# Enriching Queries for Code Search

Anonymous ACL submission

## Abstract

Existing works on natural language to code search improvements (Guo et al., 2020, 2022) focused on optimizing source code representation techniques, such as Abstract Syntax Tree (AST) or Data-flow graphs. The input object, Natural Language queries, are usually presented as sequences of tokens. Our proposed work, called QueryPlus, attempts to enrich meaningful information for NL queries in two directions: (a) extracting information from the Natural Language Parse Tree (NLPT), including Part-of-speech tagging and inter-token dependencies to provide syntactic information to natural language queries, and (b) using our designed shortened flattened sequence of tokens from the Abstract Syntax Tree, called Abstracted AST, which our proposed query-to-abstracted AST representation translation models can infer. Experiments on QueryPlus show that the proposed representations improve code search accuracy up to 2.4% of MRR improvement over the state-of-the-art (SOTA) BERT-based models UniXcoder and CoCoSoDa. Additionally, QueryPlus shows its flexibility and superiority in learning by legacy translation engine Neural Machine Translation (NMT). Compared to SOTA ASTTrans and UniXcoder' flattened AST Representation as target translation objects for supporting code search, QueryPlus achieves up to 5% of MRR improvement. Our source code and data are publicly available at: https://tinyurl.com/3zu3899k

## 1 Introduction

The field of Code Retrieval from natural language (NL) artifacts has gained significant traction within the domains of software engineering (SE) and natural language processing (NLP). Artificial Intelligence (AI) techniques have emerged as efficient tools for automated source code generation. The realm of code retrieval can be broadly divided into two approaches: generation/translation methods and code search strategies (Kulal et al., 2019; Guo et al., 2022). While translation techniques have demonstrated their effectiveness in generating source code for restricted formats of NL descriptions, such as pseudocode or behavioral exception specifications (Kulal et al., 2019; Nguyen et al., 2023), they encounter challenges when tasked with intricate datasets involving pairs of general code documentation and code snippets. In these scenarios, NL descriptions often take the form of queries, offering concise overviews of their corresponding implementations (Phan and Jannesari, 2023). As a result, code suggestion by natural language to code search is a more appropriate direction for suggesting complex code from concise NL queries (Feng et al., 2020; Guo et al., 2020; Liu et al., 2019).

Given that the efficacy of the code search process hinges heavily on the accuracy of representing both NL queries and candidates, various strategies have emerged for code representation. Prominent among these are CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), and UniXcoder (Guo et al., 2022), each offering distinct approaches. In these works, NL query's information is often represented as a mere sequence of words for these branches of approaches.

In this work, we provide two strategies for NL query representation. First, we provide an algorithm to extract information as Part of Speech (POS) tags and dependencies from the NLPT of a given NL query to be an extra input for NL query representation. Second, we propose a strategy to abstract the AST into a compact sequence of tokens. In this strategy, we train a model to generate our tokens as an abstracted version of AST using the unified cross-model translation engine UniXcoder (Guo et al., 2022). We evaluate our proposed strategies by fine-tuning the state-of-the-art (SOTA) pre-trained models for code search with our new representations for NL queries and evaluate the accuracy by well-known metrics in code search. We propose QueryPlus, a code search engine us-

ing alternative information from NLPT and AST. Overall, we have the following contributions:

1. Designing an algorithm for extracting part-of-speech tagging and dependencies information from the Natural Language Parse Tree of NL Queries.

2. Designing an algorithm for extracting Summarized Information from AST.

3. Proposing QueryPlus: a code search engine outperformed UniXcoder and CoCoSoDa models by fine-tuning SOTA approaches with our proposed NL query representations.

The next main sections are outlined as follows. Section Two introduces the baselines. Following this, we discuss motivation examples. The Approach section describes algorithms and overview architecture of our proposed tool QueryPlus. The Experiment section discusses six distinct research questions to evaluate our approach. The Result Analysis section offers a meticulous examination of the statistical significance of QueryPlus models compared to the state-of-the-art (SOTA) approaches.

## 2 Baselines

**Original BERT-based models.** For code search comparison, we have selected four models: Roberta, CodeBERT, GraphCodeBERT, and UniXcoder (Liu et al., 2019; Feng et al., 2020; Guo et al., 2020, 2022). We replicated the code search tasks over these models and achieved similar accuracy with their reported papers, showing that our replication is correct. We used and reported these replicated results as baselines for comparison. UniXcoder is the best tool compared to the other three models. **Thus, we select UniXcoder to represent the original BERT-based model for augmenting QueryPlus's representation for evaluation.**

**Optimized BERT-based models.** Following the advent of UniXcoder, numerous other code search models have arisen, prompting our inquiry. Phan and Jannesari, 2023 introduced ASTTrans, an investigation quantifying the effectiveness of their specialized AST Representation in reinforcing Neural Machine Translation's code search capabilities within the CAT benchmark (Shi et al., 2022). CoCoSoDa (Shi et al., 2023) proposed masking both NL query and candidate to eliminate noisy information. To our knowledge, CoCoSoDa is one of

