Leveraging Statistical Machine Translation for Code Search

Hung Phan  
Iowa State University  
Ames, Iowa, USA  
hungphd@iastate.edu

Ali Jannesari  
Iowa State University  
Ames, Iowa, USA  
jannesar@iastate.edu

ABSTRACT
Machine Translation (MT) has numerous applications in Software Engineering (SE). Recently, it has been employed not only for programming language translation but also as an oracle for deriving information for various research problems in SE. In this application branch, MT’s impact has been assessed through metrics measuring the accuracy of these problems rather than traditional translation evaluation metrics. For code search, a recent work, ASTTrans, introduced an MT-based model for extracting relevant non-terminal nodes from the Abstract Syntax Tree (AST) of an implementation based on natural language descriptions. While ASTTrans demonstrated the effectiveness of MT in enhancing code search on small datasets with low embedding dimensions, it struggled to improve the accuracy of code search on the standard benchmark CodeSearchNet.

In this work, we present Oracle4CS, a novel approach that integrates the classical MT model called Statistical Machine Translation to support modernized models for code search. To accomplish this, we introduce a new code representation technique called ASTSum, which summarizes each code snippet using a limited number of AST nodes. Additionally, we devise a fresh approach to code search, replacing natural language queries with a new representation that incorporates the results of our query-to-ASTSum translation process. Through experiments, we demonstrate that Oracle4CS can enhance code search performance on both the original BERT-based model UniXcoder and the optimized BERT-based model CoCoSoDa by up to 1.18% and 2% in Mean Reciprocal Rank (MRR) across eight selected well-known datasets. We also explore ASTSum as a promising code representation for supporting code search, potentially improving MRR by over 17% on average when paired with an optimal SMT model for query-to-ASTSum translation.

CCS CONCEPTS
• Software and its engineering → Software libraries and repositories; Software notations and tools.

KEYWORDS
Abstract Syntax Tree, Statistical Machine Translation, Information Retrieval

ACM Reference Format:

1 INTRODUCTION
Machine Translation (MT) is a set of approaches for solving the problem of translating from one natural language (NL) to another. While earlier works in MT focused on using rules and vocabularies of sentences to produce a sequence of words in the target language from a sequence of words in the source language, machine learning-based methods have been proposed since the 1990s to replace these models with better performance [2]. Statistical Machine Translation (SMT) [21] and Neural Machine Translation (NMT) [46] are two well-known approaches for solving the translation problem using machine learning techniques. While NMT appeared after SMT, SMT has been considered having advantages thanks to its characteristics of requiring less computation resources and learn efficiently from simplified sequence of source code [26].

In the field of Software Engineering (SE), machine learning-based Machine Translation (MT) has found valuable applications in solving research problems that can be framed as language inference challenges. A straightforward application of MT in SE involves code-to-code translation. The underlying concept here is that different programming languages can be treated as source and target languages, similar to how natural languages are handled in traditional MT. Previously, for programming language translation tasks like converting Java to CSharp [25], Statistical Machine Translation (SMT) models were employed. Another problem that MT can address is pseudocode-to-code translation. Previous research has demonstrated that both SMT and NMT perform well in inferring source code, primarily at the statement level, from pseudocode descriptions [16, 27]. Two common characteristics emerge from these studies. First, to ensure the correctness of code snippets, efforts have been made to integrate program analysis into SMT and NMT, enhancing the output of these MT models. Second, various translation evaluation metrics have been introduced to assess the quality of predicted code compared to expected code, including metrics like CodeBLEU [31] and ParaBLEU [44].

With advancements in machine translation models in natural language processing (NLP), their applications have expanded beyond traditional language translation tasks [12, 26, 30]. In these studies, MT techniques are employed as sub-modules, and their outputs serve as critical elements in determining the final output values for research problems beyond translation. Phan et al. introduced StatType [30], a tool that utilizes statistical machine translation to infer the types of incomplete code snippets found on online QA forums like StackOverflow. While StatType’s ultimate output
Abstract Syntax Tree (AST). Subsequently, ASTTrans conducted an
To accomplish this task, StatGen compares the expected generated

