Evaluating and Optimizing the Effectiveness of Neural Machine Translation in Supporting Code Retrieval Models: A Study on the CAT Benchmark

Hung Phan
Department of Computer Science
Iowa State University, Ames, USA
hungphd@iastate.edu

Ali Jannesari
Department of Computer Science
Iowa State University, Ames, USA
jannesar@iastate.edu

ABSTRACT
Neural Machine Translation (NMT) is widely applied in software engineering tasks. The effectiveness of NMT for code retrieval relies on the ability to learn from the sequence of tokens in the source language to the sequence of tokens in the target language. While NMT performs well in pseudocode-to-code translation [17], it might have challenges in learning to translate from natural language query to source code in newly curated real-world code documentation/implementation datasets. In this work, we analyze the performance of NMT in natural language-to-code translation in the newly curated CAT benchmark [31] that includes the optimized versions of three Java datasets TCodeSum, CodeSearchNet, Funcom, and a Python dataset PCSD. Our evaluation shows that NMT has low accuracy, measured by CrystalBLEU [10] and Meteor [9] metrics in this task. To alleviate the duty of NMT in learning complex representation of source code, we propose ASTTrans Representation, a tailored representation of an Abstract Syntax Tree (AST) using a subset of non-terminal nodes. We show that the classical approach NMT performs significantly better in learning ASTTrans Representation over code tokens with up to 36% improvement on Meteor score. Moreover, we leverage ASTTrans Representation to conduct combined code search processes from the state-of-the-art code search processes using GraphCodeBERT [13] and UniXcoder [12]. Our NMT models of learning ASTTrans Representation can boost the Mean Reciprocal Rank of these state-of-the-art code search processes using SOTA approaches GraphCodeBERT and UniXcoder thanks to its augmented code search process. We use four datasets of the CAT benchmark [31], for our evaluation. Overall, our contributions are as follows:

1 INTRODUCTION
Although Neural Machine Translation (NMT) has been proven effective in pseudocode-to-code translation [17, 37], applying NMT on practical datasets of real-world NL queries and code snippets might fail due to two reasons. First, in real-world benchmarks, the Natural Language (NL) queries are written to summarize long and complex code snippets, such as the study of Barone et al. [23]. NL query is a type of code documentation used to explain how its source code works with short descriptions as summarization. The lack of mapping between each Line of Code (LOC) to its description in code snippets of CAT benchmark might reduce the quality of NMT translation models on datasets in CAT benchmark compared to pseudocode-to-code translation. Second, NMT usually outputs incomplete code snippets that require error localization and fixing [17]. It considers the output a sequence of code tokens instead of an Abstract Syntax Tree (AST) representation. Unlike translation, code retrieval by code search [12] can return complete code. The idea of code search is to consider the input in the form of NL description from developers as a query and each source code snippet in a source code dataset as a candidate. Embedding models such as UniXcoder [12] and GraphCodeBERT (GCB) [13] then learn the representation as vectors for query and candidates. Next, the best candidate for each query is returned by the search process that finds the candidate with the highest similarity measured by its embedding to the query’s embedding.

In this work, we analyze the performance of NMT on learning specific information about the source code by AST’s subset of non-terminal nodes, compared to learning from NL query to code tokens. We propose ASTTrans, an NMT translation engine trained using OpenNMT toolkit [16] to learn the mapping from documentation to our tailored representation of an AST by non-terminal nodes. Then, we build a new code search approach that integrates ASTTrans to the state-of-the-art (SOTA) code search process embedding models [12, 13]. Our experiments show that ASTTrans improves code search using SOTA approaches GraphCodeBERT and UniXcoder thanks to its augmented code search process. We use four datasets of the CAT benchmark [31], for our evaluation. Overall, our contributions are as follows:
(1) We analyze and demonstrate NMT in learning our tailored representation of AST compared to learning the sequence of code tokens.

(2) We build a query-to-ASTTrans Representation model and integrate its output to improve the accuracy of the SOTA code search models GraphCodeBERT [13] and UniXcoder [12] and achieve up to 3.08% MRR improvement on TLC dataset and 1.06% on average on all datasets of CAT benchmark.

(3) We analyze how the parameters of ASTTrans can impact the performance in code search.

(4) We conduct a case study to investigate the reasons when ASTTrans can or cannot improve the code search for SOTA models.

The rest of this paper is organized as follows. In section 2, Motivation Example, we introduce an example of a query/candidate for a code search problem. Section 3 provides background information, summarizing the approaches provided in existing embedding tools GraphCodeBERT[13] and UniXcoder [12]. Section 4 shows definitions related to our proposed representation of AST. Section 5 describes in detail our approach to integrating ASTTrans into the original code search process by SOTA approaches. Section 6 mentions our experiments, including configurations, metrics, and results of our proposed research questions. In section 7, we conduct a case study about when ASTTrans can/cannot improve original models. The remaining sections are Related Work, Threats to Validity, and Conclusion. The replication package is available here1.