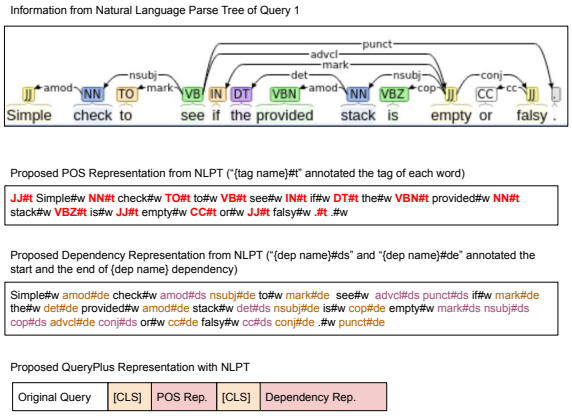


Figure 1: Proposed NLPT Representation of Query 1

the best-optimized BERT-based models for code search, with similar accuracy (77.91% compared to 77.95%) to the latest optimized BERT-based model CasCode (Gotmare et al., 2023), with the code published in April 2024. We select CoCoSoDa for optimization with QueryPlus. **In terms of optimized BERT-based models, we leverage ASTTrans and CoCoSoDa for comparison with QueryPlus.**

**CodeT5+.** CodeT5+ (Wang et al., 2023), harnesses an open large language model (LLM) for code search. We contacted the author of CodeT5+ and got the confirmation that the source code for fine-tuning the code search process with customized NL queries has not yet been available and requires large computation resources. **Thus, we didn't apply QueryPlus representation to existing CodeT5+ for our analysis.**

## 3 Motivation Example

We present a motivating example comprising a pair of NL queries and their corresponding code snippets sourced from the CodeSearchNet dataset, as depicted in Listing 1.

Listing 1: Javascript Implementation of Query 1: *"Simple check to see if the provided stack is empty or falsy"*

```
function empty(value) {
  if (!value) return true;
  switch (is(value)) {
    case "object": return !Object.keys(
        value).length;
    case "array": return !value.length;
    default: return !value;
  }
}
```

### 3.1 Proposed NLPT Representation

We extract the Natural Language Parse Tree (NLPT) of the query provided in the motivation

example using the widely recognized tool StanfordCoreNLP (Manning et al., 2014). Figure 1 illustrates the terminal nodes of the NLPT alongside their corresponding dependencies. This information encompasses two aspects. Firstly, the Part-of-Speech (POS) tags delineate the grammatical role of each word within the sentence. For instance, the terms *"check"* and *"stack"* are identified as singular nouns, while *"Simple"* assumes the form of an adjective. Secondly, the edges in the diagram signify the relationships between words as governed by the Penn Treebank project (Taylor et al., 2003). For instance, a prevalent dependency is the *"nominal subject"* relationship between the words *"see"* and *"check"*.

We approach the extraction of NLPT information through two distinct sequences. Initially, we denote the POS of each word using its corresponding POS tag. Subsequently, we incorporate details about dependencies by appending annotations indicating the initiation and termination of each dependency adjacent to the respective words. For instance, with the initial word *"Simple"*, our designated Dependency representation of the NL query encompasses the annotated token signifying that this word functions as the target of a *"amod"* dependency, with *"check"* serving as the point of origin for this dependency. Subsequently, we fuse the POS and Dependency representations to generate a novel representation of the original query. We hypothesize that this implementation strategy will enhance accuracy in code search. Ever, it is pertinent to note a potential limitation of this approach, as the revised NLPT representation may yield an excessive number of tokens.

### 3.2 Proposed Abstracted of AST Representation

In this proposed representation, we build upon two key intuitions. First, given that the Abstract Syntax Tree (AST) assumes a tree structure, it is plausible that non-terminal nodes could serve as a condensed form of summarized terminal nodes. Secondly, the Natural Language (NL) query itself can be perceived as an abstracted rendition of the AST representing its associated implementation. This proposed representation can offer two potential advantages. Initially, the abstracted AST representation, often referred to as "abstracted AST", encompasses a notably reduced number of tokens compared to the NLPT representation. Additionally, the abstracted AST captures the high-level structural information of the expected implementation of the NL query. This particular aspect remains absent in the NLPT representation.