vice versa. StatGen serves to check documentation-implementation
information from the full code snippets, potentially impacting the
that their abstracted code representation might have lost crucial
augmented code search processes. This approach carried the risk
plified code representation by utilizing non-terminal nodes from
this limitation in this existing work. First, while ASTTrans sim-
plication to identify discrepancies between the expected code and its
corresponding code documentation. In these studies, MT techniques
function as oracles, predicting valuable information to contribute
to the results of other research problems.
In the context of code search, a recent study known as ASTTrans,
introduced by Phan and Jannesari [29], harnessed Machine Transla-
tion as an oracle for improving the accuracy of code retrieval. This
enhancement was evaluated using the CAT benchmark [33], which
consists of pairs comprising a natural language description as a
query and a source code snippet at the functional level as a candi-
date. The rationale behind this approach lies in the observation that
code search datasets often feature code snippets presented as mul-
tiple lines of code (LOC) alongside very abstract natural language
queries. This abstractness posed challenges for MT models in accu-
rately translating these queries into code. ASTTrans addressed this
by simplifying the MT model’s output to represent the grammatical
structure of non-terminal nodes at specific depths within the code’s
Abstract Syntax Tree (AST). Subsequently, ASTTrans conducted an
alternative code search process, replacing the original source code
representation with the ASTTrans representation before merging
the results with those of the original process. In essence, MT served
as an oracle in ASTTrans, enhancing the code search capabilities of
established models like GraphCodeBERT (GCB) [11] and UniXcoder
[10].
Although ASTTrans managed to enhance the code search process
for various configurations and smaller datasets such as TLC
[13] and PCSD [40], along with reduced embedding sizes for code
and query representations, it fell short when applied to the well-
established CodeSearchNet [14] dataset with the default embedding
size of 768, yielding only a marginal 0.06% improvement in Mean
Reciprocal Rank (MRR). We identified two reasons contributed to
this limitation in this existing work. First, while ASTTrans sim-
plified code representation by utilizing non-terminal nodes from
the AST, this representation might still have been too complex for
the MT model to deliver high accuracy over the CodeSearchNet
benchmark. The inherent complexity of CodeSearchNet may have
posed a significant challenge. Second, ASTTrans introduced a code
search process that combined the results of both the original and
augmented code search processes. This approach carried the risk
that their abstracted code representation might have lost crucial
information from the full code snippets, potentially impacting the
quality of the results. In summary, the oracle functionality offered
by ASTTrans attempted to replicate the work of the original code
search process, which proved challenging for MT models to com-
pete with well-established pre-trained BERT-based code search
models. Consequently, ASTTrans could not provide substantial
improvements in accuracy for state-of-the-art (SOTA) code search
models.
In this project, we aim to address the limitations identified in
prior work ASTTrans [29]. Firstly, we introduce ASTSum, a novel
representation of non-terminal nodes within the AST of source
code. This representation efficiently captures comprehensive AST
information using a limited number of tokens. Secondly, we em-
ploy a Statistical Machine Translation (SMT) approach to create a
query-to-ASTSum Representation model, enhancing the NL query’s
information for the code search process. Thirdly, we present Ora-
cle4CS, an innovative code search approach that builds upon both
the original BERT-based model, UniXcoder [10], and the latest opti-
mized BERT-based model, CoCoSoDa [32]. In our work, we redefine
the role of an oracle as a “satellite” for the code search process, fo-
cusing on forecasting previously unseen information for input to
the code search, rather than attempting an additional round of code
search like ASTTrans. Our experimental results demonstrate that
Oracle4CS enhances code search performance for UniXcoder and
CoCoSoDa across our selected datasets. Furthermore, our ASTSum
Representation exhibits promise as an effective representation for
leveraging MT models to enhance code search. We present the
following key contributions:

(1) **Designation:** We introduce ASTSum Representation, a novel
code representation method that effectively captures crucial
information from Abstract Syntax Trees (ASTs) using a limited
number of code tokens.
(2) **Models:** We implement query-to-ASTSum Representation
models for six programming languages: Java; JavaScript;
Python; Go; PHP; and Ruby.
(3) **Applications:** We propose Oracle4CS, a set of search en-
gines built on top of UniXcoder (called Oracle4CS_{UniXcoder})
and CoCoSoDa (called Oracle4CS_{CoCoSoDa}). These engines
significantly enhance code search accuracy, achieving improve-
ments of up to 2.28% in Mean Reciprocal Rank (MRR)
scores.
(4) **Study:** We conduct experiments to assess the potential of
ASTSum representation in code search improvement, assum-
ing the availability of perfect Statistical Machine Translation
and combined Oracle4CS and State-of-the-Art (SOTA) mod-
els.
The remaining sections of this paper follow a structured pro-
gression. In Section 2, we offer a comprehensive background on
key concepts pertinent to the code search problem and introduce
SOTA code search pipelines. Following this, the Motivation Ex-
ample section illustrates our point with a practical code snippet,
its corresponding query, and the innovative ASTSum Representa-
tion. Section 4 outlines the intricacies of our proposed approach,
Oracle4CS. In Sections 5 and 6, we delve into our experimental as-
sessments of Oracle4CS, including an evaluation of its code search
accuracy and an exploration of the potential applications of AST-
Sum representation. Section 7 delves into the detailed analysis of
the results obtained. We also explore existing research in the field of code search in the Related Work section, address potential threats to the validity of our work in the Threats to Validity section, and conclude by summarizing our findings and contributions in the Conclusion section. Additionally, our replication package is accessible at this site 1.

2 BACKGROUND

Inferring source code from natural language descriptions is a long-standing research challenge in Software Engineering (SE). While machine translation engines have been successfully applied to tasks like code inference from specific types of code documentation, such as pseudocode or behavioral exception specifications [26, 27], addressing practical datasets comprising natural language queries and corresponding implementations extracted from large-scale software repositories required the development of a new approach known as code search to achieve improved performance [29]. This need arises from the inherent complexity of source code at the functional level, contrasted with the brevity of natural language queries, typically consisting of fewer than four sentences. The code search process tackles this challenge through various solutions centered around learning the representations of source code snippets (candidates) and natural language queries (NL queries). Notably, well-known code search pipelines [10, 32, 43] are typically composed of two sub-modules: pre-training and fine-tuning.

2.0.1 Pre-training Stage. The pre-training phase is essentially an unsupervised process that does not rely on a parallel corpus of natural language descriptions and corresponding code snippets. Its core concept involves learning embeddings for code tokens or words, driven by the idea that an effective representation for each token (whether in code or documentation) should encode the contextual information surrounding it. One of the traditional pre-training tasks employed by BERT-based models is Masked Language Modeling (MLM). In MLM, a subset of tokens is masked to simulate their absence from the pre-training input. During pre-training, these masked tokens are unveiled using classification models, with each token in the dataset serving as a candidate for classification. The MLM task leverages and enriches the embeddings of tokens surrounding the masked ones for this classification process. It’s worth noting that pre-training tasks like MLM are resource-intensive, demanding substantial computational resources, as the datasets for such tasks can encompass millions to billions of code snippets [43].