2 MOTIVATION EXAMPLE

Figure 1 shows a motivation example of a code search process. Users inputs query described in NL. The output for code search's users is the list of candidates as code snippets sorted by their relevancy to the requirement specified by the query. A good code search system tends to return the correct candidate corresponding with a query as the first (called the top-1) candidate of the output list of candidates. In the following example in Figure 1, this query asks how to perform the add function to a Map object in Java. The correct candidate (Candidate 1) accepts the key, the value as a pair to add, and the map as arguments of a method declaration. It performs the check for the validity of the key/value pair before putting it onto the map object. The incorrect candidate (Candidate 2) attempts to put two pairs on a newly constructed map, which doesn't satisfy the requirement provided by the input query. The query, Candidate 1 and Candidate 2, are extracted from the TLCodesum dataset of the CAT benchmark [31]. By UniXcoder [12], the result of this query returned candidate two as the top-1 candidate of the output list, meaning UniXcoder returned incorrectly for this query.

Non-terminal nodes of AST. The sub-ASTs of Line 3 of Candidate 1 and Candidate 2 are shown on the right side of Figure 1. We use AST-treesitter [1] for AST generation for these candidates. The sequence of terminal nodes of an AST of a code snippet generated by AST-treesitter is the sequence of code tokens of that snippet. We have two observations from this example. First, the differences between Line 3 of the two candidates are shown at both the terminal and non-terminal levels. In Candidate 1, non-terminal nodes of Line 3 represented an if statement with information about two nodes of type binary_expression nested in a sub-AST with the root as another node with type binary_expression (node C_1_6). In Candidate 2, a node with type local_variable_declaration is the ancestor of a method_invocation (node C_2_9). Second, while code tokens can be considered a sequence of terminal nodes, we can also represent the ancestors of code tokens by a set of non-terminal nodes. For Candidate 1 and Candidate 2, the sets of parent nodes that can generate all terminal nodes for Line 3 of Candidate 1 and Candidate 2 can be shown in Figure 1. While there are 14 terminal nodes in the sub-AST of Line 3 of Candidate 1 and eight terminal nodes in the sub-AST of Line 3 of Candidate 2, there are only seven nodes as parent nodes of terminal nodes for Candidate 1 and five nodes represented for Candidate 2.

3 BACKGROUND

Natural Language to Code Search (Code Retrieval). The process of code search by SOTA approaches [12, 13] is done in two steps (see steps 1 and 2 of Figure 2). The inputs of code search are a query in NL and a list of candidates as code snippets. In step 1, the embedding of the query and embeddings of candidates are generated. In step 2, the cosine similarities between the query’s and each candidate’s embedding are calculated into a matrix of similarities. Based on this matrix, the candidates will be sorted descendingly by the cosine similarity between their embedding and the query’s embedding. The output of code search is the list of candidates so that the higher a candidate is ranked in the list, the more relevant to the query it is. The best candidate suggested by the embedding model is the top-1 candidate by the code search process. In these steps, the most important step is generating vectors for queries and candidates. We use two SOTA models for this step: GraphCodeBERT [13] and UniXcoder [12].

GraphCodeBERT (GCB) and UniXcoder. Applications as downstream tasks by GraphCodeBERT and UniXcoder are built by two processes: building their pre-trained models by pre-training tasks and fine-tuning them. GraphCodeBERT uses a data-flow graph from AST, which highlights the roles of variables to be an input of pre-training tasks. UniXcoder accepts the input as the flattened sequence of all non-terminal and terminal nodes of AST of the source code candidate for pre-training tasks.

4 SEQUENCE-BASED NON-TERMINAL NODES REPRESENTATION FOR AST

This section presents definitions for our tailored representation of AST.

Definition 4.1 (ASTTrans Representation at Depth-K of a Terminal Node). Given a terminal node t inside an AST a, the representation at depth-k of t, called DepRep(t,a,k), is the ancestor that has its depth (to the root of a) equal to k if the depth of t is greater than k, or the parent node of t if the depth of t is less than or equal to k. We can formulate this function by Formula 1:

\[
\text{DepRep}(t, a, k) = \begin{cases} 
\text{ancestors}(t, a)[k] & \text{if } k \leq \text{length}(\text{ancestors}(t, a)) \\
\text{parent node of } t & \text{if } k > \text{length}(\text{ancestors}(t, a))
\end{cases}
\] (1)

In Formula 1, the ancestors(t, a) function returns the path from the root of a to the parent node of t with the root of a as the
first element of the list (i.e., ancestors(t, a[1]). Using Definition 4.1, the DepRep() of node C_1_18 (called node t) in Figure 1, the sub-AST (called a) that have the root node as node C_1_1 in Candidate 1 and depth k = 2 is the node C_1_6 (the node with type binary_expression). Since the depth from t to the root node of the sub-AST a in Candidate 1 is 5, the DepRep(t, a, 6) (and the same with k > 6) is node C_1_12, which is the parent node of t.

**Definition 4.2** (ASTTrans Textual Representation of a Non-terminal Node). The ASTTrans Textual Representation of a non-terminal node n of AST a, is the mapping function from an AST non-terminal node to a string of tokens, called TextRep(n, a) is defined by Formula 2:

\[
\text{TextRep}(n, a) = \text{type}(n)\#L \sum_{c \in \text{children}(n, a)} \text{type}(c)\#R \tag{2}
\]

Given an AST non-terminal node, this formula integrates the node type and grammatical structure into a textual representation. For example, the TextRep of node C_1_6 in example shown in Figure 1 is the string: ‘binary_expression. paren_expression & &#R paren_expression’.

