2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678929

Interactive Cross-language Code Retrieval with Auto-Encoders

Binger Chen
TU Berlin
Berlin, Germany
chen@tu-berlin.de

Ziawasch Abedjan
Leibniz Universität Hannover & L3S Research Center
Hannover, Germany
abedjan@dbs.uni-hannover.de

***Abstract*—Cross-language code retrieval is necessary in many real-world scenarios. A major application is program translation, e.g., porting codebases from an obsolete or deprecated language to a modern one or re-implementing existing projects in one's preferred programming language. Existing approaches based on the translation model require large amounts of training data and extra information or neglects significant characteristics of programs. Leveraging cross-language code retrieval to assist automatic program translation can make use of Big Code. However, existing code retrieval systems have the barrier to finding the translation with only the features of the input program as the query. In this paper, we present BIGPT for interactive cross-language retrieval from Big Code only based on raw code and reusing the retrieved code to assist program translation. We build on existing work on cross-language code representation and propose a novel predictive transformation model based on auto-encoders. The model is trained on Big Code to generate a target-language representation, which will be used as the query to retrieve the most relevant translations for a given program. Our query representation enables the user to easily update and correct the returned results to improve the retrieval process. Our experiments show that BIGPT outperforms state-of-the-art baselines in terms of program accuracy. Using our novel querying and retrieving mechanism, BIGPT can be scaled to the large dataset and efficiently retrieve the translation.**

I. Introduction

The number of open-source program resources on the internet is constantly growing. The most well-known are open source code repository hosts, such as GitHub and Bitbucket. The GitHub database, i.e., the Public Git Archive [5], contains more than 260,000 GitHub repositories, which are written in 455 different programming languages and include more than 16 billion lines of code, in the HEAD files alone. Further, community question answering sites, such as Stack Overflow, contain a large number of executable binaries that amount to billions of code snippets. These and similar resources are referred to as “Big Code” [37], which are created and modified by programmers with much effort and time. Reuse of code from these abundant databases provides opportunities for new applications, such as workflow generation [16], data preparation [44], programming assistance [25], database management [20], and transformation retrieval [43]. Another application that has recently emerged in this context is program translation [11], [39].

A useful technique that can support several of the aforementioned applications is code retrieval. In particular, *cross-language retrieval* is becoming more prominent for use cases, such as program translation and code-clone detection. Numerous programs are being developed and require corresponding versions in different languages. In cases when the developers do not make the translation efforts themselves, users have to manually rewrite the software in the needed language. For example, there are plenty of open-source prototypes developed in academia, especially in the current booming field of big data. To port codebases written in obsolete or deprecated languages to a modern one [39], or further study, reproduce, or apply them on various platforms, researchers usually need to rewrite these programs in their preferred programming languages. Manually rewriting software is time-consuming and error-prone. For instance, the Commonwealth Bank of Australia spent around $750 million and 5 years to translate its platform from COBOL to Java [39]. Therefore, new approaches for automated program translation and code migration are emerging [33]. The traditional methods are hardwired, rule-based compilers or cross-language interpreters, which require heavy human intervention for adaptation and are limited to a small set of programming languages [3]. However, if we leverage a cross-language retrieval system, we can make the most of the existing Big Code resources to support the program translation use case. In this paper, we discuss the potentials of an effective cross-language retrieval system in assisting program translation.

State of the art. Our work is inspired by two lines of research, code retrieval and supervised program translation.

Most existing code retrieval systems, such as Sourcerer [26], lack the proper capabilities for code-to-code search and/or cross-language retrieval. The cross-language system YOGO [35] requires users to provide pre-defined handwritten pattern queries or feedback on several preset metrics and questions as many other code search systems [18], [28], [40], [41], which increase the workload of users. Our recently proposed system RPT [11] aims to retrieve the translation only with the feature of the source code and ignores the language-specific differences to the target language. All of the aforementioned techniques exclude users from the retrieval/translation loop.

Similar to natural language translation, the mainstream data-driven program translation approaches train a transla-

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00025

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

tion model from large amounts of code data either in a supervised [12], [31], [32] or weakly-supervised fashion [39]. Supervised approaches require a *parallel dataset* to train the translation model. In parallel datasets, programs in different languages are considered to be "semantically aligned". Obtaining the parallel datasets in programming languages is hard because the translations have to be handwritten most of the time. A recent weakly-supervised method by Facebook AI [39] pretrains the translation model on the task of denoising randomly corrupted programs and optimizes the model through back-translation. However, this method still relies on high-quality training data and directly reuses natural language processing (NLP) approaches that neglect the special features of programming languages, such as code syntax. All these approaches require human efforts on the input, or additional information for training and evaluation, such as annotations, program descriptions, API usages, and use cases, which cannot always be provided. Furthermore, compared to retrieval methods, these machine-generated program translations suffer from grammar mistakes because of the rigor of programming languages.

Our methodology is to reuse existing Big Code resources and developing cross-language retrieval techniques to assist translation with retrieved similar code in target language. Note that such a method might fail to find the translation for very specific long and complex programs because the achievable performance is directly depending on the richness of the given repository. Nevertheless, due to the modular nature of programs and the huge data volume of the big code, it should be possible to retrieve the translation for smaller fragments of an input program, such as methods and functions. The user then can use these building blocks to assemble the complete translation. We improve and extend our preliminary attempt [11] on supporting code translation through Big Code by addressing the following open *challenges* and *requirements*:

- Big Code resources are typically imperfect and disordered. Thus, it is hard to obtain correct translations in absence of sufficient information, such as training data, hand-crafted patterns, and semantic annotations.
- Because of the different syntax structures and naming mechanisms in different programming languages, it is hard to capture program features in a unified way. Ideally one has to resort to an intermediate representation that is automatically extractable from raw code. With this, it would be possible to use similarity metrics already in place for code clone detection. But generating such an intermediate representation is yet an open task.
- Representation techniques that are in place for natural language are not designed to deal with the special syntax and grammatical rigor of code.
- The feature representation has to be incrementally updatable to enable user interaction.

In this paper, we present **BIGPT**, an interactive cross-language code retrieval system that reusing Big Code resources to assist the program translation task. Given a raw piece of code in the source language, BIGPT uses a novel query transformation approach based on auto-encoders to retrieve the most similar piece of code in the selected target language from the existing Big Code resources. We propose a query transformation model, which can be trained in an unsupervised manner on Big Code to transform the input program representation to a representation that is closer to properties of the target language. Due to the succinct form of the program representation, the user can interact with BIGPT and the query can be automatically adapted to user annotations in the target code. As BIGPT is a heuristic methodology based on retrieving real programs, it is less prone to grammar mistakes than approaches based on code generation. Our experiments show that our approach outperforms existing cross-language retrieval techniques and statistical translation models that require a large amount of training data. To this end, we make the following **main contributions**:

- We present a program feature representation for effective retrieval of possible translations in imperative programming languages. With the help of Big Code, we propose a novel auto-encoder-based query transformation model, which can transform the input code into the representation of its translation.
- We design the feature representation in a way that it can be incrementally updated based on user corrections to efficiently retrieve the translation from Big Code.
- We further expand our feature representation with a weighting scheme to enable user feedback that accelerates BIGPT's ability to improve its retrieval results.