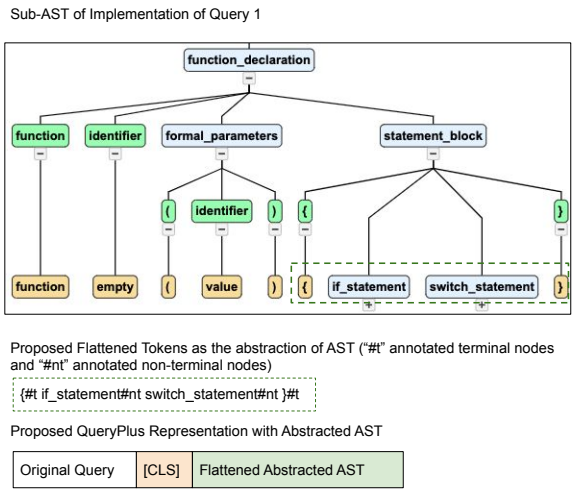


Figure 2: Proposed Abstracted AST Representation of Query 1

Our assumption is demonstrated through the abstracted AST Representation shown in Figure 2. Upon human interpretation, the query from Motivation Example One can be correlated with two distinct statement types. Firstly, an *"if"* statement is introduced to ascertain whether the stack is devoid of elements. Secondly, the conditional requirement to evaluate whether the provided stack is *"falsy"* is aligned with a switch statement, dictating the appropriate course of action for each *"value"*. Abstracting this AST by focusing solely on the children of the *"statement_block"* node necessitates a mere four nodes, incorporating two terminal and two non-terminal nodes. It is important to acknowledge a challenge associated with this representation specifically, the necessity of training a model to facilitate the translation from a query to its corresponding abstracted AST.

## 4 Approach

Illustrated in Figure 3, our model's overarching framework encompasses two following modules.

### 4.1 QueryPlus with NLPT

Given the input as the NL queries and a list of code snippets as candidates, this module will create code search models that consider the representation of NL query as the combination of original queries and their corresponding information about NLPT.

**NLPT Extraction.** This stage yields an anticipated outcome known as a graph representation

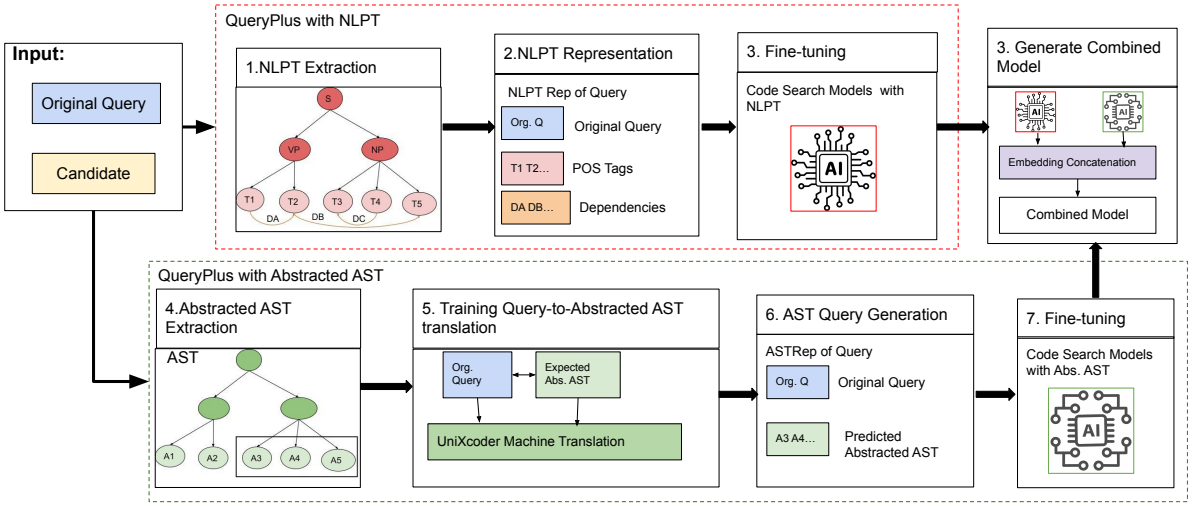


Figure 3: The Overview of QueryPlus

called the Natural Language Parse Tree. We employed StanfordCoreNLP (Manning et al., 2014) to extract NLPT.

**NLPT Token Generation.** We describe our algorithm for NLPT extraction in Algorithm 1. This algorithm accepts three inputs: the list of words, the list of POS tags, and the list of dependencies extracted from NLPT by the well-known in-order tree traversal algorithm (Cormen et al., 2022). At first, we construct 3 three lists to store information about POS tags, dependencies, and overall NLPT representation. Next, a for loop (Lines 5-9) will traverse each word in the list of words in the original NL query to enrich POS tags and dependencies to corresponding lists. After the loop, the $listNLPTReps$, as a list of strings, will be assigned as the combination of the original query, POS tag, and dependencies representation.

**Algorithm 1** NLPT Extraction

1: **function** NLPTEXTRACTION($listWords, listTags, listDeps$)
2: $listNLPTReps = []$
3: $listTagReps = []$
4: $listDepReps = []$
5: **for** $pos, word \in listWords$ **do**
6: $wordsTag = getWordAndTag(pos, listWords, listTags)$
7: $listTagReps.append(wordsTag)$
8: $wordsDep = getWordAndDep(pos, listWords, listDeps)$
9: $listDepReps.append(wordsDep)$
10: **end for**
11: $listNLPTReps = listWords + ['CLS'] + listTagReps + ['CLS'] + listDepReps$
12: **return** $listNLPTReps$
13: **end function**

**Fine Tuning Models with NLPT Representation.** We modify the fine-tuning steps of SOTA models by replacing original query with alternative queries with augmented information of NLPT.

## 4.2 QueryPlus with Abstracted AST

**Abstracted AST Extraction.** We use AST treesitter[1] to extract AST for each code snippet.We depict this process by Algorithm 2. The input of this algorithm is the AST representation of a source code snippet. There are two steps: finding the first AST node with type $statement_block$ (see Figure 2), called Function Body Root Node in Line 2, and adding its children's types to a list of tokens from Line 3 to Line 7.