**Definition 4.3** (ASTTrans Sequence of Nodes Representation at Depth-K of an AST). The ASTTrans Sequence of Nodes Representation at depth-k of an AST a called the NodeSeq(a, k), is a sequence of nodes defined by Formula 3:

\[
\text{NodeSeq}(a, k) = \text{set} \left( \sum_{l \in \text{leaves}(a)} \text{DepRep}(l, a, k) \right) \tag{3}
\]

The NodeSeq(a, k) function returns the set of nodes with max depth (from the root of the AST a) as k for input AST a. Since multiple leaves can have a common ancestor node, the set() function filters repetitive nodes in the output of the NodeSeq() function. For the sub-AST a defined in Candidate 1 in Figure 1, we have NodeSeq(a, 0) as {C_1_1}, since with depth k = 0 and sub-AST a, all terminal nodes of a can be generated from their root node C_1_1. The returned nodes from NodeSeq(a, 1) are \{C_1_1, C_1_3, C_1_4\}.

**Definition 4.4** (ASTTrans Textual Representation at Depth-K of an AST). The ASTTrans Textual Representation at depth-k of an AST a, called the TextSeq(a, k), is a sequence of tokens generated following Formula 4:

\[
\text{TextSeq}(a, k) = \sum_{n \in \text{NodeSeq}(a, k)} \text{TextRep}(n, a) \tag{4}
\]

Our work focuses on building models for learning ASTTrans Textual Representation at depth k from an NL query.
configuration, we set \( k = 5 \). We call this textual representation of AST ASTTrans Representation.

5 APPROACH

5.1 Overview

We propose an approach that integrates our ASTTrans Representation into the original code search process. In summary, we design a separate module of code search called the augmented code search process, which is done in parallel with the original code search process. The output as matrices of similarities generated by original code search and augmented code search processes are combined to contribute a combined matrix of similarities used to sort the code candidates by their relevancy to the given query.

![Figure 2: Overview of the Combined Code Search phase from the Original/ SOTA Code Search process using GraphCodeBERT/UniXcoder and the Augmented Code Search process using ASTTrans](image)

We show in detail how the augmented code search process supports the original code search process through five steps shown in the overview architecture in Figure 2. We inherit steps 1 and 2 from the SOTA approaches [12, 13]. In step 3, from a query and candidates, the augmented embedding as vectors for that query and candidates are generated by ASTTrans. A well-augmented embedding model requires that the vector representation of an NL query should have higher similarity to its correct candidate’s vector than incorrect candidates’ vectors. In step 4, the results of comparing a query’s augmented embedding to each candidate’s augmented embedding are calculated and logged in the similarity matrix of the augmented code search. In this problem, we use cosine similarity [8] as the metric for comparison. In steps 4 and 5, the matrix of similarity generated by augmented code search, called augmented similarity matrix, is combined with the matrix of similarity by original code search models (i.e., GraphCodeBERT and UniXcoder) and becomes a so-called Combined Similarity Matrix. Each element in this combined similarity matrix is the score of the similarity comparison between a query and each candidate. The final output of this combined code search model is the list of candidates sorted using the combined similarity matrix. While step 4 is similar to step 2 except for using different input, we discuss steps 3 and 5 below.

5.2 Generating Augmented Embedding for Queries by Neural Machine Translation

This module accepts the input as the query written in natural language. The expected output is the embedding as the corresponding ASTTrans representation of its respective code, i.e., the ASTTrans Representation of the correct candidate for the query. Since sequence-to-sequence translation can be solved successfully by Neural Machine Translation in prior works [25, 34, 36], we apply NMT as a sub-module to handle this task. Two sub-modules are used for this module: NMT learning from query-to-ASTTrans Representation as a sequence of tokens and the vectorization from the sequence of tokens to vector using fastText library [7].

5.2.1 Query-to-ASTTrans Representation. The query-to-ASTTrans Representation by NMT is built in two phases. In the first phase, the training model is built to learn the mapping between natural language query and sequence of tokens as textual information from ASTTrans Representation. We build training models for datasets in the CAT benchmark [31]. We inherit two advantages of learning with NMT that we illustrate in Figure 3. First, NMT allows learning with complex textual sequences by an encoder-decoder paradigm. It includes two layers of hidden units for encoding text in the source language to embedding representation, and the other two layers decode that embedding to textual representation in the target language. While older machine translation models such as Statistical Machine Translation (SMT) [25] attempt to generate each sentence from phrase to phrase, NMT allows learning from longer units such as sentences or paragraphs. The NMT model considers the translation process as a continuous token generation process. Inside it, each token in the target language is generated based on the contextual information of the previous tokens in the target language and the sequence of tokens in the source language. This advantage is achieved by the attention mechanism, a module connecting the learned information from the context of source and target tokens.
The output of the training phase is a trained model that can predict the ASTTrans Representation (defined in Formula 4) from the input query. In the second phase, these trained models are used to predict the sequence of text as ASTTrans Representation for unseen natural language queries. We use OpenNMT [16] to train our models for query-to-ASTTrans Representation.