II. Related Work

Our work is related to code search, data-drive program translation, and program representation. We also discuss related work in cross-language code clone detection and natural language retrieval to show the originality of our work.

General code search. Most of the existing methods cannot find translations with only raw code as input. Krugle and Codase are commercial engines that apply the capabilities of web search engines for code search [2], [4]. Lucene is a famous conventional code search engine behind many existing tools such as Sourcerer [26]. It retrieves code based on text and code properties, such as fully qualified name and code popularity. S6 retrieves code based on the user's specifications and modifies it through a set of transformations [38]. CodeHow is a text-based code search engine that incorporates an extended boolean model and API matching [27]. DEEPCS trains a neural network model for code snippets and their natural language description, which are used as queries [19]. FaCoY is a code-to-code search engine that requires Q&A posts from Stack Overflow as query addition to the code snippets [23]. YOGO provides a cross-language graph representation, which however requires handwritten semantic rules and patterns written in a domain-specific language for each query [35]. All these works require users or additional resources to provide keywords, specifications, rules, or natural language descriptions. Recently, we proposed RPT, which uses cross-language retrieval for translations [11]. In this

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

preliminary work, we used the features of the input program to retrieve the translation and did not support user interaction although retrieval-based methods cannot always provide off-the-shelf translations. Except YOGO and RPT, all of the aforementioned systems are designed for the mono-language setting. In contrast to these systems, BIGPT only takes raw code as the query and aims to perform cross-language retrieval.

Interactive code search. Wang et al. refine a query based on user's feedback on each result and reorder the ranking list [41]. Nie et al. extract relevant feedback from StackOverflow for the initial query and reformulate it using Rocchio expansion [34]. Dietrich et al. utilize a novel form of association rule mining to learn a set of query transformation rules from user feedback to improve the search queries. CodeExchange leverages the context from previous retrieval results to reformulate a given query and breaks it into several parts for the user to feedback [28]. Sivar et al. propose an active learning system ALICE to iteratively refine a query based on positive or negative labels [40]. All of the aforementioned methods are only applicable to mono-language scenarios and work with natural language queries. As for interaction, these methods ask the user to give feedback on preset metrics based on Likert scale or questions, e.g., StackOverflow Q&A pairs. Thus, the performance mainly depends on the developer's ability to formulate queries. BIGPT directly uses user's corrections to the retrieved results as feedback and integrates an active learning-based query transformation model to refine the query.

Data-Driven program translation. Existing work mainly focuses on building a translation model. Nguyen et al. applied the phrase-based statistical machine translation (SMT) model on the lexemes of source code to translate Java code to C# [31]. In their follow-up work, they develop a multi-phase, phrase-based SMT method that infers and applies both structure and API mapping rules [32]. But they are limited to languages that are similar on either structural or textual level, such as Java/C#. Chen et al. binarize the code tree of a piece of code and translate the code with an LSTM-based encoder-decoder model [12]. All the above methods require a large parallel dataset for training. In contrast to them, the weakly-supervised system TransCoder [39] first trains a cross-language model through the task of predicting randomly masked words, then acquires a pre-trained translation model from denoising randomly corrupted program task. Finally, they improve this model through back-translation. Although TransCoder does not need parallel translation data, this transfer learning method highly relies on the similarity of the data for pre-training. Their approach processes code like natural language, which might leave out some programming-specific language features. We propose a system that can assist program translation without parallel datasets and additional information. By reusing Big Code, it can comply with the rigorous grammar without machine-generated translation. Further, our system outperforms the aforementioned techniques with a novel program representation that captures the most crucial features of programs.

Program representation. By constructing program representations, one can enable the application of data processing to a wide range of programming-language tasks including program translation and code search. Kamiya et al. and Allamanis et al. treat a program as plain text and use the sequence of tokens as representation to detect code clones and summarize code [8], [22]. Allamanis et al. present a Gated Graph Neural Network in which program elements are represented by graph nodes and their semantic relations are edges in the graph to predict variable name and select correct variable [7]. These methods rely on semantic knowledge, which requires expert analysis and is not generalizable across programming languages. A recent approach uses paths in the program's abstract syntax trees (AST) as code representation to predict program properties such as names or expression types [9]. And they further propose Code2vec that leverages a tree-based neural network to encode these paths and generate more abstract representations [10]. Yin et al. employ neural networks to express source code edits [45]. However, these methods are only designed to represent the features in one programming language and do not capture the commonalities of multiple languages. And some methods are too abstract so that important information for effective retrieval is missing, such as low-level program syntax or token type [8], [22]. The program representation we use as the query is inspired by the work in [9] and [11]. In addition to AST, we consider features of concrete syntax trees (CST) and text to enrich the information for cross-language search. And we train a query transformation model on Big Code to transform the features between different languages.

Cross-language code clone detection. This line of research aims at identifying duplicates of a given piece of code. However, this line of research has so far only focused on languages with similar intermediate representation, such as the .NET language family [6], [24]. Others only calculate similarity on the textual level [13], [14] same as most code search methods. They simply treat programs as plain text and make certain assumptions that limit their usage in practice, i.e., they try to identify different revisions of the same piece of code. As our goal is to find program translations, our requirements go beyond the state-of-the-art in clone detection. Nevertheless, we still compare our program representation to plain text representation used in code clone methods [13], [14] in our benchmarks showing its superiority.

Cross-language text retrieval. There is a body of work on cross-language retrieval for natural language [21], [36], [42]. They take plain text as input and generate feature representations based on natural language syntax. However, code has its unique properties. Not only the code syntax can differ across languages, but the text in code does not exactly follow the same vocabulary and writing formats as in natural language. Therefore, directly using these methods on code search can lead to incorrect results or failures. We also evaluate using natural language methods in our experiment by processing code as plain text using bag-of-words and word2vec.

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

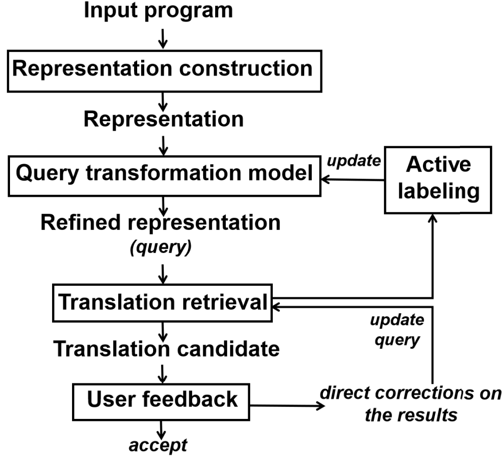

Fig. 1: BIGPT overview

III. System Overview

We propose BIGPT, an interactive cross-language code retrieval system that assists program translation by reusing Big Code. Given a piece of source program P_s written in language L_s, a selected target language L_t, and a large program repository $D_p = \{P_1, P_2, ..., P_n\}$, the goal is to find the best possible translation P_t of P_s in L_t from D_p. The essential problem is to design an effective program feature representation that generalizes to many languages, can be updated through user feedback, and enables efficient retrieval in the scale of big code.

