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

EASE 2024, June 18–21, 2024, Salerno, Italy

ACM ISBN 979-8-4007-1701-7/24/06
https://doi.org/10.1145/3661167.3661233

ACM Reference Format:
Hung Phan and Ali Jannesari. 2024. Leveraging Statistical Machine Translation for Code Search. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661233

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

consists of sets of software libraries required for code snippets, the authors applied and optimized statistical translation models to learn from sequences of code tokens without type annotations and convert them into sequences of code tokens with type annotations. Consequently, StatType's final evaluation metrics are based on the overlap between the suggested annotated types and the expected annotated types in the source code. This differs from the original translation evaluation metrics such as CodeBLEU [31]. Similarly, NMT has been employed to address type inference problems [12], using evaluation metrics akin to those in the StatType tool. StatGen [26], proposed by Nguyen et al., is another tool that applies bidirectional MT for documentation-to-implementation translation and vice versa. StatGen serves to check documentation-implementation consistency, which can be framed as a binary classification problem. To accomplish this task, StatGen compares the expected generated code with the predicted generated code from software documentation 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 Translation 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 candidate. The rationale behind this approach lies in the observation that code search datasets often feature code snippets presented as multiple lines of code (LOC) alongside very abstract natural language queries. This abstractness posed challenges for MT models in accurately 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 simplified 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 compete 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 theAST of source code. This representation efficiently captures comprehensive AST information using a limited number of tokens. Secondly, we employ 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 Oracle4CS, an innovative code search approach that builds upon both the original BERT-based model, UniXcoder [10], and the latest optimized BERT-based model, CoCoSoDa [32]. In our work, we redefine the role of an oracle as a "satellite" for the code search process, focusing 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 engines built on top of UniXcoder (called $Oracle4CS_{UniXcoder}$) and CoCoSoDa (called $Oracle4CS_{CoCoSoDa}$). These engines significantly enhance code search accuracy, achieving improvements 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, assuming the availability of perfect Statistical Machine Translation and combined Oracle4CS and State-of-the-Art (SOTA) models.

The remaining sections of this paper follow a structured progression. 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 Example section illustrates our point with a practical code snippet, its corresponding query, and the innovative ASTSum Representation. Section 4 outlines the intricacies of our proposed approach, Oracle4CS. In Sections 5 and 6, we delve into our experimental assessments of Oracle4CS, including an evaluation of its code search accuracy and an exploration of the potential applications of ASTSum 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 stage 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].

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

Table 1: Summary of Code Search Datasets

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

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 repository[2]. 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 accomplates 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].

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

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')
        _tryConnect(rst, u, 'rst')
```

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

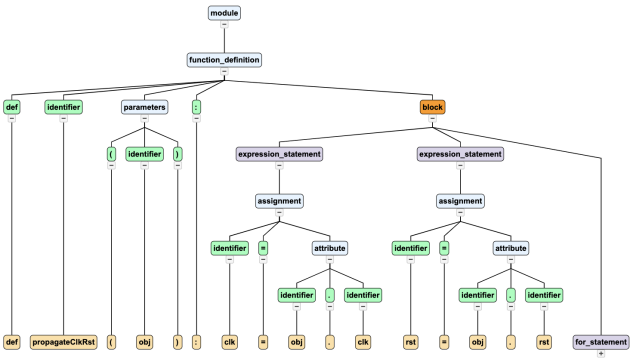

Figure 1: AST of Correct Code Snippet (Candidate) of the Motivation Example

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 ASTSum 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.

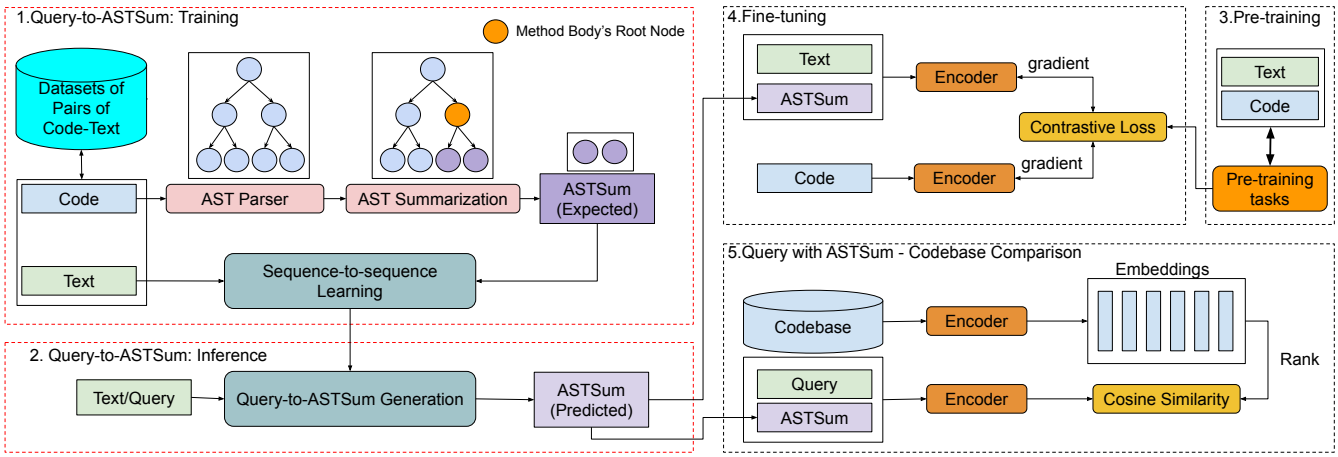

Figure 2: The Overview of Code Search Process with Oracle4CS

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.

Algorithm 1 AST Summarization