5.2.2 Text-to-Vector Conversion. The augmented embedding for query is completed with a module to handle the output of NMT as a sequence of tokens as the predicted ASTTrans Representation (see Figure 3). We use fastText [7] as the library for text-to-vector conversion for augmented embedding generation for each query. We train the models for vector generation of fastText with un supervised mode. We use the ASTTrans Representation from candidates in four datasets of CAT benchmark [31] to train fastText’s models.

5.3 Generating Augmented Embedding for Candidates by AST Extraction

An important rule of code search approaches implemented by original embedding models such as GraphCodeBERT [13], and UniXcoder [12] is that while the query contains only information about the natural language description of the code, the candidates contains only information about the source code. Thus, while building the augmented embedding model, we must follow this rule that the augmented embedding from the query accepts only the input as a natural language description of the query while the augmented embedding of the candidates accepts the source code representations of the candidate as the input. In our design selection, while the augmented embedding of the query is the predicted ASTTrans Representation of the correct candidate of source code, the augmented embedding of a candidate is the expected ASTTrans Representation of it.

Algorithm 1: Algorithm for extracting augmented embedding for code candidate

```python
Input : candidate: string, depthSize: int, modelEmb: Model
Output: vectorASTRep: Vector

astCandidate ← generateASTForCode(candidate)
listOfLeaves ← extractLeaves(astCandidate)
seqNodeASTReps ← set()
seqTextASTReps ← []
for leaf ∈ listOfLeaves do
    nodeRep ← findRepresentation(leaf, depthSize)
    if nodeRep ∈ seqNodeASTReps then
        seqNodeASTReps.add(nodeRep)
        seqTextASTReps.append(getTextRep(nodeRep))
    end if
end for
vectorASTRep ← modelEmb.getVector(seqTextASTReps)
return vectorASTRep
```

Pseudocode for extracting the embedding of a candidate can be shown in Algorithm 1. First, the AST of the candidate is generated by the `generateASTForCode()` function. Next, all the leaves of the AST are extracted by function `extractLeaves()`. From Line 3 to Line 11, a loop through each terminal node of AST is run to extract the set of nodes that can represent the AST at depth `depthSize` and its corresponding textual information. The `findRepresentation()` function implements the concept of Definition 4.1. For each node representation extracted in Line 6, its textual information is extracted by the function `getTextRep()`, which implements Definition 4.2. The ASTTrans textual representation of the candidate’s AST at depth `depthSize` will be transformed into a vector representation in Line 12. For text-to-vector conversion, we also use fastText [7] for this task. We use the same model Similar to the vector generation for the augmented embedding of queries, augmented vectors for candidates are generated from trained model of the AST Representation of candidates in training sets of CAT benchmark [31].

5.4 Calculating Combined Similarity Matrix

After generating the embedding using original embedding models (by GraphCodeBERT and UniXcoder) and using ASTTrans for a query and a list of candidates, the similarities between the vector of the query and the vector of each candidate are calculated in both the original and augmented code search process. For each pair of query-candidate, we use the cosine similarity [8] for measuring their similarity. The output of the original code search process is the original similarity matrix `matrix_simorg`, and the output of the augmented code search process is the augmented similarity matrix `matrix_simaug` (see Figure 2). We combine these matrices to produce the combined similarity matrix for code search phase with ASTTrans by the following formula:

\[
\text{matrix\_sim}_{\text{com}} = \text{matrix\_sim}_{\text{org}} \ast (1 - w) + \text{matrix\_sim}_{\text{aug}} \ast w \quad (5)
\]

Selecting combined weight \(w\). In Formula 5, the weight \(w\) represents the ratio from zero to one that the code search by augmented code search process can contribute to the original code search process. We select the weight that returned the best accuracy in the augmented code search process on the validation sets of CAT benchmark [31]. The selected standard weight \(w\) for matrix combination is up to \(w = 0.1\).

From the combined similarity matrix `matrix\_sim_{com}`, the list of candidates for an input query is sorted by the score between each candidate and the query. Candidates more relevant to the query appear at a higher rank than unrelated candidates.

6 EXPERIMENTS

In the experiment, we attempt to answer the following research questions (RQs):

(1) RQ1. How well can NMT perform in learning ASTTrans Representation?
(2) RQ2. Can code search benefit from query-to-ASTTrans Representation?
(3) RQ3. How can the parameters of ASTTrans affect the performance of code search?

6.1 Datasets

We use the CAT benchmark [31] with four datasets of NL queries and corresponding implementation of method declarations to evaluate ASTTrans. Prior works show that code search datasets have noisy data, including erroneous code documentation/ NL queries
Table 1: Number of entities (pairs of query-candidate) for training/validation/test sets for four datasets of the CAT benchmark

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>Training</th>
<th>Validation</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLC</td>
<td>java</td>
<td>53592</td>
<td>7561</td>
<td>7584</td>
</tr>
<tr>
<td>CSN</td>
<td>java</td>
<td>323225</td>
<td>8849</td>
<td>19317</td>
</tr>
<tr>
<td>Funcom</td>
<td>java</td>
<td>1184437</td>
<td>20000</td>
<td>20000</td>
</tr>
<tr>
<td>PCSD</td>
<td>python</td>
<td>57846</td>
<td>19000</td>
<td>19028</td>
</tr>
</tbody>
</table>

[31]. Si et al. [31] proposed a systematic approach to filter noisy data. They study the four datasets to identify the templates of erroneous parts inside each NL query. They published a clean version of them. There are three Java datasets in this benchmark, including TLCodeSum (TLC) (which is the clean version of the original TLC dataset proposed in [14]), Funcom (the clean version of [22]), CodeSearchNet (CSN) (the clean version of [15]) and one Python dataset named PCSD (the clean version of [33]). Statistics on four datasets can be shown in Table 1.