**Algorithm 2** AST Abstraction

1: **function** ASTABSTRACTION($ast$)
2: $sbNode = getFirstStatementBlockNode(ast)$
3: $outputs = []$
4: **for** $childNode \in sbNode.children$ **do**
5: $output.append(childNode.type)$
6: **end for**
7: **return** $outputs$
8: **end function**

**Training Query-to-Abstracted AST translation** To build the training model, we conduct the parallel corpus of the source language as the original NL query and the target language as the extracted tokens as the types of children of *"block"* node of the AST of the candidate. We use the pre-trained model UniXcoder to fine-tune it for query-to-abstracted AST Representation generation.

**AST Query Generation** Since the Abstracted AST representation of each code snippet cannot be

[1]https://tree-sitter.github.io/tree-sitter/

extracted from NL query, we leverage the trained query-to-abstracted AST generation model to predict the sequence of abstracted AST representation from the NL query.

**Fine Tuning Models with Abstracted AST Representation.** We replace the original query by the augmented query with an Abstracted AST representation outputted from the translation model above for fine-tuning SOTA models.

### 4.3 Generate Combined Models.

QueryPlus generates three distinct code search models: NLPT-based, Abstracted AST-based, and a Combined model. As previously outlined, the NLPT-based and Abstracted AST-based models stem from their respective modules. The Combined model results from merging the embeddings of two code search models from fine-tuned with two NL query representation strategies to execute code search tasks. For the combination, we combine two embedding spaces similar to the approach proposed in ASTTrans (Phan and Jannesari, 2023). Different from ASTTrans, which set a high weight on one of two embedding models when combined, in our work, we set the same weight for QueryPlus with NLPT and QueryPlus with Abs. AST as 0.5.

## 5 Experiments

We conduct the following research questions (RQ):

1. RQ 1. How well can QueryPlus with NLPT fine-tuned on UniXcoder perform compared to original BERT-based models?

2. RQ 2. How well can QueryPlus with Abstract AST fine-tuned on UniXcoder perform compared to original BERT-based models?

3. RQ 3. How well can QueryPlus with Combined Models from NLPT and AST Models fine-tuned on UniXcoder perform compared to original BERT-based models?

4. RQ 4. Can QueryPlus representation bring benefits to optimized BERT-based models?

5. RQ 5. Can QueryPlus representation work with query-to-Abs. AST generation using classical Machine Translation engines?

6. RQ 6. Can QueryPlus representation bring more advantages to code search than well-known flattened representation approaches?

Table 1: Statistics on Datasets

| Dataset | Train | Valid | Test | Codebase |
|---|---|---|---|---|
| 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 |
| AdvTest | 251820 | 9604 | 19210 | 19210 |
| cosqa | 19604 | 500 | 500 | 6267 |

**Benchmark.** We use similar datasets for evaluation with UniXcoder (Guo et al., 2022). They are the CodeSearchNet benchmark with six datasets of different programming languages and two extra datasets AdvTest and Cosqa, in Python. Information about datasets is shown in Table 1.

**Configuration.** The training of both code search and query-to-abstracted AST Representation models was performed on an HP OMEN 30L GT13 gaming desktop equipped with a Core i9 processor (16 cores), 100 GB of RAM, and an RTX 3090 GPU card boasting 24 GB of memory. We maintained consistency by utilizing the default hyperparameters for fine-tuning, as specified in the UniXcoder original model (Guo et al., 2022). For evaluation metrics, we use Mean Reciprocal Rank (MRR), a metric used in SOTA approaches.

### 5.1 Research Question (RQ) 1. Enhancing Code Search with NLPT Representation

The results presented in Table 2 underscore the efficacy of NLPT Representation in bolstering the performance of the original UniXcoder model, yielding an average accuracy enhancement as 70.52% of MRR. This augmentation extends across all eight datasets, with the Java dataset exhibiting the most substantial increase in MRR by 1.4%, while the AdvTest dataset witnessed a more modest improvement of 0.13%. A compelling illustration of NLPT Representation's potency is encapsulated in Listing 1, wherein the original UniXcoder model placed the accurate code candidate at rank 2. At the same time, the NLPT representation facilitated its ascension to the top rank. **In summary, NLPT Representation markedly amplifies the effectiveness of the code search process over conventional query representation.**

