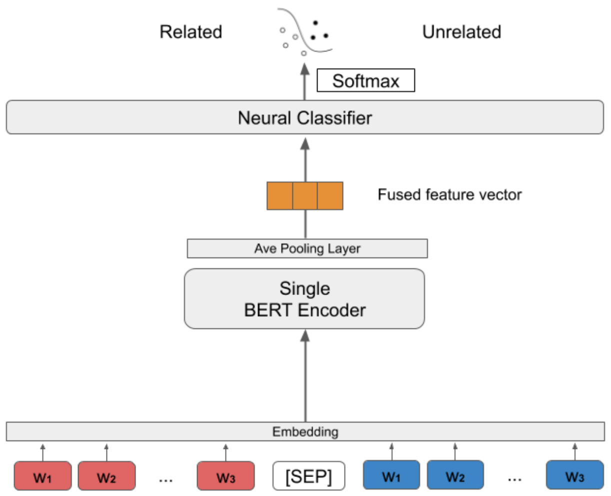
[1]CodeSearchNet dataset https://github.com/github/CodeSearchNet



(a) TWIN



(b) SIAMESE



(c) SINGLE

Fig. 3: The architectures of the three T-BERT models proposed and evaluated in our experiments.

namic under-sampling strategy to avoid inflating training data size while continually exposing the model to previously unseen negative samples. We adopted a similar technique to construct our training samples. In each epoch, a balanced training set was constructed by including all function and doc-string pairs from CodeSearchNet dataset, as well as a randomly selected equal number of non-related pairs. We updated the training set at the beginning of each epoch, so that the T-BERT model could learn from previously unseen negative examples. We refer to this training strategy as Dynamic Random Negative Sampling (DRNS), and compare it to other training strategies described in Sec. IV-B.

- **Fine-tuning:** In fine-tuning, we utilized a similar training technique to that discussed in the previous step, but addressed the traceability challenge of tracing issues to code commits using real world OSS datasets. Although the input data is formatted differently to the intermediate-training format, T-BERT uses the same architecture for both tasks. As shown in Fig. 1, the issues are comprised of a short issue summary and a long issue description while the commit is composed of a commit message and code change set. For each type of artifact, we concatenated the text to formulate input sequences for the T-BERT model (i.e., issue summary + issue description and then commit message + code change set.) In contrast to the intermediate-training step, the dataset utilized in this step is limited and fuzzy. As reported by Rath et al., link sets mined from OSS projects are unlikely to be complete and entirely accurate as engineers may forget to tag issues, or may commit multiple changes against multiple tags. [16]. Furthermore the number of links in the project-specific fine-tuning phase is significantly smaller than the number of function to doc-string pairs used in post-training. As reported in Table. I, we have approximately six hundred true links for fine-tuning compared to 824,342 pairs of function and doc-string. Furthermore, the code change set in commits has a more flexible and complex format than the short and succinct function definitions.

## B. *Negative Sampling*

Guo et al., observed a glass ceiling in terms of achieved trace link performance in which the accuracy of their neural trace model increased as the training epoch increased at the beginning, however, it then reached a peak value and started to decline with further training [10]. Our hypothesis of this phenomenon is as follows. At earlier stages of training, the trace model can effectively learn rules for distinguishing positive and negative examples, and the neural trace model was easily able to rule out many unrelated examples (i.e., cases where the source and target artifacts have little common vocabulary). Since those types of negative examples constitute the majority of negative examples, the random negative sampling strategy experiences few challenging examples and therefore starts overfitting based on näive rules, because these näive rules are applicable for the majority of simple cases. This overfitting causes accuracy to decline after a few training epochs.

Our approach adds high quality negative samples to alleviate this problem through our proposed Online Negative Sampling

(ONS) as an alternative to Dynamic Random Negative Sampling (DRNS). The principle idea of ONS is that, instead of creating a training dataset at the beginning of each epoch, the trace model will generate negative examples dynamically at the batch level. For illustrative purposes imagine we want to create a batch of size 8 containing 4 positive links. If we have 4 NL artifacts and 4 PL artifacts (i.e., a total of 16 candidate links), we include the 4 positive links, and then select 4 negative links from the 12 candidates in order to create a balanced batch. By evaluating these negative links and ranking them by predicted score, we can identify the false links which are more likely to be mistakenly classified. Then by incorporating the top-scoring negative examples into the training set, we improve the quality of negative examples and avoid early overfitting. This approach is inspired by applications in the Face Recognition domain, where face recognition models need to distinguish between people with similar appearances [35]. This negative example mining strategy is usually combined with Triplet Loss [36] in the contrastive learning [37] framework. Here we adopt it for our classification and combine it with the widely used Cross Entropy Loss.

## V. Experimental Evaluation

### A. Data Collection

In this study, we train T-BERT and evaluate it against two types of datasets. The first dataset is CodeSearchNet which is publicly available [30]. It includes functions and their associated doc-strings for six different programming languages. As previously stated, we focused on python functions.