The workflow of BIGPT is shown in Figure 1. BIGPT first constructs a feature representation for input programs to form the query (Section IV-A). Since the target is to identify a similar program in the target language, BIGPT then applies a query transformation model (QTM) to transform this representation into an estimated feature representation of the translation (Section IV). QTM is trained in an unsupervised manner on Big Code but can also be updated dynamically through active learning. This new representation will be used as a query to retrieve potential translations from the database. For efficient retrieval (Section V), BIGPT leverages an index structure that captures key feature elements and is constructed in the offline phase. As an interactive system, BIGPT allows the user to give feedback on the retrieved translation (Section VI). The user can either accept the result or make some corrections. Based on our structured and informative feature representation, BIGPT can easily and quickly adapt the query based on local user corrections. Note that the user is not necessarily correcting the whole program but only some local spots that they deem wrong. With this partial correction, BIGPT may identify a more appropriate translation candidate that can be accepted by the user in the second retrieval attempt.

IV. Query Construction

As the user only inputs the raw source code, to retrieve a relevant translation for it from a large code database, BIGPT needs to automatically construct an effective cross-language query. In this section, we will first introduce the fundamental program feature representation, then discuss how our query transformation model (QTM) generates the search query, and how BIGPT further optimizes the model with active learning.

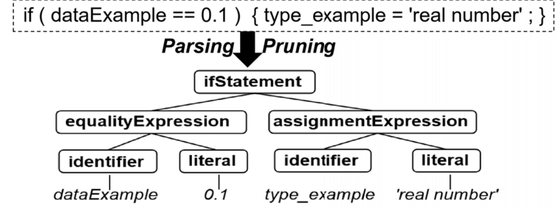

Fig. 2: Simplified syntax tree of a JavaScript program

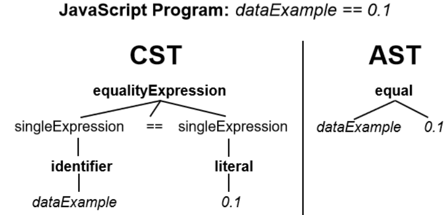

Fig. 3: CST and AST of a fragment of the program in Figure 2

A. Program Representation

As already suggested by the recent cross-language code retrieval system RPT [11], BIGPT takes both structural features and textual features as well as their dependencies into consideration to capture special aspects of programming language syntax and program semantics expressed in the text.

Structural Features. To capture the structural features of a program, we resort to a representation based on the syntax tree of a program. Each program can be represented by its syntax tree where each tree node denotes a code construct. The syntax tree depicts the structural dependencies of the code constructs. One could also use control flow graph (CFG) that captures the dependency between code blocks and procedures to approximate the code behavior. However, our goal is to assist program translation for any granularity of a program. Code behavior is hard to measure when the code fragment is not an independently executable code block. Considering that constructing CFGs requires more complex analysis than syntax trees, we pick syntax trees as the basis of our representation to also capture the low-level syntactic structure within code blocks. The syntax tree can be either a concrete syntax tree (CST) or an abstract syntax tree (AST) [9], [10], [12]. A CST depicts nodes with complete structural information, such as all the tokens in the code. As such the CST is highly language-dependent. The AST on the other hand is more abstract and misses information, such as the intermediate syntax and the type of each token.

Therefore, the CST is quite verbose and the AST may lose informative syntax and does not generalize to multiple languages. To develop a compromise solution that keeps the best of both worlds, and as previously suggested [11], we fall back on the low-level CST as a basis and take the philosophy

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Paths extracted from the tree in Figure 2

Root-Node	Leaf-Nodes	Path-Type
ifStatement	identifier, identifier	p_1
	identifier, literal	p_2
	literal, identifier	
	literal, literal	p_3
equalityExpression	identifier, literal	p_4
assignmentExpression	identifier, literal	p_5

of AST as an inspiration. Specifically, we first take CST of the program as the base structure. Then we simplify the CST by only removing semantically repetitive nodes so that redundant information can be removed and necessary information can be retained. Figure 2 shows the generated syntax tree of a JavaScript program. We use the same approach as proposed for RPT [11]. Figure 3 shows the original CST and AST of a fragment of this program. We can see that the new syntax tree is more succinct than CST and more informative than AST. However, this syntax tree is still complicated for representing the code features and retrieving the translation. And because of different control flow elements in different programming languages, it is unlikely to find programs that share the exact same syntax tree. Therefore, imitating the idea of AST, we further abstract the tree. First, we simplify the representation and transform the two-dimensional tree structure into a set of one-dimensional paths that connect the program elements inside the tree. BIGPT extracts *abstract* paths as follows: for each pair of leaf-nodes in the CST, BIGPT keeps the nodes themselves, their values, and the root-node of the statement and drops all other intermediate nodes on this path. The root-node is the summary of the whole path and the leaf-nodes directly indicate the content on this path. These three nodes enclose the most critical information on a path. Moreover, we extract all paths between two leaf-nodes from the tree to represent the features. In this way, all the intermediate nodes have the chance to be the root-nodes, which facilitates to capture more complete structural information. The paths extracted from the trees in Figure 2 are shown in Table. Then we classify these paths into different types and replace these paths with their path-type as shown in Table. In this way, we can further generalize and simplifies the features. Using the same method in RPT [11], BIGPT considers two paths are the same type if their root-nodes and leaf-nodes are the same or have the same meaning, such as `ifStatement` and `if_stmt`. Also, as the writing habit of programmers and the coding conventions for a programming language might differ, BIGPT ignores the order of left and right leaf-nodes, such as p_2. Thus, the structural feature of a program can be succinctly represented by a set of path-types $p_1, p_2, ..., p_j$ extracted from its syntax tree.

Textual Features. In contrast to existing work [13], [14] that suggest to extract all the text from a program and do not consider any context from the program structure, BIGPT considers textual features only in strong dependency with the structural features and leverages context from the structure [11]. To do so, BIGPT only processes text that appears in the extracted paths and marks the path-type where they belong. Because the text appears on or connects to those removed semantically repetitive nodes only brings in redundant information. And considering the dependency can encode the text with the features of the programming languages, not only the natural languages. This methodology can also simplify the features and improve system efficiency.

Similar to [11], first uses word tokenization and lemmatization to tokenize and stem all the words in the text. In addition, our tokenization process also considers camel case, spaces, and underlines to accommodate code-specific language. However, it does not remove and tokenize numeric values, such as hard-coded floating points and integers, as they might be integral to the purpose of a program. Then BIGPT vectorizes these generated tokens based on the Bag-of-words model (BoW). One could also resort to more sophisticated and complex embeddings, such as word2vec (W2V) [30] and BERT [17]. However, in programming languages, the structural features are more important than the textual ones and word order can be ignored to accommodate different programming styles. Besides, we only compare text for every single path-type, significantly reducing the number of words for each similarity calculation. BoW is sufficient for this process. In our experiment, W2V does not show worthwhile improvements but rather introduces extra training time for building the language model. To avoid repeated computation for every new input and to improve the efficiency, BIGPT precalculates the BoW model in the offline phase. To build the BoW model, we need to prepare a vocabulary of unique words in each program from the repository. As we consider the dependency with structural features, two textually identical tokens from different types of paths are regarded as different tokens. To this end, we count all the text tokens $t_1, t_2, ..., t_k$ of each program and store the results during the offline phase. In the online phase, BIGPT only needs to run word statistics on the input program and build the BoW model.