Table 2: Accuracy Meassured by Mean Reciprocal Rank of QueryPlus over Baselines and Varied Configurations

| | CodeSearchNet | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Model** | **java** | **python** | **javascript** | **go** | **php** | **ruby** | **AdvTest** | **cosqa** | **Avg** |
| **RQ 1-3: Comparing original BERT-based models with QueryPlus (QP) fine-tuned on UniXcoder** | | | | | | | | | |
| RoBERTa | 59.90% | 58.70% | 51.70% | 85.00% | 56.00% | 58.70% | 18.30% | 60.30% | 56.08% |
| CodeBERT | 67.60% | 67.20% | 62.00% | 88.20% | 62.80% | 67.90% | 27.20% | 65.70% | 63.58% |
| GraphCodeBERT | 68.70% | 68.57% | 65.05% | 89.57% | 64.75% | 71.01% | 38.05% | 70.10% | 66.98% |
| UniXcoder | 72.28% | 71.43% | 68.87% | 91.35% | 66.87% | 73.74% | 42.17% | 72.27% | 69.87% |
| QP (UniXcoder) | | | | | | | | | |
| **- RQ1 (NLPT)** | 73.72% | 72.72% | 69.22% | 91.55% | 67.85% | 74.23% | 42.30% | 72.55% | 70.52% |
| **- RQ2 (Abstracted AST)** | 73.33% | 72.76% | 69.75% | 91.48% | 67.42% | 75.04% | **42.65%** | 73.49% | 70.74% |
| **- RQ3 (Combine)** | **73.82%** | **72.89%** | **69.81%** | **91.61%** | **68.11%** | **75.12%** | 42.63% | **73.54%** | **70.94%** |
| **RQ 4: Comparing QP fine-tuned on CoCoSoDa vs original CoCoSoDa** | | | | | | | | | |
| CoCoSoDa | 75.14% | 74.74% | **75.32%** | 92.08% | 69.26% | 80.94% | 41.08% | 67.61% | 72.02% |
| $QueryPlus_{CoCoSoDa}$ | **75.51%** | **74.89%** | 75.26% | **92.17%** | **69.48%** | **81.10%** | **43.51%** | **70.10%** | **72.75%** |
| **RQ 5: QueryPlus (Abs. AST) with query-to-Abs. AST using NMT** | | | | | | | | | |
| $QueryPlus_{UniXcoder}$ | **73.25%** | **72.58%** | **69.48%** | **91.44%** | **67.37%** | **74.63%** | **42.47%** | **73.49%** | **70.59%** |
| $QueryPlus_{CoCoSoDa}$ | **75.46%** | **74.85%** | **75.17%** | **92.34%** | **69.56%** | **81.11%** | **43.36%** | **68.61%** | **72.56%** |
| **RQ 6: Evaluation of Replacing QueryPlus Rep by Code Tokens Rep and ASTTrans Rep** | | | | | | | | | |
| Uni. Code Rep. (UniXcoder) | **67.65%** | N/A | N/A | N/A | N/A | N/A | N/A | N/A | **67.65%** |
| ASTTrans (UniXcoder) | **68.04%** | N/A | N/A | N/A | N/A | N/A | N/A | N/A | **68.04%** |
| Uni. Code Rep. (CoCoSoDa) | **67.52%** | N/A | N/A | N/A | N/A | N/A | N/A | N/A | **67.52%** |
| ASTTrans (CoCoSoDa) | **71.58%** | N/A | N/A | N/A | N/A | N/A | N/A | N/A | **71.58%** |

### 5.2 RQ 2. Enhancing Code Search with (Predicted) Abstracted AST Representation

This configuration yielded an average MRR enhancement of 70.74%, with the AdvTest dataset again demonstrating the least notable improvement. Intriguingly, analysis of the generated tokens from the abstracted AST representation revealed an average token count of merely 4.5 across the eight datasets, significantly fewer than the number the tokens augmented by the NLPT representation (15 tokens on average). **In comparison to NLPT Representation, the projected Abstracted AST Representation enhances UniXCoder's code search performance while substantially requiring less number of augmented tokens needed to NL query representation compared to NLPT Representation.**

### 5.3 RQ 3. Enhancing Code Search with Combined Model

In this RQ, we evaluate the combined code search process using the combination of embeddings from two previous strategies of NL Representation. We achieve the best accuracy on average at 70.94%. The combined configuration achieved higher accuracies in seven of eight datasets except AdvTest. This configuration achieved the best improvement for UniXcoder in solving Java NL queries, with the enhancement of 1.5%. **In sum, QueryPlus combined model can improve original BERT-based model UniXcoder as over 1% of MRR score on average.**

### 5.4 RQ 4. Evaluating QueryPlus fine-tined on CoCoSoDa

We use our constructed QueryPlus Combined Model for this experiment, trained on the latest code search model CoCoSoDa. While Shi et al. (Shi et al., 2023) used to Nvidia Tesla A-100 GPU with 48 GB of memory for their fine-tuning process, they config their batch size as 64. Since our resource is limited, with reduced CoCoSoDa's training batch size to 48 and used other default configurations. Additionally, while CoCoSoDa only evaluated their performance on six datasets of CodeSearchNet, we extend our replicated experiments of CoCoSoDa to 8 datasets, which we include the cosqa and AdvTest dataset that is used in UniXcoder (Guo et al., 2022). We observe that our replicated result for CoCoSoDa, shown in Table 2, is about 0.5% lower than the MRR score reported in Shi et al.'s paper (Phan and Jannesari,

2023).