The other datasets[2] we leveraged were mined from three OSS projects in Github and included Pgcli, Flask and Keras as described in Table. I. We selected these projects because they are popular, actively maintained Python projects, with developers actively tagging commits with issue IDs. We retrieved issues and their discussions as the source artifacts and commits as target artifacts. For each issue, we included both the short issue summary and the longer issue description. We automatically removed stack traces from issue discussions if highlighted as Code block in markdown, as we wanted to train our approach to perform the harder job of generating links between issues and code without the more explicit information provided by stack traces. For each commit, we included the commit message and change set. However, we removed very small commits (with less than 5 LOC) from our target artifact set. Finally, due to the Github API rate limit, we retrieved a maximum of 5000 issues for each project.

After retrieving commits and issues we mined a 'golden link set' from the commit messages by using issue tags embedded into commit messages added by committers. In addition, we leveraged pull requests, as an accepted pull request automatically creates both an issue and commit in the OSS project, and connects them through an issue ID embedded into the commit message. Tags were mined using regular expressions in order to build a link set connecting issues and commits. One risk of mining links from commit message is that the link set may be incomplete. Liu et al. partially addressed this problem by pruning the dataset and only retaining artifacts appearing in links set [38]. We adopted this process to construct our dataset and report results in Table I

[2]OSS dataset https://zenodo.org/record/4511291#.YB3tjyj0mbg

TABLE I: The size of software project leveraged in traceability experiment. We applied the cleaning procedures, described in Sec. V-A, to clean artifacts and links.

| | | Commits | Issues | Links | Type |
|---|---|---|---|---|---|
| Pgcli | original | 2191 | 1197 | 645 | database |
| | cleaned | 531 | 522 | 530 | command line |
| Flask | original | 4011 | 3711 | 1159 | web framework |
| | cleaned | 752 | 739 | 753 | |
| Keras | original | 5348 | 4810 | 9375 | neural-network |
| | cleaned | 551 | 550 | 551 | library |

### B. Experiment Setup

We conducted our experiment on Supermicro SYS-7048GR-TR SuperServer with Dual Twelve-core 2.2GHz Intel Xeon processors and 128GB RAM. We utilized 1 NVIDIA GeForce GTX 1080 Ti GPU with 10 GB memory to train and evaluate our model. The T-BERT model was implemented with PyTorch V.1.6.0 and HuggingFace Transformer library V.2.8.0. We trained models for 8 and 400 epoch in intermediate-training and fine-tuning. For each task, we use a batch size of 8 and a batch accumulation step of 8. We set the initial learning rate as 1E-05 and applied a linear scheduler to control the learning rate at run time. Regarding the model selection, we split the dataset into training (train), development (dev), and test sets. We trained the model using the training dataset and tested its performance on the dev dataset. We then selected the best performing model based on the dev dataset and created an output model. We finally evaluated and compared the performance of output models on the test dataset. In the intermediate-training stage the dataset was already split by the data provider. In the fine-tuning stage, we split the dataset into ten folds, of which eight were used for training, one for development, and one for testing.

### C. Evaluation Metrics

The metrics for our experiments include F-scores, Mean Average Precision (MAP@3), Mean Reciprocal Rank (MRR), and Precision@K.

- **F-scores:** F-scores are composite metrics calculated from precision and recall, and are frequently used to evaluate traceability results. The F-1 score assigns equal weights to precision and recall, while the F-2 score favors recall over precision. Although both precision and recall are important, F-2 is usually preferred for evaluating trace results where recall is considered more important than precision. We report the best F-scores in our experiments by enumerating the thresholds.

$$F_\beta = \frac{(1+\beta^2)\cdot precision \cdot recall}{\beta^2 \cdot precision \cdot recall} \tag{1}$$

• **Mean Average Precision:** MAP evaluates the ranking of relevant artifacts over retrieved ones. Each source artifact is regarded as a query Q for retrieving artifacts. After ranking the retrieved target artifacts, an Average Precision (AveP or AP) score is obtained based on the position of all relevant target artifacts in the ranking. The Mean of AveP scores is then computed to return the MAP. In our study, we apply a stricter metric known as MAP@3, in which only artifacts ranked in the top 3 positions contribute to $AveP$ score. The formula for this metric is shown in following, k represents the total relevant target artifacts in a query and $rank_i$ refer to the ranking a target artifact :

$$AveP@3 = \frac{\sum_i^k X}{k},\ X = \begin{cases} P@i, & \text{if } rank_i <= 3 \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

$$MAP@3 = \frac{\sum_j^Q AveP_j@3}{Q} \quad (3)$$

• **Mean Reciprocal Rank:** MRR is another measurement of the result ranking. In each query, the first related target artifact with a rank of N, will provide a Reciprocal Rank of 1/N. MRR accumulates by averaging the Reciprocal Rank for all the queries Q. This focuses on the first effective results for a query, while ignoring the overall ranking. This is the standard metric used for the CodeSearchNet benchmark. While MAP is more typical for trace retrieval tasks, we include this metric to compare our intermediate-trained model against the approaches in other studies for code search problem.

$$MRR = \frac{1}{Q}\sum_{|i=1|}^{Q}\frac{1}{1stRank_i} \quad (4)$$

• **Precision@K:** Precision@K evaluates how many related artifacts are retrieved and ranked in the top K. The formula for this metric is shown in Eq. 5. We provide results with K values of 1 to 3. A trace model with high Precision@K means users are more likely to find at least one related target artifact in the top K results.

$$Precision@K = \frac{\sum_i^Q |Rel_i@K|}{|Rel|} \quad (5)$$

As we can see, MRR and Precision@K ignore recall and focus on evaluating whether the search result can find interesting results for a user. They are ideal for the code search problem but not for traceability where recall is particularly important. Therefore, we apply only F-score and MAP@3 to evaluate traceability results.As the majority of our queries have fewer than three correct links, a perfect MAP@3 score represents close to 100% recall.