6.2 Configurations

6.2.1 OpenNMT. We set up the machine translation model for inferring ASTTrans Representation from the query with the following configurations. We use two sets of neural network layers for training: the encoder and decoder layers. Each module (encoder/decoder) has two layers with 500 hidden units per layer. The gate type to use in each hidden unit is Long Short Term Memory (LSTM). We choose this gate type because LSTM has been proven as an efficient Recurrent Neural Network (RNN) model that can learn and capture the relationship between words in a long textual sequence [16, 36]. We use 100000 steps for training, with a validating step performed for every 1000 steps. A checkpoint will be saved for every 10000 steps. In other parameters, we use the default settings from OpenNMT [16].

6.2.2 fastText. We use fastText [7] for text-to-vector conversion in the augmented embedding of queries and candidates. Prior work [32] shows that fastText [7] is not only an efficient embedding model but also able to embed a sequence of text with better quality compared to other well-known models such as Doc2Vec [18] and TF-IDF [5] in many NLP problems. In experiments, we use the augmented dimension size as 100 and the skip-gram method in fastText library for training and generating vectors for the augmented code search process.

6.2.3 Original Embedding Models. We use GraphCodeBERT [13] and UniXcoder [12] as the SOTA approaches. We use the pre-trained models for Java and Python developed by the authors of GraphCodeBERT and UniXcoder, Guo et al. [12, 13]. We have had a few discussions with the SOTA approaches’ authors about the configurations of the SOTA code search process. They confirmed that there are two settings for the code search process: without fine-tuning and with fine-tuning. The configuration without fine-tuning is called zero-shot learning and always performs much less accurately than the fine-tuning setting, although the experiment without fine-tuning doesn’t require a costly fine-tuning step. We select the fine-tuning setting since it reveals the best capability of SOTA approaches. We fine-tune the pre-trained models of GraphCodeBERT and UniXcoder by their proposed dataset CodeSearchNet (CSN - the full version [15]). We use the fine-tuned models to generate the vectors for queries and candidates of TLC, CSN (the clean version from [31]), Funcom, and PCSD datasets. Our experiments run on four datasets’ test sets (see Table 1).

6.2.4 ASTTrans. For the standard configuration, we train four query-to-ASTTrans Representation models on four datasets in the CAT benchmark [31]. There are two important parameters for ASTTrans: the depth of ASTTrans Representation and the combined weight between original and augmented similarity matrices. We set the depth \( k \) of ASTTrans Representation as the depth size \( k = 5 \) and the combined weight \( w \) as \( w = 0.1 \).

6.2.5 Embedding Size of original embedding models. The default dimension size (called \( dim \)) of a vector generated by GraphCodeBERT and UniXcoder is 768, which is much bigger than other works [18, 28]. Reducing the dimension size by Principal Component Analysis (PCA) [3] can improve running time up to five times faster with the reduced dimension size \( \text{dim} = 20 \) for our code search task, as we observe in the experiments. For experiments of RQ2 and RQ3, we use two dimension sizes for \( \text{dim} = 20 \) and \( \text{dim} = 768 \). Both original models’ training and testing steps and augmented models’ training and testing steps were trained on a Linux computer with 96 GB of RAM using a Core-i9 processor with 16 cores and an RTX-3080 GPU card with 24GB of RAM.

6.3 Metrics for Evaluation.

6.3.1 Evaluating Neural Machine Translation Model. Prior works [10, 30] show that the BLEU score, a well-known metric for evaluating NMT in NLP, has drawbacks in evaluating the quality of translated SE artifacts, such as the sequence of code tokens. Pradel et al. [10] propose an approach for filtering repetitive n-grams for calculating textual similarity for SE artifacts. They define a new metric named CrystalBLEU-4 (number four stands for four-gram cumulatively, the default configuration of CrystalBLEU). Besides, in well-known NLP metrics, Roy et al. [30] show that Meteor [9] score can perform better than BLEU score by an evaluation based on human judgment. We choose CrystalBLEU-4 and Meteor as metrics for evaluating the performance of NMT for RQ1.