```
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
```

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.

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.

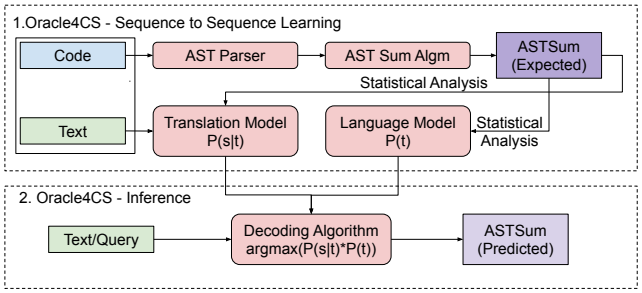

Figure 3: Overview of processes of Statistical Machine Translation in Oracle4CS

4.2 Query-to-ASTSum: Inference

The underlying technique of Statistical Machine Translation (SMT) in our work involves building models that learn the mapping between phrases of natural language (NL) queries and phrases of our constructed AST Summarization derived from NL queries' corresponding implementations. This mapping is accomplished through attention learning, which enables the model to concentrate on various parts of the input when generating the output. Figure 3 illustrates the process of query-to-ASTSum inference using Statistical Machine Translation (SMT). This process involves the use of the following models:

Translation Model: Given a phrase in the NL query as a sentence in the source language (referred to as s) and a phrase in the ASTSum representation as the target language (referred to as t), a translation model calculates the probability of a phrase in the source 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 ($Oracle4CS_{UniXcoder}$) and the same metrics as CoCoSoDa when building Oracle4CS for CoCoSoDa ($Oracle4CS_{CoCoSoDa}$).

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.

Table 2: Results by Mean Reciprocal Rank (MRR) of RQ1, RQ2, RQ3 and RQ4. Code Search Models: Roberta (R); CodeBERT (CB); GraphCodeBERT (GCB); UniXcoder (UniX.); $Oracle4CS_{UniXcoder}$ (OCS1); CoCoSoDa (CoCo.); $Oracle4CS_{CoCoSoDa}$ (OCS2)

Dataset	CodeSearchNet								
Model	**Java**	**Py.**	**JS**	**Go**	**PHP**	**Ruby**	**Adv.**	**cos.**	**Avg**
RQ1									
R	59.90%	58.70%	51.70%	85.00%	56.00%	58.70%	18.30%	60.30%	56.08%
CB	67.60%	67.20%	62.00%	88.20%	62.80%	67.90%	27.20%	65.70%	63.58%
GCB	68.70%	68.57%	65.05%	89.57%	64.75%	71.01%	38.05%	70.10%	66.98%
UniX.	72.28%	71.43%	68.87%	91.35%	66.87%	73.74%	42.17%	72.27%	69.87%
OCS1	**73.22%**	**72.60%**	**69.54%**	**91.52%**	**67.83%**	**74.65%**	**42.31%**	**72.20%**	**70.48%**
Improve	**0.94%**	**1.18%**	**0.67%**	**0.17%**	**0.96%**	**0.91%**	**0.14%**	**-0.08%**	**0.61%**
RQ2									
CoCo.	75.14%	74.74%	75.32%	92.08%	69.26%	80.94%	41.08%	67.61%	72.02%
OCS2	**75.49%**	**74.82%**	**75.15%**	**92.42%**	**69.52%**	**81.12%**	**43.14%**	**69.60%**	72.66%
Improve	**0.35%**	**0.08%**	**-0.17%**	**0.34%**	**0.27%**	**0.17%**	**2.06%**	**1.99%**	**0.64%**
RQ3 (UniX.)	**89.19%**	**90.93%**	**88.70%**	**96.23%**	**88.02%**	**87.44%**	**68.32%**	**83.01%**	**86.48%**

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 $Oracle4CS_{UniXcoder}$ (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.

5.2 RQ2. Accuracy of Oracle4CS fine-tuned on CoCoSoDa.

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 $Oracle4CS_{CoCoSoDa}$ (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, $Oracle4CS_{CoCoSoDa}$ 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 Representation 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 assumed representation. 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

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

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

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

of candidates, with the value of K from 1 to 5. We measure these metrics for four configurations, including two SOTA approaches and two models of Oracle4CS, which were built upon UniXcoder and CoCoSoDa.

Results: UniXcoder got the top-1 accuracy as 60%. The percentage of queries that have their corresponding implementation within the top 5 of the list of candidates 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 top-5 accuracy over queries on 8 selected datasets. For RQ2, the improvement in terms of top-K accuracy is consistent with RQ1. Despite that CoCoSoDa has significantly higher accuracy compared to UniXcoder, our Oracle4CS model, which was built on CoCoSoDa 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 on automatically identifying 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