## VI. Results And Discussion

We report performance results of T-BERT models for the code search problem and the NLAs-PLAs traceability problem, and address the RQs defined in Sec. I.

### A. Evaluating the Code Search Problem

Our first evaluation explores how well T-BERT models perform when datasets have adequate labeled examples. We trained T-BERT models for the three architectures introduced in Sec. III-C, using the training part of the CodeSearchNet dataset. For the T-BERT models which are trained with ONS technique, we add a star sign after the model names to distinguish them from the T- BERT model trained with DRNS. For example, SINGLE* refers to the model with single BERT architecture and trained with online negative sampling. The performance of these six types of models are reported in Table. II. In addition, we compare T-BERT models against three classical tracing techniques of VSM, LDA, LSI and also a deep learning trace model, TraceNN [10] for this dataset.

Other researchers such as Hamel et al. [30] and Feng [26] leveraged the same dataset to conduct code search study, and so we select the methods which achieved the best MRR scores in their study as a comparison to T-BERT, and created our evaluation dataset for CodeSearchNet in the same way. For each doc-string, we combined the related function with 999 unrelated ones, and charged the retrieval models with finding the correct function among one thousand candidates. However, the SINGLE models were not able to efficiently process the entire dataset, and so in this case we evaluated only 100 out of the total 22,176 queries. The comparison of MRR scores are shown in Table. II.  To observe the learning process for each model, we visualized the learning curve in Fig .4 for the first

TABLE II: Evaluation of T-BERT models on the CodeSearch-Net Challenge dataset

| | F1 | F2 | MAP | MRR | Pr@1 | Pr@2 | Pr@3 |
|---|---|---|---|---|---|---|---|
| TWIN | 0.497 | 0.563 | 0.735 | 0.756 | 0.646 | 0.787 | 0.842 |
| SIAMESE | 0.604 | 0.668 | 0.814 | 0.825 | 0.729 | 0.866 | 0.915 |
| SINGLE | 0.482 | 0.572 | 0.825 | 0.839 | 0.730 | 0.900 | 0.930 |
| TWIN* | 0.559 | 0.626 | 0.794 | 0.809 | 0.712 | 0.846 | 0.890 |
| SIAMESE* | 0.594 | 0.655 | 0.817 | 0.829 | 0.738 | 0.866 | 0.910 |
| SINGLE* | 0.612 | 0.678 | 0.837 | 0.851 | 0.750 | 0.910 | 0.930 |
| VSM | 0.219 | 0.255 | 0.314 | 0.351 | 0.251 | 0.341 | 0.397 |
| LDA | 0.005 | 0.010 | 0.012 | 0.021 | 0.008 | 0.013 | 0..017 |
| LSI | 0.003 | 0.007 | 0.014 | 0.025 | 0.009 | 0.015 | 0.020 |
| TNN-LSTM | 0.179 | 0.245 | 0.351 | 0.400 | 0.269 | 0.386 | 0.457 |
| TNN-BiGRU | 0.221 | 0.29 | 0.392 | 0.438 | 0.304 | 0.432 | 0.504 |
| JV-biRNN | | | | *0.321 | | | |
| JV-SelfAtt | | | | *0.692 | | | |
| MSC | | | | *0.860 | | | |

JV=Joint Vector [30]; MSC=MS-CodeBERT [26]; TNN = TraceNN [10]; *Previously reported results against same CodeSearch-Net challenge dataset.

TABLE III: Training and testing time for T-BERT models on code search problem. The test time is recorded for a test set with 100 queries.

| | Strategy | TWIN | SIAMESE | SINGLE |
|---|---|---|---|---|
| Train(hr) | DRNS | 156h | 138h | 164h |
| Test(sec) | DRNS | 3254s | 3264s | 183357s |
| Train(hr) | ONS | 146h | 142h | 283h |
| Test(sec) | ONS | 3211s | 3265s | 193667s |

35,000 steps of optimizations. We evaluate the performance of each model at intervals of 1000 steps during the training by applying the intermediate model against small testing sets composed of 200 development examples.

## *B. Evaluate NLA-PLA Traceability*

We then evaluated the T-BERT models on the NLA-PLA Traceability problem. As previously described, we used 8 folds of trace links for training, 1 fold for developing and 1 fold for testing. To explore our RQ3, we conducted a controlled experiment, in which we trained two groups of T-BERT models. In the first group, we continued training the T-BERT model which had been intermediate-trained in our previous experiment. In the second group, we trained the T-BERT models without applying transferred knowledge learned from intermediate-training . When we conducted model training, we applied the same training dataset and ONS techniques to the two groups. To maintain consistency of abbreviations, we name the models in the first group, for example, as SINGLE*+T; and the models in the second group as SINGLE*. The result of this experiment are shown in Table. IV, while the learning curve for the T-BERT models showing the fine-tuning to traceability tasks is shown in Fig. 5. We show only the learning curve for Pgcli data due to space constraints.