6.3.2 Evaluating Effect of ASTTrans in Code Search. Mean Reciprocal Rank (MRR) is used for code search evaluation in many approaches [12]. The effect in MRR over code search on set \( C \) of cases by ASTTrans embedding \( a \) to original embedding model \( o \) with original embedding size \( d \), is calculated by Formula 6:

\[
\text{EffectMRR}(C, a, o, d) = \text{MRR}_{\text{com}}(C, a, o, d) - \text{MRR}_{\text{org}}(C, o, d) \quad (6)
\]

In this Formula, \( \text{MRR}_{\text{org}} \) is the Original MRR returned by the original code search. \( \text{MRR}_{\text{com}} \) is the Combined MRR returned by code search with the combined similarity matrix for a specific set of queries \( C \), the original model \( o \) reduced to dimension size \( d \) by PCA [3] and the augmented model \( a \). If the score of this metric is positive, it means the augmented code search process improves the accuracy of the original code search process by MRR. We define the metric \( \text{AverageEffectMRR}(C, a) \ (\text{Avg. Eff.}) \) on a set of queries \( C \) using an
Table 2: RQ1. Comparison of NMT’s performance on Query-to-ASTTrans Representation versus Query-to-Code Tokens on CAT Benchmark [31]

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query-to-ASTTrans Rep.</th>
<th>Query-to-Code Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vocab.</td>
<td>Crystal BLEU-4</td>
</tr>
<tr>
<td>TLC</td>
<td>234</td>
<td>0.51</td>
</tr>
<tr>
<td>CSN</td>
<td>265</td>
<td>0.22</td>
</tr>
<tr>
<td>Funcom</td>
<td>274</td>
<td>0.28</td>
</tr>
<tr>
<td>PCSD</td>
<td>196</td>
<td>0.29</td>
</tr>
</tbody>
</table>

ASTTrans model $a$ by Formula 7:

$$Average_{EffectMRR}(C, a) = \frac{\sum_{d \in \{GC, BU, nX\}} EffectMRR(C, a, d)}{4}$$  \tag{7}$$

6.4 RQ1. How well can NMT perform in learning ASTTrans Representation?

The result for RQ1 is shown in Table 2. From the translated result of the test sets in four datasets, we see that the inference from query to code tokens confronts challenges for Neural Machine Translation. Due to the large vocabulary in the training sets, NMT achieves low accuracy measured by the CrystalBLEU-4 and Meteor scores. The best dataset for Query-to-Code Tokens translation is TLC, with the scores with a CrystalBLEU-4 score of 0.16. This dataset included over 216000 distinct tokens in their training set of 53592 candidates. Similarly, other datasets in the Java language, such as CSN and Funcom or the Python dataset PCSD, also achieve low scores by CrystalBLEU-4 and Meteor. These results confirm our assumption in practical datasets of real-world queries and source code, such as the CAT benchmark, sequences of code tokens are too complicated for the NMT model to learn their information from NL queries.

While there was low accuracy when NMT was used for Query-to-Code Tokens translation, Table 2 shows that Query-to-ASTTrans Representation achieves much better accuracy. Compared to the accuracy of learning code tokens, the NMT model to learn the sequence of non-terminal nodes achieves the highest accuracy in translation of the TLC dataset, while it achieves the lowest accuracy on the CSN dataset. NMT can perform more than 3x better on learning sequences of non-terminal nodes than sequences of terminal nodes as code tokens in terms of CrystalBLEU-4 score for the TLC dataset at the score of 0.54. The output of ASTTrans brings the CrystalBLEU-4 score at 0.28 in the Funcom dataset, which is 9x better than learning code tokens. With the Python dataset PCSD, ASTTrans also achieves a significantly higher CrystalBLEU-4 score. The similarities between predicted and expected results measured by Meteor are also consistent with the first metric. One of the reasons for these improvements is that our representation of AST requires a vocabulary of types of non-terminal nodes, and its size is significantly less than the vocabulary size of learning code tokens.

Table 3: RQ2. Comparison between combined code search and original code search on CAT Benchmark [31] with metrics: Combined MRR (Com.), Original MRR (Org.), Effect in MRR (Eff) and AverageEffectMRR (Avg. Eff.)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Dim.=20</th>
<th>Dim.=768</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCB</td>
<td>54.18%</td>
<td>1.59%</td>
</tr>
<tr>
<td>UniX</td>
<td>54.44%</td>
<td>1.33%</td>
</tr>
<tr>
<td>CSN</td>
<td>36.16%</td>
<td>0.10%</td>
</tr>
<tr>
<td>Funcom</td>
<td>35.11%</td>
<td>0.28%</td>
</tr>
</tbody>
</table>

6.5 RQ2. Can code search benefit from query-to-ASTTrans Representation?

The results of RQ2 are shown in Table 3. Our augmented embedding approach for queries and candidates brings positive effects (Eff) in MRR for both test sets of four datasets. ASTTrans got the best accuracy in the TLC dataset for three Java datasets, which has the AverageEffectMRR as 3.08% for four configurations. For a low dimension size of 20, the performance using the original (Org.) embedding models decreases to 49.71% for GraphCodeBERT and 50.12% for UniXcoder. With the augmented code search process, the combined (Com.) code search improved by 4.47% for GraphCodeBERT and 4.33% for UniXcoder in MRR for this embedding size. With the high embedding size of the original model as 768, ASTTrans’s augmentation improved the MRR to 1.93% for GraphCodeBERT and 1.59% for UniXcoder. For CSN, ASTTrans has positive effects with 0.1% improvement in MRR on average. On the CSN dataset, ASTTrans performs best with a dimension size of 20 and embedding model as UniXcoder. ASTTrans improves the MRR to 0.15% on average for the Funcom dataset. For this dataset, we achieve positive EffectMRR scores for both four configurations using GraphCodeBERT and UniXcoder.