With the new representation of QueryPlus for fine-tuning CoCoSoDa, we were able to improve the CoCoSoDa to be up to 72.75%. For additional datasets AdvTest and cosqa, QueryPlus improved as up to 2.5% of MRR improvement. We observed the low accuracy of CoCoSoDa on these datasets was due to the fact that it was pre-trained and fine-tuned on CodeSearchNet datasets only. We also compare our results over the tool developed by Salesforce, CodeT5+ (Wang et al., 2023) (see Appendix). Our query representation can improve CoCoSoDa to achieve comparable accuracy as CodeT5+ (72.75% of QueryPlus compared to 72.86% of CodeT5+ reported results). It's worth to note that CodeT5+ is industrial tool which cannot be replicated on a single GPU like CoCoSoDa. Thus, QueryPlus can improve CoCoSoDa's accuracy without maintaining its low cost of fine-tuning process.

**In summary, QueryPlus can gain the accuracy of CoCoSoDa as 0.7% of MRR on average, with its advantages on code search over datasets that are unseen from the pre-training process.**

### 5.5 RQ 5. Evaluating the performance of QueryPlus using Query-to-Abstracted AST Representation translation by Neural Machine Translation

In this RQ, we want to test our hypothesis that QueryPlus representation can work well with classical machine translation engines like Neural Machine Translation (NMT). We use OpenNMT library (Klein et al., 2017) with default configuration for training our model for query-to-Abstract AST representation translation. We modified the configuration in RQ3 by using NMT as the translation models for the configuration of RQ5.

Overall, we achieved 70.59% of MRR over eight datasets when we fine-tuned QueryPlus on UniXcoder. For CoCoSoDa, QueryPlus using NMT can achieve an MRR of 72.56%. Compared to the running time, the UniXcoder translation model required 0.2 seconds to generate the predicted result for each query, while NMT required 0.14 seconds to do the same task. **This shows that QueryPlus can be flexible using cheaper machine translation engines.**

Table 3: Analysis on Top-K accuracy of RQs 1-3' Results

| | Top-K accuracy | | | | |
|---|---|---|---|---|---|
| **Model** | 1 | 2 | 3 | 4 | 5 |
| UniXcoder | 60.63% | 71.50% | 76.19% | 79.28% | 81.39% |
| RQ1 (NLPT) | 61.34% | 72.15% | 76.88% | 79.73% | 81.87% |
| RQ2 (AST) | 61.80% | 72.23% | 76.96% | 79.74% | 81.76% |
| RQ3 | 61.96% | 72.49% | 77.24% | 79.88% | 82.09% |

Table 4: Analysis by case-to-case comparison of QueryPlus (Combined) models in RQ 3 over original UniXcoder models

| Language | Better | Draw | Worse |
|---|---|---|---|
| java | 17.37% | 73.26% | 9.37% |
| python | 17.98% | 71.81% | 10.20% |
| javascript | 18.72% | 68.25% | 13.04% |
| go | 3.20% | 94.41% | 2.39% |
| php | 20.14% | 66.98% | 12.88% |
| ruby | 13.48% | 77.64% | 8.88% |
| Avg. | 15.15% | 75.39% | 9.46% |

### 5.6 RQ 6. Evaluating QueryPlus's code search pipeline over ASTTrans Representation

In this RQ, we want to test the hypothesis that whether or not other code representation techniques can replace our proposed NLPT and Abstract AST representation for code search improvement. We select two techniques: ASTTrans (Phan and Jannesari, 2023) and UniXcoder's flattened AST (Guo et al., 2022) code representation. We replicate UniXcoder and CoCoSoDa with the integration of two machine translation models: query-to-ASTTran and query-to-Flattened AST models. In the evaluation, we show that our QueryPlus representations outperform both ASTTrans and UniXcoder code representation techniques by up to 5% as shown in RQ6 of Table 2. **QueryPlus performed significantly better than SOTA approaches for code representation in enriching natural language queries for code search improvement.**

## 6 Result Analysis

### 6.1 Top-K accuracy

The outcomes of Top-K, measured similarly with ASTTrans (Phan and Jannesari, 2023), accuracy for the code search procedures conducted by UniXcoder and our three models are presented in Table 3. It is evident that all four models have attained

a remarkable range, with top-1 accuracy exceeding 60% and top-5 accuracy exceeding 80%. The trends in Top-K accuracy across the three models mirror those observed in MRR accuracy. Notably, the combined model showcased the most impressive performance in Top-K accuracy, boasting a notable enhancement of over 1.3% in top-1 accuracy and 0.7% in top-5 accuracy compared to the SOTA models.

### 6.2 Case-to-case Comparison

We delve into three distinct scenarios of comparison for each query: "Better" (indicating the outcome from the combined model surpassing that of the SOTA model), "Worse" (denoting the original model outperforming the combined model), and "Draw" (suggesting that the correct candidate held the same rank in both the original model and QueryPlus combined model's suggestions). The results, as showcased in Table 4, illustrate that QueryPlus influenced approximately 24% of queries across the eight datasets. Notably, it exhibited a 6% increase in cases where it yielded improved results compared to those where it had worse accuracy. We discern two potential causes for the "Worse" cases. For NLPT representation, these scenarios often surfaced when the NLPT representation required too many tokens (see Appendix's example). As for abstracted AST representation, performance degradation typically occurs when the quality of the query-to-abstracted AST translation is compromised. These observations hint at the potential for QueryPlus's enhancement through strategies like curtailing the NLPT representation length and developing mechanisms to predict instances where the query-to-abstracted AST representation didn't output good translated outcomes.