## *C. RQ2: How does ONS alleviate the glass ceiling problem?*

In this section, we discuss how the effectiveness of ONS for T-BERT models alleviates the glass ceiling problem. This question helps us to identify the best approach for training T-BERT models. To answer this question we apply both ONS and DRNS to the same T-BERT architecture to create test and control groups. Fig. 4 shows the learning curve of T-BERT models on CodeSearchNet dataset during training. The orange lines represent the models trained with ONS while blue lines represent those trained with DRNS. We find that for TWIN and SINGLE model, the orange line is always above the blue line, meaning that ONS can accelerate the learning for T-BERT models but also let it converge at a higher value. While for SIAMESE we find the orange line is above blue line in the early steps, but soon converges at a similar level. This result indicates ONS benefits the T-BERT model training by introducing harder negative examples. The evaluation results shown in the first six rows of Table. II also support this finding, as the TWIN* and SINGLE* models (line 4,6) achieve better results than T-BERT models (line 1,3) from the perspectives of all metrics. SIAMESE* (line 5) and SIAMESE (line 2) have a very close result where the difference of all metrics are within 0.5% except F2 score (1.3%).

We report the training time for DRNS and ONS in Table III. ONS introduces initial overheads in constructing each batch, but then has fewer candidates to evaluate and sort. In contrast, DRNS has no upfront construction costs, but must sample data from a large list, creating a performance bottleneck. The use of ONS only significantly increased training time except for the SINGLE model which is particularly slow on evaluation.

We conclude that ONS delivers better accuracy than DRNS, and only increases training times for models (e.g., SINGLE) which have slow evaluation processes.

## *D. RQ1: Which T-BERT architecture is better?*

This RQ focuses on comparing the accuracy and efficiency of T-BERT when used for trace retrieval tasks. Performance comparisons were conducted on T-BERT* models, as we have shown in RQ2 that T-BERT* models returned better accuracy than T-BERT ones. To answer this question, we further divided our questions into three sub RQs as following.

• **RQ1.1:** Are T-BERT models capable of resolving the CodeSearchNet and NLA-PLA traceability problem?

Table. II shows that SINGLE*, TWIN*, and SIAMESE* (line 4-6) can achieve F scores around 0.6 and MAP scores around 0.8. The Precision@3 score for the three models are around 0.9 which means T-BERT* models can return related functions in around 9 of 10 user queries. And in 75% to 80% of cases the correct answer definition is ranked at the first position. This result shows that all three models are effective for the CodeSearchNet challenge. Among these three models, SINGLE* achieves the best performance with respect to all metrics. However, the gap between these three models is small, and all three models clearly outperform the base line created by the three IR models of VSM, LSI, and LDA.

In Table. IV, the first three rows show that T-BERT* models applied to the traceability challenge and trained without the benefit of transfer knowledge are ranked in the same way as for the code search problem. However, the performance gap between these three models increases. This suggests that the size of training data has different impacts on the three types of architecture. Since, TWIN* model includes two inner BERT models, the parameters in this architecture are doubled, and it therefore requires more training examples to tune the models. Nevertheless, all T-BERT* models achieved achieve better results than the IR model baselines. Especially in Keras dataset, SIAMESE* and SINGLE* (line2 and 3) have an F score above 0.95 and MAP of 0.99 indicating that T-BERT can provide perfect tracing results in some scenarios.

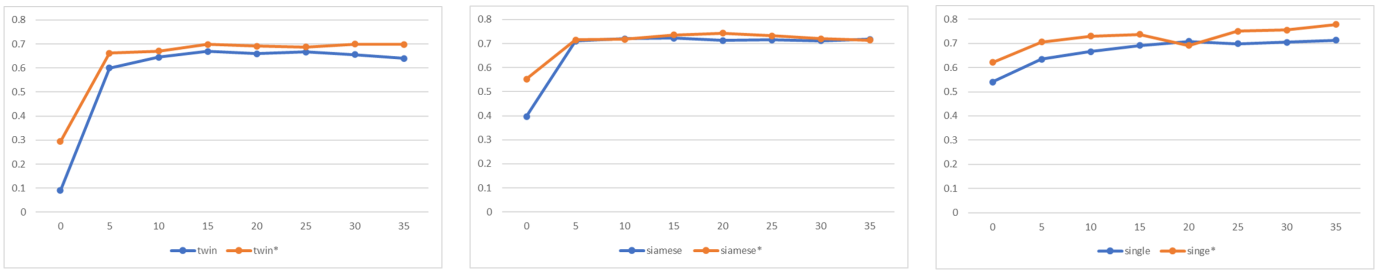
• **RQ1.2:** Which T-BERT model most effectively addresses the two problems of data sparsity and performance ?