ASTTrans also improves the accuracy of code search for the Python dataset PCSD with 19028 queries. It positively affects the MRR for both low-dimension and high-dimension sizes of the original embedding and for both original models. For low dimension size, ASTTrans improves 1.02% in MRR for GraphCodeBERT and
Table 4: RQ3 Part 1: Results by metric \textit{AverageEffectMRR} of configuration using Concatenated Embedding

<table>
<thead>
<tr>
<th>Dataset</th>
<th>\textit{AverageEffectMRR}</th>
<th>Dataset</th>
<th>\textit{AverageEffectMRR}</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLC</td>
<td>0.16%</td>
<td>Funcom</td>
<td>0.06%</td>
</tr>
<tr>
<td>CSN</td>
<td>-0.03%</td>
<td>PCSD</td>
<td>0.04%</td>
</tr>
</tbody>
</table>

1.45% in MRR for UniXcoder. With a high dimension size of 768, while the original models achieve competitive accuracies of 73.38% and 76.3% for GraphCodeBERT and UniXcoder, ASTTrans can still improve the code search up to 0.73% of \textit{EffectMRR}. ASTTrans increases by 0.91% the MRR of code search over the PCSD dataset.

Summary of RQ2: ASTTrans improves the SOTA approaches with an average of 1.06% MRR improvement in code search on CAT benchmark.

6.6 RQ3. How can the parameters of ASTTrans affect the performance of code search?

In this RQ, we attempt to validate the performance of ASTTrans in improving code search for SOTA embedding models with different settings of ASTTrans’s parameters. We select the following parameters.

6.6.1 Concatenating Vectors versus Combining Similarity Matrices.

In the default configuration, we create two matrices representing two code search processes. While the original code search process uses original embedding models, the output of the augmented code search process as the augmented similarity matrix is calculated by the augmented vectors of queries and candidates (see Figure 2). There is another strategy to combine the embedding of ASTTrans Representation. In this alternative strategy, we concatenate elements of embedding in ASTTrans to the vectors generated by original embedding models. For example, given a query \( q_1 \), the original embedding \( \mathbf{n} = \{n_1, n_2\} \) and the augmented embedding \( \overset{\top}{\mathbf{n}} = \{n_1', n_2'\} \), the concatenated vector for code search is \( \mathbf{n} = \{n_1, n_2, n_1', n_2'\} \). The concatenated vectors are then used in a single code search process. In the first part of RQ3, we change the strategy of integrating ASTTrans by using concatenated vectors and doing the code search experiment.

We show the \textit{AverageEffectMRR} of the code search with the vectors of queries and candidates by concatenated embedding in Table 4. The effect of MRR on four datasets significantly decreased in this configuration. The TLC, Funcom, and PCSD datasets still positively affect MRR scores in code search. Concatenated embedding strategy from ASTTrans has a negative effect on the code search on CSN dataset. The average of \textit{AverageEffectMRR} over four datasets in this configuration is 0.06% compared to 1.06% of the standard configuration of ASTTrans.

Summary of RQ3(1): ASTTrans performs better using the combined similarity matrix than concatenated embedding.

6.6.2 Combined weight between matrices. In the second part of this RQ, we run the code search with different weights as 0.2, 0.3, and 0.4 to analyze the performance of ASTTrans with different combined weights. We illustrate this variation by Table 5. We show that with a weight higher than 0.1, the augmented code search process caused negative impacts on the original model for three datasets Funcom, CSN, and PCSD. With a weight of 0.2, the augmented embedding positively impacted the TLC dataset. It shows that although ASTTrans can improve code search, integrating ASTTrans in code search require a proper approach to adjust the attention weight of their contribution to the final output of code search.

Summary of RQ3(2): For combined weight \( w \) varies up to 0.4, ASTTrans performs best with \( w = 0.1 \).

6.6.3 Depth of ASTTrans Representation. While the default depth of ASTTrans Representation is \( k = 5 \), we demonstrate how well a sequence of non-terminal nodes with different depths from the root can improve code search. A sequence of non-terminal nodes close to the root node of an AST might be easier for NMT to learn the information due to its simplicity. However, it has the risk of not giving enough information to distinct candidates. For example, the ASTTrans Representation at depth \( k = 0 \) returns the root node for every candidate, meaning that representation cannot differentiate candidates for code search. Since the non-terminal nodes at \( k < 2 \) are too abstract, we select the range of \( k \in \{2 - 9\} \) to evaluate the code search process.