2.0.2 Fine-tuning Stage. The fine-tuning phase follows the pre-training phase, utilizing the pre-trained models produced earlier. In fine-tuning, the goal is to understand the relationship between the embeddings of NL queries and code candidates generated by the pre-trained models. After each training step, these representations are updated to better capture this correlation. Multiple loss functions are employed to assess each training step in the fine-tuning process. In the most recent BERT-based models like UniXcoder [10] and CoCoSoDa [32], contrastive learning loss is utilized. Notably, the fine-tuning process is considerably less computationally and resource-intensive compared to pre-training, a characteristic emphasized in the works of UniXcoder and CoCoSoDa [32].

1https://github.com/pdhung3012/Oracle4CS-RP

2https://github.com/microsoft/CodeBERT/

Table 1: Summary of Code Search Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Train</th>
<th>Valid</th>
<th>Test</th>
<th>Codebase</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdvTest</td>
<td>251820</td>
<td>9604</td>
<td>19210</td>
<td>19210</td>
</tr>
<tr>
<td>cosqa</td>
<td>19604</td>
<td>500</td>
<td>500</td>
<td>6267</td>
</tr>
<tr>
<td>Java</td>
<td>164923</td>
<td>5183</td>
<td>10955</td>
<td>40347</td>
</tr>
<tr>
<td>Python</td>
<td>251820</td>
<td>13914</td>
<td>14918</td>
<td>43827</td>
</tr>
<tr>
<td>Javascript</td>
<td>58025</td>
<td>3885</td>
<td>3291</td>
<td>13981</td>
</tr>
<tr>
<td>Go</td>
<td>167288</td>
<td>7325</td>
<td>8122</td>
<td>28120</td>
</tr>
<tr>
<td>PHP</td>
<td>241241</td>
<td>12982</td>
<td>14014</td>
<td>52660</td>
</tr>
<tr>
<td>Ruby</td>
<td>24927</td>
<td>1400</td>
<td>1261</td>
<td>4360</td>
</tr>
</tbody>
</table>

2.1 Baselines for Oracle4CS

In our study, we have chosen BERT-based models as the original baselines for comparison with our proposed approach. This choice was made because BERT-based models have a well-established reputation for their high accuracy in various SE tasks, including code search. Additionally, the source code repositories for these BERT-based models are publicly available. We also considered including CodeT5+ [43] as an alternative baseline for replication. However, as of the time of submission of this paper, the source code for fine-tuning CodeT5+ was still in development. We have divided the BERT-based models we used into two categories: original BERT-based models and optimized BERT-based models.

2.1.1 Original BERT-based Models. This category includes all BERT-based models developed by Daya Guo and Microsoft teams, and their source code can be found at the following repository². We have replicated the results of several models in this category, namely RoBERTa [20], CodeBERT [8], GraphCodeBERT [11], and UniXcoder [10] across 8 selected datasets, and these results are reported in our experiments.

2.1.2 Optimized BERT-based Models. These categories of BERT-based models aim to enhance the accuracy of the original BERT-based models through various approaches. This includes altering the fine-tuning loss function, as demonstrated in the work by Shi et al. [34], or both modifying the fine-tuning loss function and making adjustments to the original input data, as seen in their work on CoCoSoDa [32]. To the best of our knowledge, as of October 2023, CoCoSoDa stands out as the model that has achieved the highest accuracy in code search. CoCoSoDa accomplishes this by modifying both the pre-training and fine-tuning processes of the GraphCodeBERT model to achieve optimization. In our study, we have replicated the work of CoCoSoDa on our selected datasets to enable a direct comparison with our code search models.

2.2 Selected Metrics and Datasets for Evaluating Code Search Models

We have chosen Mean Reciprocal Rank (MRR) as the primary metric for our evaluation, as it is a widely used metric in existing code search pipelines, including UniXcoder [10]. Additionally, we calculate Top-K accuracy (with different values of K) in the Result Analysis section, which provides further insights into code search performance, as seen in many previous studies [29].
Following the approach of UniXcoder [10], we have utilized the same eight datasets provided by the CodeSearchNet benchmark [14]. We have also included two optimized versions of the Python dataset, namely AdvTest and cosqa datasets. Our dataset split, including training, validation, test, and codebase subsets, aligns with UniXcoder for consistency. The statistics on our selected datasets is shown in Table 1.

### 3 MOTIVATION EXAMPLE

In this section, we delve into the components that constitute our proposed code search model, Oracle4CS, by elucidating their input and output using the Motivation Example presented in Listing 1. This example was extracted from the test set of the Python dataset within the CodeSearchNet benchmark [14].

```
def propagateClkRst(obj):
    clk = obj.clk
    rst = obj.rst
    for u in obj._units:
        _tryConnect(clk, u, 'clk')
        _tryConnect(rst, u, 'rst_n')
```

**Listing 1: Motivation Example: Correct Candidate in Python for Query "Propagate clk clock and reset rst signal to all subcomponents"**

### 3.1 Natural Language Query and Candidate

In the realm of state-of-the-art (SOTA) code search models, the user input typically comes in the form of a natural language (NL) query. In the example at hand, the input query comprises two distinct phrases. The first phrase pertains to an instruction involving the propagation of the clk clock. The second phrase addresses the subsequent action, which involves handling the object denoted as rst signal by resetting it, including all of its subcomponents. This query is succinctly composed, providing information about two variables, each represented as a separate token.