We need to take both accuracy and efficiency into consideration when selecting a model for use in production. As discussed in RQ1.1, the SINGLE* model achieves best performance; however, it is very slow for processing large scale datasets. As shown in Table. III, TWIN and SIAMESE need around 3000 seconds to evaluate 100 queries while SINGLE needs around 20000 seconds. We estimate that it will take around 6000 hours for SINGLE to evaluate the whole CodeSearchNet test with our current experiment setup. But for TWIN and SIAMESE, it took us only around 20 hours to evaluate the whole test set in practise. In the traceability challenge, the test set is relatively tiny. Taking Pgcli for example, it contains 2704 candidate links compose of 52 source artifact and 52 target artifacts. TWIN and SIAMESE both take around 160 seconds to finish the task while the

TABLE IV: Evaluation of models on NLAs-PLAs traceability

| | | Pgcli | | | Flask | | | Keras | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | F1 | F2 | MAP | F1 | F2 | MAP | F1 | F2 | MAP |
| TWIN* | | 0.450 | 0.491 | 0.574 | 0.524 | 0.577 | 0.683 | 0.450 | 0.491 | 0.574 |
| SIAMESE* | | 0.621 | 0.654 | 0.728 | 0.681 | 0.731 | 0.801 | 0.962 | 0.962 | 0.990 |
| SINGLE* | | 0.707 | 0.745 | 0.785 | 0.841 | 0.873 | 0.952 | 0.931 | 0.925 | 0.971 |
| TWIN* | T | 0.686 | 0.709 | 0.766 | 0.750 | 0.781 | 0.869 | 0.953 | 0.970 | 0.978 |
| SIAMESE* | T | 0.729 | 0.748 | 0.779 | 0.820 | 0.830 | 0.920 | 0.971 | 0.977 | 0.990 |
| SINGLE* | T | 0.730 | 0.789 | 0.859 | 0.884 | 0.862 | 0.92 | 0.972 | 0.989 | 0.990 |
| VSM | | 0.376 | 0.424 | 0.506 | 0.509 | 0.474 | 0.540 | 0.532 | 0.512 | 0.703 |
| LDA | | 0.121 | 0.226 | 0.208 | 0.182 | 0.241 | 0.227 | 0.290 | 0.367 | 0.333 |
| LSI | | 0.085 | 0.145 | 0.147 | 0.127 | 0.164 | 0.142 | 0.072 | 0.126 | 0.109 |
| TNN-LSTM | | 0.138 | 0.179 | 0.128 | 0.106 | 0.126 | 0.080 | 0.053 | 0.087 | 0.034 |
| TNN-BiGRU | | 0.062 | 0.116 | 0.006 | 0.066 | 0.100 | 0.044 | 0.063 | 0.119 | 0.073 |

T=Transfer learning, TNN = TraceNN [10];



(a) TWIN (b) SIAMESE (c) SINGLE

Fig. 4: The learning curve for T-BERT and T-BERT* models on code search challenge. This figure shows the MAP scores (Y-Axis) over the first 35K (X-Axis) Adam optimization steps.

SINGLE model takes around one hour. SIAMESE and TWIN architectures accelerate the process by decoupling the feature vector creation steps. In the SINGLE architecture, NL and PL document pairs are fed to BERT to create a joint feature vectors. This step is extremely expensive and creates the main performance bottleneck for the SINGLE model. Assuming we have N source and N target artifacts. SINGLE has a time complexity of $O(N^2 * K)$ for creating feature vectors for all the candidate links, where K refer to the time consumed by the BERT model to convert an input token sequence into a feature vector. TWIN and SIAMESE only need $O(N * K)$ to convert artifacts into feature vectors and then $O(N^2)$ time to concatenate the feature vectors together. The time complexity of TWIN and SIAMESE is one order of magnitude lower than the SINGLE model thus more scalable to projects with massive artifacts. We argue that SIAMESE is the most appropriate model for addressing NLA-PLA traceability taking both accuracy and efficiency into consideration, because it can achieve an accuracy close to the SINGLE architecture for the traceability challenge while maintaining the low time complexity of the TWIN architecture. However, in cases where accuracy is the primary concern, e.g. traceability for safety critical projects, users should adopt SINGLE model supported by high-performance hardware.

TABLE V: Model performance on Pgcli dataset for NLA-PLA.

| | TWIN* | SIAMESE* | SINGLE* |
|---|---|---|---|
| Train(hr) | 12h | 12h | 13h |
| Test(sec) | 170s | 163s | 5395s |

• **RQ1.3:** How do T-BERT models compare to other approaches ?

For the CodeSearchNet challenge, we compared the performance of T-BERT models to Joint Vector Embedding (JVE) and MS-CodeBERT. JVE's architecture is similar to TWIN, and leverages two encoders to create feature vectors for a classification network. Previous studies have reported 60% as the highest MRR achieved by JVE on the same dataset, which is lower than T-BERT models. MS-CodeBERT, provided by Microsoft, used the same architecture as SINGLE in our experiment. However, MS-CodeBERT was trained with a batch size of 256 on a cluster with 16 Tesla GPU, and no special techniques were applied during training. Our machine only allows one small batch due to memory limitations, but SINGLE* model's MRR results were only 0.9% lower than MS-CodeBERT, indicating that our training techniques partially alleviate limitations introduced by less powerful hardware.