### 6.3 Running Time

Compared to SOTA approaches, our approach required two extra processes: generating NLPT representation and Abstracted AST representation. Nevertheless, the running time of these two processes does not remarkably slow down the code search process. We calculate the running times for evaluating code search SOTA CoCoSoDa and QueryPlus fine-tuned on CoCoSoDa (RQ4) (over 72271 test queries in total, shown in Table 1) are 1089 and 1175 seconds. We consider the difference to be acceptable, thanks to the fact that the NLPT extraction usually has input as short queries, while the Abstracted AST extraction module only needs to generate a short sequence of tokens using machine translation.

### 6.4 Roles of NLPT

We want to analyze why information from NLPT can be helpful by conducting the following experiment. We randomly select 50 queries and ask evaluators to manually check whether or not the queries' POS tags or dependencies relate to one aspect of their corresponding code. For example, in Figure 1, we can manually see that the two last adjectives ("empty" and "falsy"), along with their dependency as "conjunction," actually related to two conditions in implementation. We use two evaluators who are computer scientists with English as their first language to experiment, and they are required to discuss and select the final answers when they have different evaluations on the same query. The experiment shows that 43 (86%) of the selected queries have their NLPT content mapping to code's information.

## 7 Related Work

Ma et al., 2023 constructed a semantic graph representation at Intermediate Representation (IR) for Java, C/C++ and Python. Tang et al., 2023 proposed HyCoQA and leveraged the characteristics of Hyperbolic spaces to extract the relationship between code fragments and intricate interaction layers. Gotmare et al., 2023 proposed CasCode, an optimized code search model by providing two strategies of query-code comparison. Different from CoCoSoDa and our work QueryPlus, this work applied the optimization at query-code comparison step that it reranked the list of candidates after the fine-tuning steps. CasCode achieved similar accuracy over six standard datasets of CAT benchmark compared to our replicated result from CoCoSoDa.

## 8 Conclusion

In this work, we propose a new direction for improving the content of NL query by adding two sources of information: NLPT and Abstracted AST Representation. Experiments show that our technique for code search, QueryPlus, can bring benefits for both original BERT-based models and optimal BERT-based models. In future works, we will improve our query-to-abstracted AST representation and add SE-based properties to NLPT representation to enhance their benefits for code search and other SE tasks.

## 9 Limitations

We identify fourth limitations of our work. First, for RQ3, due to limited computation resources, we have to reduce the training batch size of CoCoSoDa (Shi et al., 2023) from 64 to 48, which causes the accuracy of CoCoSoDa decreased by 0.5% from the results reported by Shi et al. We mitigated this threat by cloning the source code from CoCoSoDa authors' repository and got confirmation from the authors that our replication process is correct. Second, we didn't fine-tune our models with different hyperparameters due to the restriction of our computation resources. We mitigate this limitation by selecting default parameters used in SOTA approaches. Third, due to space limitations, we didn't report our other studies on the statistical significance of our approaches besides the metrics report in the Result Analysis section. We upload reports on these studies in our replication package repository. Finally, for RQ6, we only replicate ASTTrans for Java language. We contacted the author of ASTTrans and got confirmation that ASTTrans extraction was only available for Java language at the time of submission. Since ASTTrans operated on AST treesitter[2], it can be extendable for supporting different languages in the future.

## 10 Ethical Consideration

We ensured that all the data we generated while creating this paper was anonymized and that no personally identifiable information (PII) was shown in our data. We didn't use any human annotating process for this paper. While we do not anticipate that the novel work presented here to raise new ethical concerns by itself, we realize that there may also be pre-existing concerns and issues with the input natural language to code search benchmark CodeSearchNet (Husain et al., 2019) we used as the input. Due to resource limitations, we cannot check all of the natural language queries and code snippets in this benchmark to check whether or not they contain any sensitive content. However, we checked manually and randomly 100 pairs of natural language queries and their code snippets, and we didn't see any ethical issues. Moreover, the CodeSearchNet benchmark has been considered a well-known benchmark and to the best of our knowledge, there wasn't any critical about this benchmark's content.

[2]https://tree-sitter.github.io/tree-sitter/

## References


Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Akhilesh Deepak Gotmare, Junnan Li, Shafiq Joty, and Steven C.H. Hoi. 2023. Efficient text-to-code retrieval with cascaded fast and slow transformer models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 388–400, New York, NY, USA. Association for Computing Machinery.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint*.

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.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. Opennmt: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada. Association for Computational Linguistics.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. Spoc: Search-based pseudocode to code. *arXiv preprint*.

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. *Preprint*, arXiv:1907.11692.

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)*, pages 120–131.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland. Association for Computational Linguistics.

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*, ASE '22, New York, NY, USA. Association for Computing Machinery.

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*.