Another crucial input for conducting code search is the code snippet itself, which represents a practical implementation capable of fulfilling the requirements outlined in the NL query. In Listing 1, we observe a correct implementation aligned with the given NL query. Within the code search process, the task involves locating the correct code snippet, often referred to as the candidate, from a pool of existing candidates. This pool is typically referred to as the Codebase, which serves as the input for the code search operation. A successful code search outcome entails placing the correct candidate at the very top of the list, surpassing all other candidates contained within the Codebase set.

### 3.2 Abstract Syntax Tree of Candidate

The Abstract Syntax Tree (AST) is a widely recognized code representation utilized in numerous research endeavors [10, 11]. In our example, the AST for the correct candidate, as illustrated in Figure 1, is presented as a tree structure. Within this tree, the terminal nodes contain information related to code tokens, while the non-terminal nodes encapsulate details about the grammatical relationships among these code tokens. The flattened compilation of all terminal nodes essentially constitutes the source code featured in Listing 1.

### 3.3 ASTSum Representation of Candidate

We introduce our concept of AST summarization, termed “ASTSum Representation,” which is composed of the immediate child nodes of the Method Body Root Node. In Figure 1, the Method Body Root Node is depicted as the orange node labeled Block. Consequently, the ASTSum Representation consists of a list of three tokens: "expression_statement," "expression_statement," and "for_statement." These tokens form a highly concise representation of non-terminal nodes, offering an efficient alternative to representing code with numerous non-terminal nodes as seen in ASTTrans Representation.

### 3.4 Augmented Query in Oracle4CS

Our objective is to employ the ASTSum Representation for each candidate as valuable information to enhance the existing natural language (NL) query. The resulting query representation, which we term the “augmented query,” incorporates two primary sources of information. Firstly, the initial section of the augmented query mirrors the original NL query, encompassing the NL description. Secondly, we append the ASTSum Representation to the original query, demarcating the original segment from the ASTSum tokens using a separator. It’s important to note that, as the ASTSum Representation of the correct candidate remains concealed from users who provide the query, we must develop models for query-to-ASTSum Representation generation.

### 4 APPROACH

The overview of our proposed code search process with Oracle4CS is illustrated in Figure 2. The primary concept behind this process is to utilize Statistical Machine Translation as a predictor, which forecasts a list of high-level abstract AST nodes known as AST-Sum Representation. These additional tokens are appended to the original query prior to the fine-tuning and code search phases. We provide detailed descriptions of each module as follows.
### 4.1 Query-to-ASTSum: Training

Oracle4CS incorporates an MT engine for predicting the ASTSum Representation of the anticipated implementation based on the NL query. We employ Statistical Machine Translation along with our proposed program extraction module to carry out this task. The input for this process comprises a parallel corpus consisting of text in the form of code documentation and code snippets at the functional level. Each data point in this corpus consists of a pair of corresponding documentation and implementation for the same programming task.

**ASTParser.** Within this module, we extract the Abstract Syntax Tree (AST) from the input, which represents the source code. AST extraction is a fundamental task in various software engineering applications, including type inference [30], code search [10], and parallelism detection [5]. The AST provides valuable information about the syntactic structure of the source code and the relationships between code tokens. We utilize Treesitter [1], a widely recognized tool for AST extraction. Treesitter boasts error recovery capabilities and supports multiple programming languages, making it an ideal choice for our ASTParser implementation. The output of this step is the AST of each source code snippet, returned in JSON format.

**ASTSummarization Algorithm.** Our algorithm for ASTSum Representation, known as ASTSummarization, is presented in Algorithm 1. Initially, given the JSON dictionary representing the AST of a code snippet as input, the algorithm seeks to identify the node that serves as the root node of the method’s body. To accomplish this, we have implemented a recursive function called `findBodyNode`, which performs an in-order traversal starting from the AST’s root node and navigating through its children nodes. The function halts when it locates the root node of the method’s body, which is recognized as the first “Block” node. It’s worth noting that the label for the method body root node may vary across different programming languages. For instance, in JavaScript, it is labeled as “statement_block.” Given that our evaluation covers six programming languages, we determine the label for the method body root node by inspecting the grammatical rules provided by treesitter for each language. Once the method body root node is identified, the subsequent step involves gathering the types (labels) of all its children nodes and storing them in the `listSeqs` variable. In contrast to ASTTrans [29], we exclusively select the nodes that are direct children of the method body root node as the target for SMT models.

#### Algorithm 1: AST Summarization

```plaintext
1: function ASTSummarization(objAST) = objAST; the AST of a given code snippet in JSON format
2: bodyNode = findBodyNode(objAST) // Inorder-traversal to find body’s root node
3: listSeqs = []
4: for node ∈ bodyNode.children do
5:     listSeqs.append(node.type) // Add type of each child node to output
6: end for
7: return listSeqs
8: end function
```

**Sequence-to-sequence Learning.** The training process for sequence-to-sequence learning involves treating the NL query as the source language and the target language as the expected ASTSum Representation for the correct implementation of that query. In a manner similar to pseudocode-to-code training using Machine Translation [16], we aim to learn the mapping between sequences of words in one language (NL query) to sequences of code tokens, which corresponds to our defined representation (ASTSum). To implement this module, we utilize the Phrasal toolkit [9]. We configure our sequence-to-sequence learning process using the default settings of Phrasal with the phrase length (i.e. number of consecutive tokens for processing) as 7. This training process allows the model to learn the translation mapping between natural language queries and the corresponding ASTSum Representations, enabling the model to generate ASTSum Representations for NL queries during the inference stage.
A high probability indicates a likelihood that phrase $t$ in the source language being correctly translated to the probability of phrases in the target language based on their frequency in a corpus. Phrases that appear more frequently are calculated from the translation model and the language model. This model calculates the probability of a phrase in the target language appearing with a phrase in the target language. A high probability indicates a likelihood that phrase $t$ is a correct translation of phrase $s$.