TNN [10] is an RNN based trace model proposed by Guo et al. and designed for generating NLA-NLA links. We reconstructed the model according to the authors' specifications and applied it to our NLA-PLA problem for comparison purposes. TNN utilizes Word2Vec embedding to transform tokens into vectors. It uses two alternate RNN networks, LSTM or Bidirectional GRU (BiGRU) to generate semantic representations of NLA and PLA, and feeds these semantic hidden states to the integration layer to generate a new hidden state representing

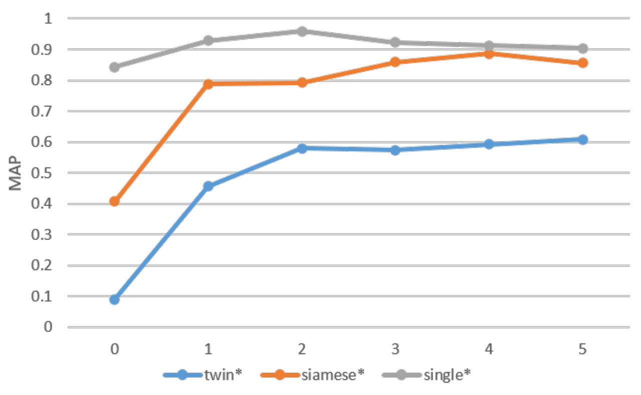


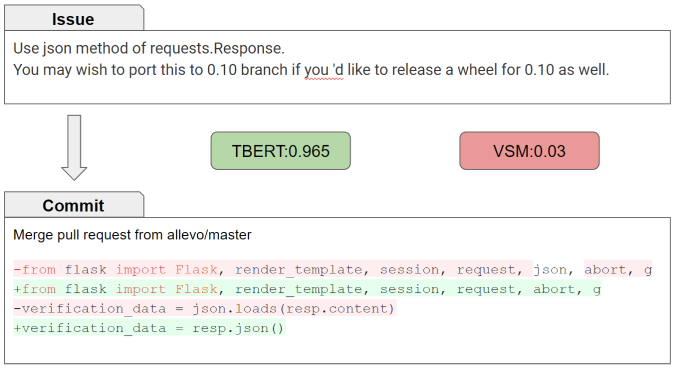
Fig. 5: Learning curve of T-BERT* models on Pgcli dataset for NLA-PLA trace challenge. This figure shows the MAP scores (Y-Axix) over 5k Adam optimization steps (X-Axis)

the correlations between the NLA and PLA from which links are generated. Our embedding layer was constructed by unsupervised training of a skip-gram Word2Vec model using artifacts from the three OSS reported in Table I. We evaluated both LSTM and BiGRU for the RNN layer in this study. TNN results are shown at the bottom of Table. IV, and show that it underperformed all BERT models and VSM on all three OSS projects. We provide an illustrated example in Fig. 6, showing T-BERT and VSM results for a commit-issue pair, tagged by the committer as related.

Guo et al., reported improvements over the VSM model for their NLA-NLA dataset, we were unable to replicate these for our NLA-PLA problem. An inspection of the TNN learning curve indicated that TNN effectively reduced the loss and improved the link prediction accuracy for all three training datasets in training dataset, but converged early in the validation datasets and then decreased in accuracy - indicating an overfitting problem. There are several possible explanations for these results. First, the dataset used by Guo et al., contained 1,387 positive links versus our 530-739 links, which could be insufficient for RNN training. Second, programming languages have an open vocabulary in which new terms can be created as variable and function names, and TNN may therefore need a larger training set to generate NLA-PLA links versus NLA-NLA ones. Our hypotheses are supported by the observation that TNN does not overfit when applied to the CodeSearchNet where larger numbers of training examples are provided. T-BERT models leverage transferred knowledge from pretrained language models and adjacent problems to reduce the requirements of the training dataset size, and are therefore able to handle tracing challenges which can not easily be addressed by classical Deep Learning trace models. This characteristic makes T-BERT more practical for industrial applications.

### *E. RQ3: To what extent can T-BERT leverage transfer knowledge from code search to software traceability*

Table IV the T-BERT model trained with and without transferred knowledge from the post-pretrained model. The results show that intermediate-training T-BERT models on the code search task can significantly improve their performance on the traceability problem. Taking SIAMESE on Pgcli for example, the F2 score increased from 0.654 to 0.748, while the MAP score increased from 0.728 to 0.779. Similar results are observed for the other datasets with different T-BERT models. This suggests that the knowledge learned from text to structure code (function definition) can be effectively transferred to cases where 1) code formats are more fuzzy and 2) training data has limited labels. Intermediate-training improvements



Use json method of requests.Response.
You may wish to port this to 0.10 branch if you 'd like to release a wheel for 0.10 as well.