Dataset	**CodeSearchNet**								
Metric	**java**	**python**	**JS**	**go**	**php**	**ruby**	**AdvTest**	**cosqa**	**Avg**
EM	0.006	0.006	0.003	0.003	0.033	0.010	0.010	0.066	0.017
BLEU	0.109	0.357	0.097	0.110	0.211	0.261	0.257	0.340	0.218
CodeBLEU	0.292	0.354	0.145	0.221	0.332	0.351	0.352	0.374	0.303
CrystalBLEU	0.104	0.342	0.096	0.104	0.190	0.185	0.185	0.272	0.185
ROUGE-1	0.501	0.704	0.490	0.510	0.650	0.607	0.605	0.733	0.600
ROUGE-2	0.128	0.352	0.117	0.126	0.262	0.319	0.317	0.399	0.253
ROUGE-L	0.484	0.697	0.461	0.497	0.631	0.598	0.594	0.729	0.586
Meteor	0.322	0.553	0.300	0.337	0.434	0.475	0.478	0.584	0.435
CodeBERTScore	0.873	0.937	0.900	0.903	0.928	0.929	0.928	0.953	0.919

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 evaluation 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 Horawalavithana and Dr. Sai Munikoti 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[3] at Iowa State University for their constant support.

REFERENCES

[1] [n. d.]. Article on treesitter. https://tinyurl.com/y2a86znt. Accessed: 2022-6-20.
[2] [n. d.]. Articles about Machine Translation. https://tinyurl.com/5n7d6wrd. Accessed: 2023-5-1.
[3] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-Domain Deep Code Search with Meta Learning. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 487–498. https://doi.org/10.1145/3510003.3510125
[4] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "Naturalizing" source code. arXiv:2206.07585 [cs.PL]
[5] Le Chen, Quazi Ishtiaque Mahmud, Hung Phan, Nesreen K. Ahmed, and Ali Jannesari. 2023. Learning to Parallelize with OpenMP by Augmented Heterogeneous AST Representation. arXiv:2305.05779 [cs.LG]
[6] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2023. CodeScore: Evaluating Code Generation by Learning Code Execution. arXiv:2301.09043 [cs.SE]
[7] Aryaz Eghbali and Michael Pradel. 2023. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 28, 12 pages. https://doi.org/10.1145/3551349.3556903
[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT:

[3] https://researchit.las.iastate.edu/

A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[9] Spence Green, Daniel Cer, and Christopher D Manning. 2014. Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of the ninth workshop on statistical machine translation*. 114–121.

[10] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. https://doi.org/10.48550/ARXIV.2203.03850

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* abs/2009.08366 (2020). arXiv:2009.08366 https://arxiv.org/abs/2009.08366

[12] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[13] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 2269–2275.

[14] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. https://doi.org/10.48550/ARXIV.1909.09436

[15] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. arXiv:2003.13848 [cs.SE]

[16] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based Pseudocode to Code. https://doi.org/10.48550/ARXIV.1906.04908

[17] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG]

[18] Haau-Sing Li, Mohsen Mesgar, André F. T. Martins, and Iryna Gurevych. 2023. Python Code Generation by Asking Clarification Questions. arXiv:2212.09885 [cs.CL]

[19] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81. https://aclanthology.org/W04-1013

[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[21] Adam Lopez. 2008. Statistical machine translation. *ACM Computing Surveys (CSUR)* 40, 3 (2008), 1–49.

[22] Yingwei Ma, Yue Yu, Shanshan Li, Zhouyang Jia, Jun Ma, Rulin Xu, Wei Dong, and Xiangke Liao. 2023. MulCS: Towards a Unified Deep Representation for Multilingual Code Search. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 120–131. https://doi.org/10.1109/SANER56733.2023.00021

[23] Yuetian Mao, Chengcheng Wan, Yuze Jiang, and Xiaodong Gu. 2023. Self-Supervised Query Reformulation for Code Search. arXiv:2307.00267 [cs.SE]