Feature Representation. Unlike RPT that uses a list of path-types and text collections as the final representation [11], we construct a numerical feature vector to represent the program. The final feature representation is thus more compendious as one single vector, which can be directly used to further train a learning model. Our final representation is constructed as follows: we regard the tokens (textual) together with different types of paths (structural) as feature elements e of a program and generate a feature vector consisting of the feature element frequencies. Let f be the occurrence frequency of feature elements, then the final vectorized feature representation of a program will be $[f_{e_1}, f_{e_2}, ..., f_{e_n}] = [f_{p_1}, f_{p_2}, ..., f_{p_{n_p}}, f_{t_1}, f_{t_2}, ..., f_{t_{n_t}}]$. In our experiments, the number of different path-types is about 5,000 on average for each language pair. In the subsequent retrieval process, the potential translation in the target language can be identified by calculating the similarity of feature vectors.

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

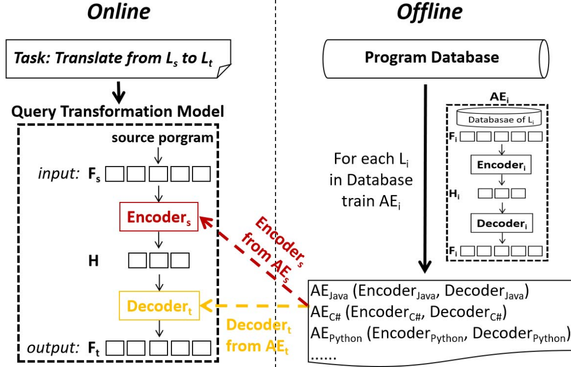

Fig. 4: Query transformation model (QTM)

B. Query Transformation Model

One could directly use the feature representation described in Section IV-A as a query to retrieve translation candidates. However, the feature vector will fail to accommodate some cross-language hurdles. For example, C# supports `goto` statements while its Java translation has to use so-called labelled statements with `break` or `continue` [1] instead of `goto`. In this case, BIGPT cannot directly use the features of the C# program to retrieve its Java translation. The result can be improved if the retrieval can be conducted based on the features of the translation. While the translation of a complete program is our original problem and hard to solve without large amounts of training data, we propose a QTM that solves a smaller problem. QTM transforms the original query, i.e., the feature vector of the input program, into an optimized query, which will be the estimated feature vector of the translation.

Model Description. As shown on the left part of Figure 4, in the online phase, BIGPT extracts the features as F_s of a program from source language L_s and feeds it into the QTM to translate the program features to features in language L_t. The QTM first selects previously trained auto-encoders (see the next paragraph for details) of L_s and L_t respectively, then extracts the encoder of the former and the decoder of the latter to compose a new encoder-decoder model. In this model, a one-layer encoder (red in Figure 4) maps the original feature vector to a low dimensional latent space and produces a shorter hidden vector H. Then H is reconstructed to the estimated translation feature vector F_t by a one-layer decoder (yellow in Figure 4). F_t will be used as a query to retrieve potential translation in target language L_t.

Since there is no available training data for QTM, we leverage an unsupervised method based on - auto-encoders (AE) to train an encoder and a decoder. An AE is an encoder-decoder system that aims to reproduce its input. That is, it encodes the input to a hidden vector, then reconstructs the input from this hidden vector. Therefore, no extra label for the training data is needed. With this approach, BIGPT learns the weights of the encoders and decoders separately. As shown in the right part of Figure 4, in the offline phase, BIGPT trains a separate AE_i for each programming language L_i in the database on all programs that are written in L_i. Thus it obtains a pair of $Encoder_i$ and $Decoder_i$ for each programming language. For the actual translation task, we combine the appropriate encoder and decoder depending on the source and target language of a translation task. In Figure 4, the QTM selects the encoder $Encoder_s$ of the source language L_s from AE_s to transform F_s into the hidden layer representation. And it picks the trained decoder $Decoder_t$ of the target language L_t from AE_t to estimate the F_t. This way, we can build a pre-trained model with an encoder that learns significant information from the feature vector of the input program and a decoder that can generate features of its translation.

Active Learning Mode. To increase the accuracy of QTM, we also provide an active learning mode to enable the user to fine-tune the model. For each pair of languages, it is possible to train the corresponding QTM via active learning. As shown in Figure 1, during each translation retrieval, the most useful input programs in the source language are selected with a sampling strategy. Then BIGPT will retrieve its translation in the target language. The user either accepts the retrieved result or annotates the correct translation herself. Then with this user-approved correct translation as the label of the input, we can further train the QTM and update the weights. To choose the appropriate input program, we propose an aggregation of four sampling strategies that capture the informativeness of a program as follows:

- **Coverage sampling** picks the programs that cover a wider range of different feature elements, which may reveal more information. We consider programs that cover more than 50% of the total amount of all feature elements as programs with high coverage. In a database, if the average amount of feature elements contained in a program A is λ and the program contains more than $\lambda/2$ different feature elements, it is a qualified sample.
- **Rarity sampling** considers programs with rare feature elements, i.e. programs that contain features that appear in at most ϵ% of the program database. For example, if feature element e_1 from program A appears in x% ($x < \epsilon$) of the database programs, A is a qualified sample.
- **Uncertainty sampling** picks retrieved programs with low certainty, i.e., lower similarity score than 75%. For example, if program B is the top retrieved translation of program A, but their similarity score is 50%, which is lower than 75%, program A is a qualified sample.
- **Random sampling** randomly selects a program [46].

BIGPT employs the query-by-committee method to aggregate the results of the four sampling methods [15]. With this approach, we make sure to have incorporated a diverse set of characteristics that might be relevant for sampling. The final decision is made by selecting program data where the largest disagreement occurs among those sampling strategies.The level of disagreement of a program x is measured by vote entropy VE [15]:

$$VE(x) = -\frac{V(x)}{N_s}\log\frac{V(x)}{N_s} - \frac{N_s - V(x)}{N_s}\log\frac{N_s - V(x)}{N_s} \quad (1)$$

$V(x)$ is the number of sampling strategies that select/vote x

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

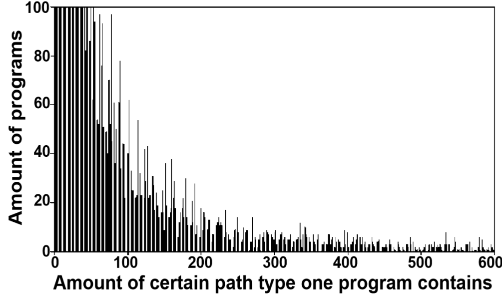

Fig. 5: Frequency histogram of path-types in one program

as a valuable sample. N_s is the total number of sampling strategies, which equals 4 in our case. Programs with higher vote entropy are returned as samples.