Merge pull request from allevo/master

```
-from flask import Flask, render_template, session, request, json, abort, g
+from flask import Flask, render_template, session, request, abort, g
-verification_data = json.loads(resp.content)
+verification_data = resp.json()
```

Fig. 6: In this example, a link is tagged by developers, retrieved by the T-BERT model (with a high score o 0.965 due to semantic similarity and context) and missed by VSM, because the key terms ’request‘ and ’json‘ are common terms.

were observed to different extents across the three architecture types. As shown in Fig. 5, the blue line (SINGLE) converged at a very early stage, showing that SINGLE needed only relatively few epochs on the smaller task-specific dataset to localize its transferred knowledge. SIAMESE converged slower, while TWIN converged slowest of all, indicating that each architecture has a different capacity for transferring knowledge.

## VII. Related Work

Our study constructs T-BERT models using three different architectures, all of which have previously been used to address related problems in other domains. Lu et al. leveraged a TWIN-like model, named TwinBERT, as a search engine to deliver ads alongside organic search results [39]. They used reinforcement training techniques and found that TwinBERT could return results with high accuracy and low latency. Reimers et al. proposed a SIAMESE architecture to address problems such as Semantic Textual Similarity [29]. They trained their model to determine whether two sentences were related through contradiction or entailment, and utilized SNLI [40] and the Multi-Genre NLI [41] dataset for training and evaluation. Their results showed that SIAMESE BERT could achieve a high Spearman Rank Correlation score around 0.76. They also found that use of averaging pooling was more effective than max pooling and first token (the [CLS] token) pooling; and that concatenating source and target hidden states as $(u, v, \|u - v\|)$ achieved best results. We adopted these findings when we built T-BERT models for this study. However, no study has yet been conducted comparing the TWIN, SIAMESE and SINGLE architectures.

To address the NLA-PLA challenge, we adopted the code search problem as our intermediate solution. Several studies have addressed the code search problem using a recurrent neural network (RNN). We have already made comparisons to the work by Feng et al. [26] and Huian et al. [30] in Table. II; however, in another study, Gu et al. [42] converted method specifications into API call sequences and then processed the sequence with RNN. They reported achieving 0.6 MRR in a test set with 100 queries. However, we can not directly adapt this method to the traceability challenge, because, unlike API calls, the statements in code change sets are not structured. A related domain for addressing NLA-PLA is source code embedding. By converting both source code and documents into distributed representations, the relevance between these two type of artifacts can be effectively calculated through distance metrics such as Cosine and Euclidean Distance. Code2Vec [43] belong to this type of approach. T-BERT models can adapt to this type of training by integrating Cosine Embedding Loss in the classification header. We leave this exploration for future work.

## VIII. Threats to Validity

There are several threats to validity in this study. First, Our current experiments have only been conducted on Python projects, and results could differ when applied to other programming languages. Also, due to time constraints we fine-tuned and evaluated the T-BERT model performance on only three OSS projects, which may not be enough to draw generalized conclusions. Second, we construct our experiment datasets from OSS projects by mining the issues and commits whose IDs are explicitly marked as related by project maintainers. Although, this is a conventional way of leveraging OSS projects for traceability, true links may be missed. For example, a bug report may have hidden dependencies on several other issues such as a feature request or other bug report even though a commit addressing the parent bug report is not marked as 'related'. We alleviate the impact of this phenomena by adopting the data processing suggested by Liu et.al. [38]. Another important threat is that while the SINGLE architecture, trained for code search problem, does not outperform CodeBERT, further improvements could be achieved using hyper parameter optimization. Our experiments were limited by hardware availability for conducting excessive hyper parameter tuning. However, the performance comparison across T-BERT models should still be valid because all experiments were conducted with the same parameters. Finally, due to processing time constraints, we evaluated the SINGLE model on 100 queries whilst using the entire testing set for the other models (in Table. II). Although not reported we also evaluated TWIN and SIAMESE on 100 queries and observed that they achieved almost identical results to those obtained from the whole test set indicating that 100 queries was a reasonable sample size for the SINGLE model.

## IX. Conclusion and Future Work

This study has explored several different BERT architectures for generating trace links between natural language artifacts and programming language artifacts. Our experimental results showed that the SINGLE architecture achieved the best accuracy but at long execution times, whilst the SIAMESE architecture achieved similar accuracy with faster execution times. Second, we showed that ONS training (based on negative sampling) improved both performance and model convergence speed without incurring significant performance overheads when compared to DRNS. Third, we found that T-BERT was able to effectively transfer knowledge learned from the code search problem to NLA-PLA traceability, meaning that intermediate-trained T-BERT models can be effectively applied to software engineering projects with limited training examples, alleviating the data sparsity problem for deep neural trace models. Regarding the training time, we showed that the same intermediate-trained T-BERT can be applied for OSS projects in three different domains. By avoiding the need for intermediate training on each individual project, our approach was able to efficiently adapt to new domains. In conclusion, our results show that T-BERT generates trace links at far higher degrees of accuracy than existing information retrieval and RNN techniques – bringing us closer to achieving the vision of practical and trustworthy traceability.

To support replication and reproducibility, we have provided links throughout this paper to the datasets that we used and we provide a complete implementation of T-BERT and execution instructions on github[3].

In future work we will evaluate our approach across more diverse project domains and programming languages, and will explore its application to more diverse types of software artifacts such as requirements, design, and test cases.