[24] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive Synthesis of API-Centric Code. arXiv:2201.03758 [cs.SE]

[25] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Migrating Code with Statistical Machine Translation. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 544–547. https://doi.org/10.1145/2591062.2591072

[26] Hoan Anh Nguyen, Hung Dang Phan, Samantha Syeda Khairunnesa, Son Nguyen, Aashish Yadavally, Shaohua Wang, Hridesh Rajan, and Tien Nguyen. 2023. A Hybrid Approach for Inference between Behavioral Exception API Documentation and Implementations, and Its Applications. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. https://doi.org/10.1145/3551349.3560434

[27] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 574–584.

[28] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[29] Hung Phan and Ali Jannesari. 2023. Evaluating and Optimizing the Effectiveness of Neural Machine Translation in Supporting Code Retrieval Models: A Study on the CAT Benchmark. *arXiv preprint arXiv:2308.04693* (2023).

[30] Hung Phan, Hoan Anh Nguyen, Ngoc M. Tran, Linh H. Truong, Anh Tuan Nguyen, and Tien N. Nguyen. 2018. Statistical Learning of API Fully Qualified Names in Code Snippets of Online Forums. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 632–642. https://doi.org/10.1145/3180155.3180230

[31] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[32] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. CoCoSoDa: Effective Contrastive Learning for Code Search. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2198–2210. https://doi.org/10.1109/ICSE48619.2023.00185

[33] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. 2022. Are We Building on the Rock? On the Importance of Data Preprocessing for Code Summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 107–119. https://doi.org/10.1145/3540250.3549145

[34] Zejian Shi, Yun Xiong, Yao Zhang, Zhijie Jiang, Jinjing Zhao, Lei Wang, and Shanshan Li. 2023. Improving Code Search with Multi-Modal Momentum Contrastive Learning. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*. 280–291. https://doi.org/10.1109/ICPC58990.2023.00043

[35] Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos. 2023. A Lightweight Framework for High-Quality Code Generation. arXiv:2307.08220 [cs.SE]

[36] Nikita Sorokin, Dmitry Abulkhanov, Sergey Nikolenko, and Valentin Malykh. 2023. CCT-Code: Cross-Consistency Training for Multilingual Clone Detection and Code Search. arXiv:2305.11626 [cs.CL]

[37] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2019. TreeGen: A Tree-Based Transformer Architecture for Code Generation. arXiv:1911.09983 [cs.LG]

[38] Xunzhu Tang, zhenghan Chen, Saad Ezzini, Haoye Tian, Yewei Song, Jacques Klein, and Tegawende F. Bissyande. 2023. Hyperbolic Code Retrieval: A Novel Approach for Efficient Code Search Using Hyperbolic Space Embeddings. arXiv:2308.15234 [cs.SE]

[39] Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. 2023. StructCoder: Structure-Aware Transformer for Code Generation. arXiv:2206.05239 [cs.LG]

[40] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[41] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. arXiv:2303.15822 [cs.SE]

[42] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable Neural Code Generation with Compiler Feedback. arXiv:2203.05132 [cs.CL]

[43] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).

[44] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. 2022. BabelTower: Learning to Auto-parallelized Program Translation. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 23685–23700. https://proceedings.mlr.press/v162/wen22b.html

[45] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. arXiv:2308.10462 [cs.SE]

[46] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[47] Wang Yue, Wang Weishi, Joty Shafiq, and Hoi Steven, C.H. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[48] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. (2023). https://arxiv.org/abs/2302.05527