V. Translation Retrieval

The output of the QTM is an approximate representation of the translation. BIGPT uses this output as a query to retrieve the candidate with the highest feature similarity. To avoid a full scan while retrieving the most relevant translations, we use a path-type-aware index and an efficient retrieval mechanism.

Index Structure. In the offline phase, BIGPT constructs representations as described in Section IV-A for each program from the code database and stores them inside a feature database. To avoid a brute-force similarity computation, we need an index structure. For two programs to be similar, they have to share similar structure features, which are captured through common path-types as described in Section IV-A. So we need to first find all the programs that share at least one path-type with the source program. A naive approach is to index each path-type as the key. However, there are millions of programs inside the database, and there are only about 5,000 types of paths on average for each language pair, which makes this index structure highly sparse and ineffective. Inspired by [11], we design our index structure to also harbor the frequency of each path-type inside a program as many programs share the same path-type but differ in the frequency of such path-types. Since the source program and its translation candidate may not always share the amount of the same path-type, it would be too strict to have an index entry for every combination of path-type and frequency. Instead, we divide the frequency of path-types into multiple buckets and use the bucket intervals as indexes. We observed the frequency of each path-type in each program roughly obeys exponential distribution as shown in Figure 5. To ensure the size of each bucket is equal, we fix a bucket size and create as many buckets as are needed. Then, we sort the programs based on the frequency of the corresponding path-type and add them gradually to the sorted fix-sized buckets. In our experiments, a bucket size of 200 already shows a high-performance gain.

Example 1: p_1 occurs $[0, 1)$ times in 6 programs, $[1, 2)$ times in 3 programs, $[2, 3)$ times in 2 programs, and $[3, 4)$ times in one program. If we evenly divide the interval, we will have 9 programs in $[0, 2)$ and 3 programs in $[2, 4)$. But if we divide the interval into $[0, 1)$ and $[1, 4)$ based on the frequency distribution, each interval contains 6 programs. This interval leads to a more balanced index structure.

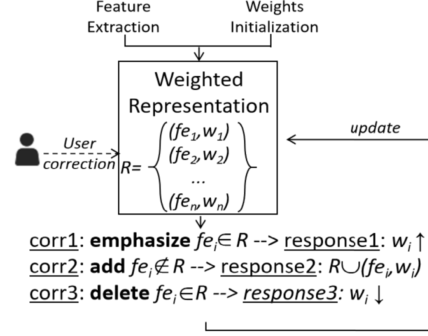

Fig. 6: User feedback mechanism

Efficient Retrieval. For each pair of query program and candidate program, BIGPT calculates the weighted sum of structural similarity and textual similarity to measure the overall similarity. To make the retrieval process as efficient as possible, we first use the index that filters all programs that do not contain a similar amount of the same path-types. The number of candidates for the similarity calculation is typically still quite high. Thus, BIGPT consecutively calculates the independent similarity components - structural similarity and textual similarity and drops candidates that fail to meet a minimum threshold concerning any of the two. The thresholds for both components are chosen based on the inflection points of each score distribution, respectively. Similar to RPT [11], BIGPT first calculates the structural similarity to each candidate, because the syntax structure of a program is more discriminative than textual features. This will significantly reduce the number of irrelevant programs and avoid the unnecessary calculation of textual similarity with them. For the remaining candidates, a textual similarity filter is employed which uses a weighted Jaccard index that accommodates the relevance of common textual features within the same path-types. For all the final remaining programs, the weighted sum of both previously calculated similarities will be generated to obtain the final similarity score.

VI. Query Adaption with User Feedback

It can happen that the desired translation is not among the top-k retrieved results. In this case, BIGPT can change the query based on the user's feedback. Existing interactive code retrieval methods ask the user to give feedback on preset metrics and questions as discussed in related work [34], [40], [41]. As each feature element in our query directly maps to a code fragment, BIGPT can directly pass the user's corrections on the code to adapt the query. The simplest form of using the user corrections is to just update the feature vector of the corrected program. However, we can also make use of the fact that user corrections lead to manually curated features. To reflect this in our feature representation, we extend it with a weighting scheme. As shown in Figure 6, we obtain a new

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

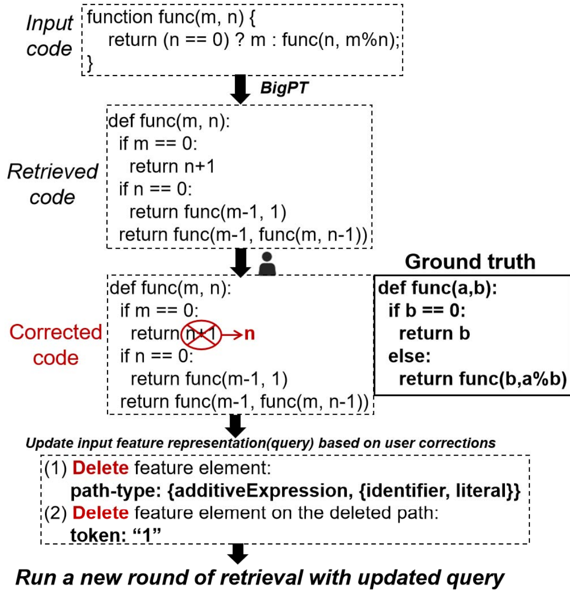

Fig. 7: An example of user feedback

feature representation R, where each element consists of a feature element f_{e_i} and its weight w_i. The initial weights are uniform. After the user makes one or several corrections to the result, BIGPT featurizes each correction the same way and compares it with the original code to generate the weights. We classify corrections into three categories and the weights are tuned accordingly:

- **Emphasize.** If the correction increases the number of a feature element $f_{e_i} \in R$, its weight will be increased.
- **Add.** If the correction adds a feature element $f_{e_i} \notin R$, f_{e_i} and its initial weight w_i will be added to R.
- **Delete.** If the correction decreases feature element $f_{e_i} \in R$, its weight w_i will also be decreased.

After a correction, the feature representation is updated and used for a new round of retrieval. When calculating the similarity between query and candidates in the database, BIGPT prefers the candidate that has a higher similarity in higher weighted features.

Example 2: Figure 7 is an example of the user feedback module. The input is a greatest common divisor function in JavaScript. The ground truth in Python is also shown in the figure. In the first round, BIGPT retrieved an Ackermann function as its Python translation. The possible reasons are the variable names are different in the ground truth and the weight of each feature element is not assigned properly. After user corrects the first wrong line (change `return n+1` to `return n`), BIGPT constructs feature representation for the corrected code. Then BIGPT compares it with the feature representation of the input code and summarizes user's corrections. Based on this, BIGPT updates the query: in Figure 7, the corrections are deleting the path-type {`additiveExpression`, {`identifier, literal`}} and token `1` on this path-type. As a result, these two feature elements will be removed and the weight of other feature elements will be increased accordingly. Finally, with the updated query, BIGPT will run a new round of retrieval. Without more precise features, the ground truth will be more likely to be retrieved.