## Acknowledgment

This work has been partially funded under US National Science Foundation Grant SHF:1901059.

## References

[1] L. Rierson, *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[2] A. Mahmoud, N. Niu, and S. Xu, "A semantic relatedness approach for traceability link recovery," in *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE, 2012, pp. 183–192.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[4] J. Huffman Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 4–19, 2006.

[5] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Enhancing an artefact management system with traceability recovery features," in *20th IEEE International Conference on Software Maintenance (ICSM)*, 2004, pp. 306–315.

[3]T-BERT source code https://github.com/jinfenglin/TraceBERT

[6] P. Rempel, P. Mäder, and T. Kuschke, "Towards feature-aware retrieval of refinement traces," in *7th International Workshop on Traceability in Emerging Forms of Software Engineering, TEFSE 2013, 19 May, 2013, San Francisco, CA, USA*, N. Niu and P. Mäder, Eds. IEEE Computer Society, 2013, pp. 100–104. [Online]. Available: https://doi.org/10.1109/TEFSE.2013.6620163

[7] A. Dekhtyar, J. H. Hayes, S. K. Sundaram, E. A. Holbrook, and O. Dekhtyar, "Technique integration for requirements assessment," in *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*. IEEE Computer Society, 2007, pp. 141–150. [Online]. Available: https://doi.org/10.1109/RE.2007.17

[8] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, Eds. ACM, 2010, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806817

[9] H. Sultanov, J. Huffman Hayes, and W.-K. Kong, "Application of swarm techniques to requirements tracing," *Requirements Engineering*, vol. 16, no. 3, pp. 209–226, 2011.

[10] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 3–14.

[11] G. Spanoudakis, A. Zisman, E. Pérez-Miñana, and P. Krause, "Rule-based generation of requirements traceability relations," *Journal of Systems and Software*, vol. 72, no. 2, pp. 105–127, 2004.

[12] J. Guo, J. Cleland-Huang, and B. Berenbach, "Foundations for an expert system in domain-specific traceability," in *21st IEEE International Requirements Engineering Conference (RE)*, 2013, pp. 42–51.

[13] J. Cleland-Huang, P. Mäder, M. Mirakhorli, and S. Amornborvornwong, "Breaking the big-bang practice of traceability: Pushing timely trace recommendations to project stakeholders," in *2012 20th IEEE International Requirements Engineering Conference (RE), Chicago, IL, USA, September 24-28, 2012*, M. P. E. Heimdahl and P. Sawyer, Eds. IEEE Computer Society, 2012, pp. 231–240. [Online]. Available: https://doi.org/10.1109/RE.2012.6345809

[14] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 378–388.

[15] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "On integrating orthogonal information retrieval methods to improve traceability link recovery," in *27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 133–142.

[16] M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 834–845.

[17] Y. Liu, J. Lin, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Towards semantically guided traceability," in *International Conference on Requirements Engineering*, vol. 2020, 2020.

[18] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 378–388.

[19] M. Borg, C. Englund, and B. Duran, "Traceability and deep learning-safety-critical systems with traces ending in deep neural networks," *In Proc. of the Grand Challenges of Traceability: The Next Ten Years*, pp. 48–49, 2017.

[20] Y. Zhao, T. S. Zaman, T. Yu, and J. H. Hayes, "Using deep learning to improve the accuracy of requirements to code traceability," *Grand Challenges of Traceability: The Next Ten Years*, p. 22, 2017.

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

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

[23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[24] 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.

[25] J. Liu, Y. Lin, Z. Liu, and M. Sun, "Xqa: A cross-lingual open-domain question answering dataset," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 2358–2368.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[27] J. Cleland-Huang, O. C. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: trends and future directions," in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 55–69.

[28] Wikipedia, "Language model — Wikipedia, the free encyclopedia," 2020, [Online; accessed 22-July-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Language_model_Bidirectional

[29] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[30] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," *arXiv:1909.09436 [cs, stat]*, Sep. 2019, arXiv: 1909.09436. [Online]. Available: http://arxiv.org/abs/1909.09436

[31] [Online]. Available: https://huggingface.co/huggingface/CodeBERTa-small-v1

[32] [Online]. Available: https://huggingface.co/codistai/codeBERT-small-v2

[33] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.

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

[35] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[36] Wikipedia contributors, "Triplet loss," 202-, [Online; accessed 22-Aug-2020]. [Online]. Available: https://en.wikipedia.org/wiki/Triplet_loss

[37] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," *arXiv preprint arXiv:2002.05709*, 2020.

[38] Y. Liu, J. Lin, and J. Cleland-Huang, "Traceability support for multi-lingual software projects," *arXiv preprint arXiv:2006.16940*, 2020.

[39] W. Lu, J. Jiao, and R. Zhang, "Twinbert: Distilling knowledge to twin-structured bert models for efficient retrieval," *arXiv preprint arXiv:2002.06275*, 2020.

[40] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[41] A. Williams, N. Nangia, and S. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, 2018, pp. 1112–1122. [Online]. Available: http://aclweb.org/anthology/N18-1101

[42] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 933–944. [Online]. Available: https://doi.org/10.1145/3180155.3180167

[43] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.