Language Model: The importance of each phrase $t$ in the target language is determined by a language model. This model calculates the probability of phrases in the target language based on their frequency in a corpus. Phrases that appear more frequently are considered better candidates for translation.

Decoding Algorithm: The combination of probabilities calculated from the translation model and the language model is generated using a decoding algorithm. This algorithm approximates the probability of a phrase $t$ being the correct translated candidate for input phrase $s$ by finding the best multiplication between the probabilities from the translation model and the language model.

In summary, these models work together to estimate the likelihood of a phrase in the source language being correctly translated to a phrase in the target language based on both translation probabilities and language model probabilities.

4.3 Integrating Oracle4CS to Original Code Search Models

For code search, BERT-based models accomplish this task through three modules: pre-training, fine-tuning, and query-code comparison. We have previously discussed the pre-training and fine-tuning processes in the Background section. The query-code comparison module involves extracting the embeddings of the query and each code snippet candidate, and then comparing them using similarity metrics such as cosine similarity to identify the code snippet with the highest similarity score to the query’s embedding.

We made modifications to the existing BERT-based code search models in both the fine-tuning and comparison stages. In the fine-tuning stage, we used the training subsets of our selected dataset. To enhance the query representation, we included information about the predicted ASTSum representation as an additional sentence alongside the original representation. In the comparison stage, we also augmented the basic query with predicted code tokens. It’s worth noting that our approach maintains consistency in the representation of the source code snippets with the code representation used in the BERT-based models we built upon, namely UniXcoder [10] and CoCoSoDa [32].

Configurations. We conducted all code search experiments, which involved replicating baselines and building Oracle4CS models, on a computer equipped with a Core i9 processor and an Nvidia RTX 3090 graphics card with 24GB of memory. With this computational resource, we set the training batch size to 48 for all experiments. Regarding other configuration metrics, we followed the same metrics as UniXcoder when building Oracle4CS for UniXcoder ($\text{Oracle4CS}_{\text{UniXcoder}}$) and the same metrics as CoCoSoDa when building Oracle4CS for CoCoSoDa ($\text{Oracle4CS}_{\text{CoCoSoDa}}$).

In the next two sections, we describe about the current accuracy of Oracle4CS over BERT-based model and about the potential of improving code search process by enhancing the role of our proposed query-to-ASTSum translation model.

5 EVALUATION ON THE ACCURACY OF ORACLE4CS

In this part of evaluation, we attempt to answer the following research questions:

1. Research Question (RQ) 1. How well can Oracle4CS support for optimizing code search for original BERT-based models?
2. RQ2. How well can Oracle4CS support for optimizing code search for optimized BERT-based models?
5.1 RQ1. Accuracy of Oracle4CS fine-tuned on UniXcoder.

Results: In Table 2, we present the accuracy of Oracle4CS\text{UniXcoder} (OCS1) in code search compared to the original BERT-based models. To provide context, our replicated results for the baseline original BERT-based models show that UniXcoder stands out as the most powerful tool for code search among our selected baselines, achieving an average Mean Reciprocal Rank (MRR) of 69.87%. Among the other models, Roberta [20] attained the lowest accuracy at 56.08%, while CodeBERT [8] and GraphCodeBERT [11] achieved MRR scores exceeding 63% and 69.8%, respectively. In general, our replicated results align with the reported accuracy levels in these baseline papers.

Comparison with ASTTrans: Our proposed model, Oracle4CS, outperforms the original BERT-based models, achieving the highest accuracy with a 70.48% Mean Reciprocal Rank (MRR). In comparison, ASTTrans only saw a marginal 0.06% MRR improvement. Additionally, while ASTTrans did not perform experiments on the other seven datasets, we conducted experiments and obtained results. Oracle4CS, denoted as OCS1, showed notable improvements, particularly on the Python dataset where it achieved a 1.18% MRR improvement over the UniXcoder model. It performed well on datasets related to Java, Python, Ruby. However, the improvements were less significant, at less than 0.5% MRR, for the Go and AdvTest datasets. These results are in line with expectations since UniXcoder already achieved a very high accuracy of 91.52% on Go language, while the AdvTest dataset is known to be particularly challenging for code search, as indicated in previous work. In summary, Oracle4CS, when built upon UniXcoder, demonstrated a significant up to 0.61% MRR improvement over the state-of-the-art model UniXcoder, highlighting its substantial contribution to enhancing code search compared to ASTTrans.


In RQ2, we conducted fine-tuning experiments with the CoCoSoDa pre-trained models, using our alternative representation of the input query. Additionally, we replicated the fine-tuning stage of CoCoSoDa to provide a baseline for comparison. With CoCoSoDa, our model achieved an average MRR of 72.02% across the eight datasets. Notably, CoCoSoDa achieved the highest accuracy on the Go dataset, with an MRR of 92.08%, while obtaining the lowest MRR score of 41.08% on the challenging AdvTest dataset. These results align with the performance trends observed in the original BERT-based models.

In our experiments with Oracle4CS\text{CoCoSoDa} (OCS2), we observed that it improved the accuracy of CoCoSoDa by more than 0.6%. With the exception of the JavaScript dataset, OCS2 demonstrated a positive impact on CoCoSoDa, achieving a maximum MRR improvement of 2.28% on some datasets. Notably, OCS2 achieved significant MRR improvements on the AdvTest and cosqa datasets. It’s worth noting that CoCoSoDa was pre-trained on six standard datasets from the CodeSearchNet benchmark. While it can enhance code search accuracy on these six datasets, it may face challenges when dealing with unseen datasets not encountered during pre-training. Overall, Oracle4CS\text{CoCoSoDa} showcased its effectiveness in boosting the performance of CoCoSoDa, particularly on datasets that were not part of the pre-training tasks.