The average of \textit{AverageEffectMRR} Result over four datasets by augmented embedding models trained from different depths for ASTTrans Representation is shown in Figure 4. Overall, ASTTrans performs best in improving code search with our standard configuration as \( k = 5 \). With the depth \( k \) higher than 5, the improvement of \textit{AverageEffectMRR} in four datasets tends to decrease slowly. The
Table 6: Percentages (Perc.) and average of CrystalBLEU-4 (CB) Cyclomatic Complexity (CC) of Improved, Decreased and all cases by ASTTrans to original models

<table>
<thead>
<tr>
<th></th>
<th>Improved</th>
<th>Decreased</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CB</td>
<td>CC</td>
<td>CB</td>
</tr>
<tr>
<td>TLC</td>
<td>26.99%</td>
<td>0.64</td>
<td>3.62</td>
</tr>
<tr>
<td>CSN</td>
<td>17.49%</td>
<td>0.25</td>
<td>4.18</td>
</tr>
<tr>
<td>Fun.</td>
<td>21.91%</td>
<td>0.36</td>
<td>1.92</td>
</tr>
<tr>
<td>PCsD</td>
<td>25.91%</td>
<td>0.42</td>
<td>4.65</td>
</tr>
<tr>
<td>Avg</td>
<td>23.08%</td>
<td>0.42</td>
<td>3.59</td>
</tr>
</tbody>
</table>

Summary of RQ3(3): For the depth of AST Representation $k \in \{2 – 9\}$, ASTTrans performs best at $k = 5$ and worst at $k = 2$.

7 CASE STUDY

From the results in RQ2, we do a head-to-head comparison between the original results and the results from our combined code search process for each query. We call a triple of three elements: a query, a suggested list of candidates from the original code search process and a suggested list from the combined code search process as a case of search. There are three possible outputs of cases: Improved (means ASTTrans improved the rank of the correct candidate of the query), Same (the ranks are the same), and Decreased (the combined process returned better results).

**Percentage and CC.** We measure the Cyclomatic Complexity (CC) using the Python library HFCCA [2]. We have two observations from Table 6. First, the cases ASTTrans that can improve the original model are more than the cases ASTTrans that decrease the original accuracy in both datasets. Second, ASTTrans can improve the cases with relatively higher CC on the datasets at 3.59 compared to the average CC of all cases at 3.52. However, ASTTrans still has challenges on solving very complex code snippets, shown by the CC of Decreased cases of 3.72.

**Cases when ASTTrans improved/ worse the results of SOTA approaches.** We found out that ASTTrans always improved the accuracy of code search when the quality of NMT on Query-to-ASTTrans Representation is high. The average CB score for the Improved cases by ASTTrans over four datasets is 0.42. The average CB score for Decreased cases is 0.15. We got the highest CB score of Improved cases for the TCLCodeSum dataset as 0.64 and the lowest CB score of Improved cases for the CodeSearchNet dataset as 0.25.

8 RELATED WORK

Research on code embedding models has been proposed for at least a decade [4, 6, 11, 21, 27, 35]. Before these tools, NLP models were used to represent source code API. Nguyen et al. propose API2Vec [27], a tool for learning the embedding of Java APIs and CSharp APIs based on Word2Vec [24]. Guo et al. build CodeBERT [11], the pre-trained model for natural and programming languages.

CodeBERT aims to strengthen its models by pre-training tasks such as Mask Language Modeling to prepare it for other downstream tasks like code search and code translation. CodeReviewer [19] is a code embedding approach for representing code changes and reviews. code2vec [6] is another embedding tool for SE tasks such as method name generation. Other embedding models for code search are CodeTS [35], which learns information of paths of AST, and PLBART [4] uses sequence-to-sequence denoising pre-training for code representation.

Besides Neural Machine Translation, other machine translation techniques have also been applied in SE. Two of them are Statistical Machine Translation (SMT) [20], and CodeGPT [21]. Compared to NMT, SMT has been considered an older approach based on learning the mapping by phrases-to-phrases from the source language to the target language. SMT has been applied in code translation [25], type inference [29], and behavior exception mining [26]. CodeGPT is considered the newer approach for applying translation in SE. CodeGPT provides different mechanisms for different problems of translation.

9 THREATS TO VALIDITY

We identify several threats to the validity of our work. The threats to internal validity relate to biases and errors in experiments. Since authors of GraphCodeBERT [13] and UniXcoder [12] didn’t publish their fine-tuned models for code search, we re-trained their models on Java and Python programming languages. This could pose the threat that the SOTA models might have different performances. We asked the authors of GraphCodeBERT and UniXcoder about the validity of our configurations for fine-tuning. They confirmed that our configurations are correctly set and that our reported results of experiments on the CAT benchmark are reasonable. The threats to external validity are about the generalizability of our work. While our evaluation is only in Java and Python datasets, our algorithm for integrating ASTTrans to original code search models is language-independent and can be applied easily to other programming languages.

10 CONCLUSION

In this research, we analyze to optimize NMT by learning from the summarization of the Abstract Syntax Tree by a set of non-terminal nodes. We demonstrate that NMT can learn our AST sequence-based representation for non-terminal nodes more accurately than terminal nodes. Moreover, our proposed query-to-ASTTrans Representation model, ASTTrans, can improve the accuracy of code search by original embedding models GraphCodeBERT [13] and UniXcoder [12] on datasets of CAT benchmark as up to 3.08% MRR improvement. In the future, we will further analyze the performance of other machine translation models such as CodeGPT [21] in learning our ASTTrans Representation and improving ASTTrans Representation for other SE tasks such as code summarization.

ACKNOWLEDGEMENT

This research was supported by the National Science Foundation under Grant number 2211982. We would also like to thank the ResearchIT team at Iowa State University for their constant support.

2https://researchit.las.iastate.edu/