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*, ICSE '23, page 2198–2210. IEEE Press.

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*, ESEC/FSE 2022, page 107–119, New York, NY, USA. Association for Computing Machinery.

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. *Preprint*, arXiv:2308.15234.

Ann Taylor, Mitchell Marcus, and Beatrice Santorini. 2003. The penn treebank: an overview. *Treebanks: Building and using parsed corpora*, pages 5–22.

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*.

## A Artifacts

The replication package of this paper (mostly source code) can be found at: https://tinyurl.com/3zu3899k Related data and results that were too heavy to upload to the replication package can be found in: https://tinyurl.com/chxyby3x

## B Patterns of Results

We manually and randomly checked 100 results returned by QueryPlus models compared to original SOTA models from configurations of RQ1, RQ2, RQ3, and RQ4. We observe a pattern that long NL queries (more than 12 tokens) tend to have better accuracy with Abstracted AST augmentation (i.e. models provided in RQ1), while short queries can have better accuracy with NLPT augmentation. An example of cases when Abstracted AST augmentation improved the accuracy compared to NLPT augmentation is shown in Listing 2. Here, while the NLPT Representation necessitates over 30 tokens, the Abstracted AST Representation requires a mere three tokens: "lexical_declaration", "if_statement" and "return_statement".

Listing 2: Case Study: Javascript Implementation of Query: *"_removeSubscribers remove the subscribers to one channel and return the number of subscribers that have been unsubscribed."*

```
function _removeSubscribers(aSubscribers
    , oSubscriber) {
  let nUnsubscribed = 0;
  if (!isTypeOf(aSubscribers,
      sNotDefined)) {
    let nIndex = aSubscribers.length -
        1;
    for (; nIndex >= 0; nIndex--) {
      if (aSubscribers[nIndex].
          subscriber === oSubscriber) {
        nUnsubscribed++;
        aSubscribers.splice(nIndex, 1);
      }
    }
  }
  return nUnsubscribed;
}
```

## C Hyper Parameters

The details of all hyperparameters can be seen from the documentation of the replication package.

**Training query-to-abstracted AST Translation Models with UniXcoder.** We use configuration metrics shown in Table 5. The training time for eight datasets is around 4 hours.

**Training query-to-abstracted AST Translation Models with Neural Machine Translation.** We use configuration metrics shown in Table 6. The training time for eight datasets is around 3 hours.

**Fine-tuning Code Search Models.** We use the configurations shown in Table 7 for replicating SOTA models and our proposed models that were fine-tuned on UniXcoder and CoCoSoDa. For fine-tuning eight datasets, the training time is around 8.5 hours.

Table 5: Configuration for Training Translation Models with UniXcoder

| Hyper Parameter | Value |
|---|---|
| beam_size | 3 |
| train_batch_size | 32 |
| eval_batch_size | 32 |
| learning_rate | 5.00E-05 |
| gradient_accumulation_steps | 1 |
| num_train_epochs | 30 |

Table 6: Configuration for Training Translation Models with Neural Machine Translation

| Hyper Parameter | Value |
|---|---|
| train_steps | 50000 |
| valid_steps | 10000 |
| encoder_type | RNN |
| decoder_type | RNN |
| hidden_size | 512 |
| layers | 2 |
| learning_rate | 1 |
| batch_size | 48 |

## D Comparison of QueryPlus fine-tuned on CoCoSoDa and CodeT5+

. We reported the comparison of MRR scores between original CoCoSoDa and optimized CoCoSoDa by QueryPlus along with CodeT5+ reported accuracy in Table 8.

Table 7: Configuration for Fine-tuning Code Search Models

| Hyper Parameter | Value |
|---|---|
| num_train_epochs | 10 |
| train_batch_size | 48 |
| eval_batch_size | 48 |
| learning_rate | 2.00E-05 |
| seed | 123456 |

Table 8: Comparison between optimized CoCoSoDa with QueryPlus Representation vs CodeT5+ Reported Accuracy

| Language\Tool | CoCoSoDa | QueryPlus | CodeT5+ |
|---|---|---|---|
| **java** | 75.14% | 75.51% | 76.20% |
| **python** | 74.74% | 74.89% | 75.80% |
| **javascript** | 75.32% | 75.26% | 71.30% |
| **go** | 92.08% | 92.17% | 92.70% |
| **php** | 69.26% | 69.48% | 70.10% |
| **ruby** | 80.94% | 81.10% | 78.00% |
| **AdvTest** | 41.08% | 43.51% | 44.70% |
| **cosqa** | 67.61% | 70.10% | 74.00% |
| **Avg** | 72.02% | 72.75% | 72.85% |

## E Roles of NLPT (Artifacts)

We stored the artifact with 50 random selected queries spanning over six standard datasets of CodeSearchNet benchmark, along with their human judgment about whether or not their NLPT contained code-related information in folder "result_roles_NLPT" folder in the related data/results link. Please see the "Readme.md" file for know details about this artifact.

## F Additional Metrics for Significant Analysis

We saved results for additional metrics for analysis of the statistical significance of models in folder "result_extra_significant_test" in the related data/results link.