6 EVALUATION ON THE POTENTIAL ON QUERY-TO-ASTSUM TRANSLATION IN IMPROVING CODE SEARCH

In this evaluation, we aim to test the effectiveness of query-to-ASTSum translation if the following criteria are satisfied:

6.1 RQ3. How well can an optimal query-to-ASTSum translation improve for code search over BERT-based models?

Scenario: In this RQ, we assume we have an optimal SMT translation model that can correctly translate the output as ASTSum Representation for any input query. From this scenario, we can observe how much benefit for code search we can have if we can improve the accuracy of current SMT models.

Setup: We define an optimal query-to-ASTSum translation model as the model that always gives the code search model the expected ASTSum Representation. Thus, from the Oracle4CS code search model shown in Figure 2, we replaced the predicted ASTSum with the expected ASTSum for each augmented query. Next, we ran the fine-tuning stage and the comparison stages of code search with this assumption. We evaluated both two baselines, UniXcoder and CoCoSoDa.

Results: We show the results of RQ3 on Table 2. We can see that, on average, the UniXcoder model using the expected ASTSum Representation for augmentation got close to 17% of MRR improvement over the default configuration. The improvement gained at most for the PHP dataset, which increased over 21% of accuracy in code search. This scenario achieved the lowest improvement on the Go dataset, which is reasonable since the accuracy of SOTA models for code search is already high on the cosqa dataset. Overall, while the current Oracle4CS model got 0.61% of MRR improvement (on average) for UniXcoder compared to over 17% of MRR improvement with a perfect SMT model, there is room for improvement of the quality of SMT when applied in SE problems.

7 RESULT ANALYSIS

In this section, we go into detail about our analysis of different aspects of the accuracy of Oracle4CS besides the MRR scores. They are the top-K accuracy, the case-to-case comparison, and the quality of the query-to-ASTSum Representation translation measured by multiple translation metrics. Since RQ3 and RQ4 are the assumptions of the best scenarios of Oracle4CS, we didn’t analyze the results from these RQs since they are different from the current output of Oracle4CS. We analyze the results from RQ1 and RQ2 for this analysis.

7.1 Top-K accuracy

We calculate the top K accuracy of SOTA models and our proposed models for comparison. We measure the percentage of queries that their correct implementation is at the top-K of the suggested list
Table 3: Results as Top-K Accuracy of Oracle4CS over SOTA Approaches

<table>
<thead>
<tr>
<th>RQs</th>
<th>Model</th>
<th>R@k</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>Improve</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RQ1</td>
<td>UniXcoder</td>
<td>60.63%</td>
<td>71.50%</td>
<td>76.19%</td>
<td>79.28%</td>
<td>81.39%</td>
<td>0.76%</td>
<td>0.59%</td>
<td>0.66%</td>
<td>0.35%</td>
<td>0.36%</td>
</tr>
<tr>
<td></td>
<td>OCS1</td>
<td>61.39%</td>
<td>72.09%</td>
<td>76.86%</td>
<td>79.63%</td>
<td>81.76%</td>
<td>0.67%</td>
<td>0.69%</td>
<td>0.28%</td>
<td>0.24%</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td>Improve</td>
<td>62.15%</td>
<td>73.76%</td>
<td>79.22%</td>
<td>82.34%</td>
<td>84.54%</td>
<td>0.96%</td>
<td>0.69%</td>
<td>0.28%</td>
<td>0.24%</td>
<td>0.03%</td>
</tr>
<tr>
<td>RQ2</td>
<td>CoCoSoDa</td>
<td>63.11%</td>
<td>74.45%</td>
<td>79.50%</td>
<td>82.58%</td>
<td>84.57%</td>
<td>0.96%</td>
<td>0.69%</td>
<td>0.28%</td>
<td>0.24%</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td>OCS2</td>
<td>63.11%</td>
<td>74.45%</td>
<td>79.50%</td>
<td>82.58%</td>
<td>84.57%</td>
<td>0.96%</td>
<td>0.69%</td>
<td>0.28%</td>
<td>0.24%</td>
<td>0.03%</td>
</tr>
<tr>
<td></td>
<td>Improve</td>
<td>63.11%</td>
<td>74.45%</td>
<td>79.50%</td>
<td>82.58%</td>
<td>84.57%</td>
<td>0.96%</td>
<td>0.69%</td>
<td>0.28%</td>
<td>0.24%</td>
<td>0.03%</td>
</tr>
</tbody>
</table>