VII. EXPERIMENTS

To show the feasibility of our BIGPT and evaluate its effectiveness and efficiency in assisting program translation, we conducted a series of **experiments**:

A. We compare different variations of BIGPT with existing work from program translation and code search;
B. We evaluate BIGPT on more languages;
C. We discuss the influence of user interaction;
D. We evaluate the scalability and efficiency of BIGPT.

All experiments have been carried out on a PC with an Intel Xeon E5-2650 v2 2.60GHz CPU and an NVIDIA Tesla K40m GPU.

Datasets. We have two different types of datasets:

1) Dataset for training the auto-encoders in QTM, Word2vec, and Code2vec:
   - **Public Git Archive** (PGA). We use this database with more than 260,000 bookmarked repositories [5] to train the AEs of QTM. We cleaned the dataset by gradually removing duplicates at files, and files that cannot be successfully parsed due to format, errors, version compatibility. We then collected data in four popular languages (JavaScript, Python, Java, C++). Finally, we obtain a dataset with a size of 260GB. We split all the files into methods or functions.
2) Datasets for evaluation: Datasets with ground truth are generally scarce. We run our experiments on two small parallel datasets and one larger unlabeled dataset:
   - **Java-C#** used in **experiment A, C, D**. It was used in previous studies [11], [12], [31], [32]. We use the same dump that was used for the cross-language retrieval engine RPT [11], which contains 39,797 matched methods.
   - **GeekforGeeks** used in **experiment B**. It was used in TransCoder [39], which gathered and aligned 698 coding problems and their solutions in Java, Python, and C++.
   - **PGAS** used in **experiment D**. To save experiment equipment and time, under the premise of ensuring the validity of the experiment and the reproducibility of the data, we randomly pick 1% files from PGA to obtain a dataset including 2,023,546 methods/functions. As this dataset has no labels, we manually judge the correctness.

Effectiveness metric. We use *program accuracy* (PA) as proposed by prior work [12], [32]. PA is the percentage of the predicted translation that is the same as the ground truth. Note that, PA is an underestimation because it does not account for programs that only differ in writing habits and style. We use this standard to manually judge the correctness for the dataset without ground truth. For our retrieval-based method, we consider the top-1 retrieved result as the "predicted translation" and report the percentages of search where the correct translation was the top-ranked result. Note that PA is an underestimation of *computational accuracy* which evaluates

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

whether the translation generates the same outputs as the source program when given the same inputs [39].

A. Comparison with Baselines

We compare BIGPT with one rule-based tool (**J2C#** [3]) and four data-driven program translation baselines (**1pSMT** [31], **mppSMT** [32], **Tree2tree** [12], **TransCoder** [39]), which use a translation model to generate the results. The supervised baselines use 90% matched method pairs as training data to predict the translations for the rest of the programs. We used the openly available implementation of **Tree2tree**. For TransCoder, we follow their method to pre-train the cross-language model on the Public Git Archive dataset (30GB of Java and C# data). For the program translation baselines 1pSMT and mmpSMT, we report the results from their work on the same dataset as their code and configurations are not available. Finally, we report the results of two mono-language code search systems (**Sourcerer** [26], **CodeHow** [27]) and one cross-language system (**RPT** [11]).

We generate different versions of BIGPT with variations in feature representation and interaction:

- **Representations**: We analyze $\text{BIGPT}_{\text{WORD2VEC}}$ and $\text{BIGPT}_{\text{CODE2VEC}}$ as two feature representations variations of BIGPT that retrieving translation based on **Word2vec** [29] and **Code2vec** [10], respectively.
- **Interactive versions**: We discuss three different interactive versions of BIGPT. All of which use the same feature representation and the QTM module. $\text{BIGPT}_{\text{no_AL}}$ uses the original QTM that has not been improved by active learning. BIGPT is the default setting of our system. $\text{BIGPT}_{\text{+FB}}$ is the full-fledged interactive system when user feedback is available as described in Table II.

Table II shows the results and the degree of supervision. We observe that the full-fledged BIGPT with at most one user correction per task and optimized QTM outperforms all the baselines. The improvement in program accuracy ranges from 19.5% to 65.5%. As expected, the mono-language code search baselines perform poorly because they are designed for retrieval with more accurate and detailed input than raw code in another language. The cross-language code search system RPT performs better than other baselines, which shows the feasibility of the translation retrieval methodology. The results of partial components of BIGPT are encouraging. $\text{BIGPT}_{\text{no_AL}}$, which does not leverage any supervision, outperforms the Tree2tree, which shows the effectiveness of our program feature representation and QTM module. Our feature representation equipped with QTM successfully improves the result by 7.5% compared to RPT showing that generating features in the target language is more promising than using features of the source language. In our default system (BIGPT), the QTM is further trained by active learning, the accuracy can increase by 8.5%. The table also shows that the Word2vec and Code2vec variants of BIGPT cannot perform better. Word2vec is designed for natural languages so that it can not capture the special features of programming languages. Although Code2vec is designed specifically to represent code, their model can only be trained within the same programming language, which makes it less suitable for cross-language similarity comparisons.

We further explored the supervision impact on BIGPT. In this experiment, we let the user give at most one correction to each retrieved task. With such limited user feedback, $\text{BIGPT}_{\text{+FB}}$ can still slightly improves on BIGPT. Compared to the fully supervised methods 1pSMT, mmpSMT, and Tree2tree, BIGPT and $\text{BIGPT}_{\text{+FB}}$ leverage very limited human supervision to achieve better results. In the first retrieval round, where the user does not make corrections to any wrong results, BIGPT achieves 87.1% accuracy with only 80 labels for QTM. Also, the reproduced weakly-supervised approach TransCoder does not achieve better results than BIGPT. We observed that their model often generates invalid translations with regard to grammar. For example, it often mistakes the input type of a function. This phenomenon is also acknowledged in their own paper and can be attributed to the fact that only textual features have been used. BIGPT avoids this problem by reusing existing code.

B. Evaluation on Multiple Programming Languages

We further evaluate BIGPT on more languages. We compare BIGPT and the state-of-the-art methods TransCoder and RPT on the GeeksforGeeks benchmark that contains ground truth for Java, Python, and C++. Table III shows that the accuracy of BIGPT is significantly higher than TransCoder on their own datasets as reported in their own paper. Note that, for TransCoder the authors report *computational accuracy* instead of program accuracy. As the dataset has ground truth, the reported *program accuracy* for our method, which is an under-estimation of the possible *computational accuracy* for BIGPT. We infer that TransCoder has two drawbacks: (1) TransCoder outputs machine-generated translations while BIGPT directly retrieves existing programs as translations, which makes BIGPT always output syntactically correct programs. (2) TransCoder generally considers programming languages as plain text and aims to generate semantically similar programs. The black-box neural network model may neglect some non-trivial syntactical features of programming languages. BIGPT also outperforms the cross-language retrieval system RPT with over 10%. The reason is that BIGPT uses a query that represents the features of the translation rather than the input program. Also, the user-interaction mechanism can improve the performance in most cases. This experiment further shows that BIGPT is generalizable for multiple languages including dynamic languages, such as Python.

C. Influence of User Interaction

In Table II, we showed the influence of a single user correction on the result. We further investigate the required number of user corrections to retrieve the correct translations. We simulate the user correction with the ground truth in our parallel dataset and for each returned result we fix the first differing line between true result and returned result.

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Comparison of different methods on PA and supervision extent (Java-C#)

Genre	Method	Description	PA	Supervision Extent
Rule-based	J2C#	manually defined translation rules	16.8%	fully supervised
Data-driven program translation	1pSMT	Phrase-based SMT	24.1%	fully supervised
	mmpSMT	multi-phase phrase-based SMT	41.7%	
	Tree2tree	tree-to-tree neural networks	70.1%	
	TransCoder	weakly-supervised neural translation	49.9%	weakly supervised
Code search system	Sourcerer	Lucene-based code search, free-text queries	13.5%	no labels (directly retrieve translation with input)
	CodeHow	free-text queries	13.5%	
	RPT	cross-language code search	71.1%	
Variations of BIGPT	BIGPT$_{\text{WORD2VEC}}$	word2vec as queries	67.7%	no labels (directly retrieve translation with input)
	BIGPT$_{\text{CODE2VEC}}$	code2vec as queries	63.4%	
	BIGPT$_{\text{no_AL}}$	QTM without active learning	78.6%	
	BIGPT	**the default system**	**87.1**%	80 labels for QTM
	BIGPT$_{\text{+FB}}$	user feedback is available	***89.6%***	80 labels for QTM + at most 1 correction per task

TABLE III: Comparison of accuracy on GeeksforGeeks

	C++ -Java	C++ -Python	Java -C++	Java -Python	Python -C++	Python -Java
TransCoder	60.9%	44.5%	80.9%	35.0%	32.2%	24.7%
RPT	69.2%	65.3%	70.9%	59.3%	55.4%	54.2%
BIGPT$_{\text{no_AL}}$	76.6%	74.2%	78.1%	68.1%	59.2%	59.6%
BIGPT	87.2%	79.5%	84.8%	72.5%	66.2%	68.8%
BIGPT$_{\text{+FB}}$	84.8%	83.0%	90.5%	77.5%	67.8%	68.1%

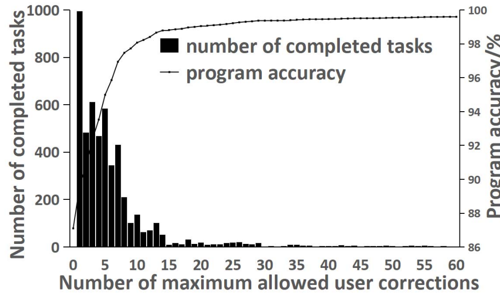

Fig. 8: Required amount of user feedback

Figure 8 shows the number of required user corrections to obtain the correct result for all 5,134 failed translation tasks from the first retrieval round and the improvement in the overall accuracy. We observe that in most cases, BIGPT only requires a single user correction to successfully complete the translation task. About 98% of the failed retrieval tasks can succeed after 10 user corrections. Considering the average length of an input program is 168 lines, we can conclude that with a limited number of user corrections the accuracy of BIGPT can be significantly improved. Note that, we might even achieve better results if we do not restrict the users to fix the first difference each time. A real user might fix more significant errors that lead to faster convergence.

D. Evaluation of scalability and efficiency

To evaluate BIGPT on a larger dataset with multiple languages, we run it on the PGAS dataset. As we have to manually judge the correctness, we carry out a sampling inspection to make it feasible. We randomly pick 270 programs and BIGPT retrieves the best possible translation from the dataset for each of these programs. As we spent up to 15 minutes on each result to make the judgement as accurate as possible, we had to limit the labeling process due to time and human resources.To show the advantage of BIGPT, we compare its results to BIGPT$_{\text{WORD2VEC}}$ and BIGPT$_{\text{CODE2VEC}}$.

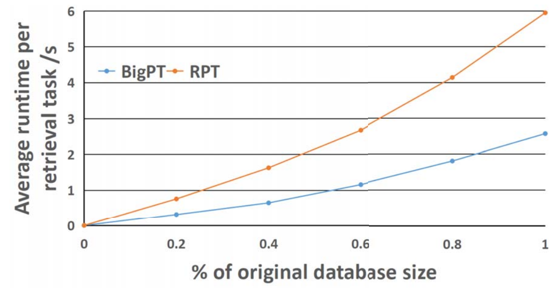

Fig. 9: Relationship between the efficiency and database scale

In Tables IV, we observe that BIGPT can successfully retrieve the correct translation for 58.8% programs among four languages. Considering BIGPT is positioned as a translation assistant system, the accuracy is considerable. Translation tasks that cannot be fully automated can still be assisted by the results returned by BIGPT. This result shows that it is feasible to scale BIGPT to more data volume. Compared to the other program representations, our novel representation performs significantly better. W2V does not contribute much to the results and introduces extra model training time compared to the simple BoW model. In BIGPT, we retain BoW to trade-off response time for a slight decrease of accuracy. Code2vec also considers the structure of programming languages. However, their model can only be trained within the same language, which limits its performance in the cross-language setting.

We also compare the average runtime of each translation retrieval task between RPT and BIGPT. Table V shows that both RPT and BIGPT can complete the translation task on average in **0.2s** on the small Java-C# dataset. On the larger PGAS dataset, BIGPT can retrieve the translation within **2.57s**. We further investigate how the runtime increases with the scale of the database. We randomly pick 20%, 40%, 60%, and 80% of the PGAS dataset and run the retrieval process on these subsets. For each subset, we retrieve translation candidates

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Comparison of program accuracy on the PGAS dataset

Source language	JS			Python			Java			C++		
Target language	C++	Python	Java	JS	C++	Java	JS	Python	C++	JS	Python	Java
BIGPT$_{\text{CODE2VEC}}$	51.0%	32.0%	40.2%	41.9%	43.4%	43.1	50.0%	38.6%	63.4%	58.9%	47.5%	64.4%
BIGPT$_{\text{WORD2VEC}}$	61.4%	**53.9%**	**60.2%**	47.9%	**54.7%**	**48.3%**	59.1%	57.1%	67.7%	64.0%	**63.6%**	66.9%
BIGPT	**61.4%**	52.3%	59.8%	**48.4%**	54.2%	47.9%	**60.6%**	**57.5%**	**69.1%**	**64.0%**	63.5%	**67.0%**

TABLE V: Efficiency of BIGPT (runtime per retrieval)

Method	Java-C# (39,797 matched methods)	PGAS (2,023,546 methods/functions)
RPT	0.20s	5.95s
BigPT	0.20s	**2.57s**

for 1,000 randomly picked input programs and calculate the average runtime per retrieval task. Including the results on the whole dataset, all the results are shown in Figure 9. BIGPT is here significantly faster than RPT because it filters more irrelevant programs due to its advanced QTM. And the runtime is slowly growing with the database scale with an approximate exponential pattern. In addition, we measured the efficiency during user interaction. The average response time of BIGPT to each intermediate user feedback is **9.4ms**, which shows that BIGPT can respond to corrections in real-time.

Scope of application. Theoretically, BIGPT can be applied to all static and dynamic imperative languages. According to our experiments, BIGPT is more effective in finding translation candidates for grammatically similar programming languages, such as C++ and Java. Furthermore, when the target language is a low-level programming language, such as C++, the accuracy is generally higher than Python, which is a high-level dynamic programming language.

Failed cases. There are cases where our approach fails:

- Programs with special structures that do not exist in the target language. For example, deterministic destruction and pointer arithmetic in C++ cannot be translated to Java.
- Programs with APIs that do not exist in the target language cannot be translated.
- Short programs, i.e., fewer than 3 lines, are highly ambiguous and lead to more candidates with similar scores.

VIII. CONCLUSION

We presented a novel interactive cross-language code retrieval system that can assist program translation by reusing Big Code. We propose a novel cross-language program representation with a QTM that can learn a succinct but informative feature vector to retrieve the possible translation of an input raw program. Our querying and retrieving mechanism makes the system scalable and efficient on Big Code. Further, this succinct representation can be easily adapted to user corrections for interactive retrieval improvements. Our experiments show that BIGPT outperforms existing solutions and requires no parallel training dataset and additional user input.

Limitations. Although BIGPT's performance is promising, it still has some limitations. A translation task cannot be assisted if there is no potential translation for any subset of the input code inside the database. Besides, the quality of the data can affect the results. For large program inputs, a postprocessing step for verification might be necessary. BIGPT cannot work well on functional programming languages like Haskell and Erlang because they typically do not contain control flow elements.

Future work. First and foremost, there is a potential research direction on creating more convenient and intuitive user interfaces that allow users to make relevant corrections to translation suggestions and enable the system to converge faster to the desired result. Second, it might also be interesting to look into other forms of user interaction that do not require the user to provide corrections in the target language. Third, there is still room to explore post-processing of cross-language code retrieval and aggregation of retrieval results to obtain translations for larger programs. Finally, developing a user error model that can imitate user errors in code retrieval seems like a promising research direction to better assess the limitations of retrieval-assisted programming.

ACKNOWLEDGMENT

This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

REFERENCES

[1] Branching statements. website. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html [Online; accessed 20-April-2021].

[2] codase. website. http://www.Codase.com [Online; accessed 20-April-2021].

[3] Java2csharp. website. https://sourceforge.net/projects/j2cstranslator/ [Online; accessed 20-April-2021].

[4] krugle. website. http://www.krugle.com [Online; accessed 20-April-2021].

[5] Public git archive. website. https://github.com/src-d/datasets/tree/master/PublicGitArchive [Online; accessed 20-April-2021].

[6] Farouq Al-Omari, Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Detecting clones across microsoft .net programming languages. In *19th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2012.

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018.

[8] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning (ICML)*. JMLR.org, 2016.

[9] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018.

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019.

[11] Binger Chen and Ziawasch Abedjan. Rpt: Effective and efficient retrieval of program translations from big code. In *43nd International Conference on Software Engineering, Companion Volume (ICSE)*. ACM, 2021.

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.

[12] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *6th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018.

[13] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. Mining revision histories to detect cross-language clones without intermediates. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2016.

[14] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. *CLCMiner*: Detecting cross-language clones without intermediates. *IEICE Trans. Inf. Syst.*, 100-D(2):273–284, 2017.

[15] Ido Dagan and Sean P. Engelson. Committee-based sampling for training probabilistic classifiers. In *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning (ICML)*. Morgan Kaufmann, 1995.

[16] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*. ACM, 2020.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. Association for Computational Linguistics, 2019.

[18] Timothy Dietrich, Jane Cleland-Huang, and Yonghee Shin. Learning effective query transformations for enhanced requirements trace retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013.

[19] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018.

[20] Zack Ives, Yi Zhang, Soonbo Han, and Nan Zheng. Dataset relationship management. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, 2019.

[21] Zhuoren Jiang, Yue Yin, Liangcai Gao, Yao Lu, and Xiaozhong Liu. Cross-language citation recommendation via hierarchical representation learning on heterogeneous graph. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR)*. ACM, 2018.

[22] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[23] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018.

[24] Nicholas A. Kraft, Brandon W. Bonds, and Randy K. Smith. Cross-language clone detection. In *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, 2008.

[25] Jing Li, Aixin Sun, Zhenchang Xing, and Lei Han. API caveat explorer - surfacing negative usages from practice: An api-oriented interactive exploratory search system for programmers. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR)*. ACM, 2018.

[26] Erik Linstead, Sushil Krishna Bajracharya, Trung Chi Ngo, Paul Rigor, Cristina Videira Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Min. Knowl. Discov.*, 18(2):300–336, 2009.

[27] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on API understanding and extended boolean model (E). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2015.

[28] Lee Martie, Thomas D. LaToza, and André van der Hoek. Code-exchange: Supporting reformulation of internet-scale code queries in context (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2015.

[29] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations (ICLR)*, 2013.

[30] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2013.

[31] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013.

[32] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code (T). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2015.

[33] Anh Tuan Nguyen, Zhaopeng Tu, and Tien N. Nguyen. Do contexts help in phrase-based, statistical source code migration? In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2016.

[34] Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. Query expansion based on crowd knowledge for code search. *IEEE Trans. Serv. Comput.*, 9(5):771–783, 2016.

[35] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020.

[36] Razieh Rahimi and Azadeh Shakery. Online learning to rank for cross-language information retrieval. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku (SIGIR)*. ACM, 2017.

[37] Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2015.

[38] Steven P. Reiss. Semantics-based code search. In *31st International Conference on Software Engineering (ICSE)*. IEEE, 2009.

[39] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[40] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. Active inductive logic programming for code search. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2019.

[41] Shaowei Wang, David Lo, and Lingxiao Jiang. Active code search: incorporating user feedback to improve code search relevance. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM, 2014.

[42] Jingfang Xu, Feifei Zhai, and Zhengshan Xue. Cross-lingual information retrieve in sogou search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 2017.

[43] Cong Yan and Yeye He. Synthesizing type-detection logic for rich semantic data types using open-source code. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 2018.

[44] Cong Yan and Yeye He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*. ACM, 2020.

[45] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. Learning to represent edits. In *7th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2019.

[46] Xingquan Zhu, Peng Zhang, Xiaodong Lin, and Yong Shi. Active learning from data streams. In *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM)*. IEEE Computer Society, 2007.

Authorized licensed use limited to: Technische Universitaet Berlin. Downloaded on February 13,2024 at 14:21:32 UTC from IEEE Xplore. Restrictions apply.