Table 4: Result Analysis: Case-by-case comparison for RQ1 and RQ2

<table>
<thead>
<tr>
<th>RQs</th>
<th>RQ1</th>
<th>RQ2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>15.67%</td>
<td>74.23%</td>
<td>10.10%</td>
<td>8.96%</td>
<td>81.68%</td>
<td>9.36%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>python</td>
<td>16.86%</td>
<td>73.68%</td>
<td>9.46%</td>
<td>6.21%</td>
<td>87.14%</td>
<td>6.65%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javascript</td>
<td>13.67%</td>
<td>75.84%</td>
<td>10.48%</td>
<td>4.83%</td>
<td>85.93%</td>
<td>9.24%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>go</td>
<td>2.67%</td>
<td>94.96%</td>
<td>2.36%</td>
<td>3.64%</td>
<td>93.67%</td>
<td>2.68%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>php</td>
<td>19.29%</td>
<td>67.73%</td>
<td>12.98%</td>
<td>8.40%</td>
<td>83.49%</td>
<td>8.11%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ruby</td>
<td>13.80%</td>
<td>73.75%</td>
<td>12.45%</td>
<td>2.85%</td>
<td>93.10%</td>
<td>4.04%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AdvTest</td>
<td>23.11%</td>
<td>56.03%</td>
<td>20.86%</td>
<td>36.13%</td>
<td>38.88%</td>
<td>24.99%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cosqa</td>
<td>6.78%</td>
<td>88.35%</td>
<td>4.87%</td>
<td>18.64%</td>
<td>65.25%</td>
<td>16.10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>13.98%</td>
<td>75.57%</td>
<td>10.45%</td>
<td>11.21%</td>
<td>78.64%</td>
<td>10.15%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Results: UniXcoder got the top-1 accuracy as 60%. The percentage of queries that have their corresponding implementation within the top 5 is over 81%. With OCS1, the top-1 accuracy of our model improved by 0.76% over the existing work UniXcoder. With $K = 5$, Oracle4CS improved 0.36% of the top-5 accuracy over queries on 8 selected datasets. For RQ2, CoCoSoDa has significantly higher accuracy compared to UniXcoder, our Oracle4CS model, which was built on CoCoSoDa and can improve 0.96% of top-1 accuracy. In summary, Oracle can improve the top-K accuracy of the baselines approach by up to 0.96% in top-1 accuracy, which is promising but also has room for improvement in future works.

7.2 Case-by-case Comparison

We perform the same study with ASTTrans [29] to analyze when our proposed model performed better than the SOTA approaches and when it didn’t. We call a case of code search a phase in which the user gives a query to multiple code search systems to have multiple lists of candidates. The Improve (Imp.) cases are the cases when Oracle4CS returned better results than the SOTA approach for a query as input, while the Decrease (Dec.) cases are the cases in the SOTA model returned better results. We perform the case-by-case comparison for both RQ1 and RQ2.

Results: For RQ1, Oracle4CS made effects (including both positive and negative effects) on 25% of the queries. In these queries, there are about 14% of all queries were improved by Oracle4CS, while around 11% of queries have better results with the original BERT-based model UniXcoder. For RQ2, Oracle4CS made an impact on over 21% of their queries. Different from RQ1, the cases in Oracle4CS improved the accuracy of CoCoSoDa model are just around 1.3% more than the percentage of the cases when it downgraded the performance of its SOTA approach. In terms of programming languages, Oracle4CS shows their effect most significantly in AdvTest dataset. Oracle4CS had little effect on the Go and cosqa datasets for RQ1 and the Ruby dataset for RQ2. We conclude that although the number of Improve cases is higher than the number of Decrease cases, future works will identify Decrease cases can be a good direction to improve the performance of our proposed oracle.

7.3 Quality of Query-to-ASTSum Translation

Table 5: Result Analysis on the accuracy of query-to-ASTSum Translation

<table>
<thead>
<tr>
<th>Metric</th>
<th>java</th>
<th>python</th>
<th>JS</th>
<th>go</th>
<th>php</th>
<th>ruby</th>
<th>AdvTest</th>
<th>cosqa</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>0.906</td>
<td>0.006</td>
<td>0.003</td>
<td>0.005</td>
<td>0.033</td>
<td>0.010</td>
<td>0.010</td>
<td>0.020</td>
<td>0.017</td>
</tr>
<tr>
<td>BLEU</td>
<td>0.109</td>
<td>0.357</td>
<td>0.097</td>
<td>0.110</td>
<td>0.211</td>
<td>0.261</td>
<td>0.237</td>
<td>0.340</td>
<td>0.218</td>
</tr>
<tr>
<td>CodeBLEU</td>
<td>0.292</td>
<td>0.345</td>
<td>0.145</td>
<td>0.221</td>
<td>0.322</td>
<td>0.351</td>
<td>0.352</td>
<td>0.374</td>
<td>0.303</td>
</tr>
<tr>
<td>CrystalBLEU</td>
<td>0.109</td>
<td>0.342</td>
<td>0.096</td>
<td>0.109</td>
<td>0.190</td>
<td>0.185</td>
<td>0.185</td>
<td>0.218</td>
<td>0.303</td>
</tr>
<tr>
<td>ROUGE-1</td>
<td>0.501</td>
<td>0.704</td>
<td>0.490</td>
<td>0.510</td>
<td>0.650</td>
<td>0.607</td>
<td>0.605</td>
<td>0.733</td>
<td>0.600</td>
</tr>
<tr>
<td>ROUGE-2</td>
<td>0.128</td>
<td>0.352</td>
<td>0.117</td>
<td>0.126</td>
<td>0.262</td>
<td>0.319</td>
<td>0.317</td>
<td>0.399</td>
<td>0.253</td>
</tr>
<tr>
<td>ROUGE-L</td>
<td>0.464</td>
<td>0.697</td>
<td>0.461</td>
<td>0.497</td>
<td>0.631</td>
<td>0.598</td>
<td>0.594</td>
<td>0.729</td>
<td>0.568</td>
</tr>
<tr>
<td>Meteor</td>
<td>0.522</td>
<td>0.533</td>
<td>0.308</td>
<td>0.337</td>
<td>0.434</td>
<td>0.475</td>
<td>0.476</td>
<td>0.584</td>
<td>0.435</td>
</tr>
<tr>
<td>CodeBERTScore</td>
<td>0.875</td>
<td>0.937</td>
<td>0.980</td>
<td>0.903</td>
<td>0.928</td>
<td>0.925</td>
<td>0.928</td>
<td>0.933</td>
<td>0.917</td>
</tr>
</tbody>
</table>

For this analysis, we use multiple active metrics that are used to evaluate the quality of Machine Translation. We derive 9 metrics for this measurement. There are two categories of our selected metrics: NLP-based metrics and Programming Language (PL)-based metrics. NLP-based metrics are Exact Match, BLEU score [28], ROUGE scores [19] (we used 3 metrics ROUGE-1, ROUGE-2, and ROUGE-L in the set of this scoring approach) and Meteor. PL-based metrics are CodeBLEU [31], CrystalBLEU [7] and CodeBERTScore [48]. Compared to NLP-based metrics, PL-based metrics included the optimization techniques that highlighted the characteristic of comparison between code tokens considering syntactic and semantic similarity [48].

Results: We show that traditional NLP-based scores returned low scores for several datasets, including the Java dataset and the Javascript dataset. On average, the BLEU score of query-to-ASTSum translation is over 22%. However, with other PL-based scores, such as CodeBLEU and CrystalBLEU, the score is much higher. The CodeBERTScore metric returned the best accuracy for this evaluation. Given the fact that the CodeBERTScore metric was calculated based
on the vector representation of code tokens, we show that CodeBERTScore can be a better evaluator when the translated output is used for SE tasks that involve code representation as vectors.

8 RELATED WORK

**Code Search.** Wang et al. [41] prove that there was a gap between pre-trained models and fine-tuned models due to the mismatch between model parameters. Chai et al. [3] tackled the problem of cross-domain code search. They introduced CDCS, a tool that adapted transfer learning to optimize the pre-trained LLMs for well-known PLs to be usable for Domain Specific Language (DSL). Ma et al. [22] explored another code representation, Intermediate Representation (IR), to construct the embedding for SE tasks, including code search. Mao et al. [23] introduced SSQR, a tool for reformulating queries for code search. This work extended the pre-training tasks provided by the T5 model [47] to form a new pre-training task for enhancing query formulation. CCT-Code [36] proposed a training approach over a newly curated dataset named XCD, which enhances the cross-lingual and multilingual abilities of language models for code search. The most recent work on code search besides CoCoSoDa is HyCoQA, developed by Tang et al. [38]. They relied on BERT embedding for code and text representation and formulated the code search problem as the scoring problem given a tuple of query, correct candidate, and incorrect candidate as the input. However, we didn’t include this work as a baseline for our work since their replication packages had not yet been available.

**Code Generation.** Sun et al. [37] introduced TreeGen, a tool that integrated AST’s grammar structure for domain-specific language to Python translation. Kim et al. [15] incorporated the syntactic structure of code into existing Transformer architecture to enhance the code completion tasks. Nam et al. [24] focused their work on code generation at API levels. Wang et al. [42] integrated compiler feedbacks, which are frequently ignored, to enhance the code generation process in its ability of deriving compilable code. Tipimenti et al. [39] modified a traditional transformer with a new encoder-decoder Transformer model that has the capability of recognizing syntax and data flow of code snippets. Chakraborty et al. [4] proposed NatGen, a new pre-training task that generated semantically equivalent code from the original code. CodeRL, proposed by Le et al. [17], leveraged reinforcement learning to construct a program synthesis framework that can predict the functional correctness of generated code. Dong et al. [6] proposed CodeScore, which also predicted the correctness of generated code using LLMs. Li et al. [18] reformulated the code generation problem as a question-answering dialog system. Siddiq et al. [35] introduced FRANC, a static filter to avoid uncompilable code for code generation with LLM models. Weyssow et al. [45] proposed an approach for parameter-efficient fine-tuning for LLM models. In summary, recent code generation works have been directed toward solutions supporting unsupervised learning and efficient learning on LLMs.

9 THREATS TO VALIDITY

There are two threats to the validity of our work. First, due to resource’s restriction using one GPU with 24GB RAM, we constructed CoCoSoDa’s model with about 1% lower MRR than the reported result in CoCoSoDa’s paper [32]. To mitigate this threat, we contacted the authors of CoCoSoDa and got the confirmation that our replication process on single GPU is correct. Second, our proposed Oracle4CS might not be available for other programming languages. Since our selected AST generation tool treesitter [1] supported multiple popular programming languages, our work can be easily extendable for other languages to extract ASTSum Representation at the functional level.

10 CONCLUSION

In this work, we propose Oracle4CS, a novel approach that leverages our defined ASTSum Representation for source code and Statistical Machine Translation, a classical MT technique, to improve the code search process at the fine-tuning stage. Experiments show that Oracle4CS can not only improve the performance of the original BERT-based model but also integrate successfully with other optimized BERT-based models such as CoCoSoDa. We prove that classical models like SMT can still be helpful in code search if they learned simple but meaningful code representation such as ASTSum. In future work, we attempt to evaluate our work with newer Machine Translation techniques and optimize our code representation for other SE tasks.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under Grant number 2211982. We thank the PolicyAI and WindAI teams of the Pacific Northwest National Laboratory (PNNL) since the first author was a PhD intern at PNNL when he submitted this paper to the EASE conference. We thank Dr. Sameera Howravalithana and Dr. Sai Mumikoti for their efficient mentorship at PNNL, to help the first author to be able to complete his internship work on time thus he could spend 16 hours per day (8 for internship work, 8 for this paper’s preparation) to work when the conference’s deadline was approaching in 2 weeks. We would also like to thank the ResearchIT team at Iowa State University for their constant support.

REFERENCES

[8] Zhangyin Feng, Daya Guo, Dayu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: