Post-Incorporating Code Structural Knowledge into Pretrained Models via ICL for Code Translation

Yali Du†, Hui Sun†, and Ming Li, *Member, IEEE*,

***Abstract*—Code translation migrates codebases across programming languages. Recently, large language models (LLMs) have achieved significant advancements in software mining. However, handling the syntactic structure of source code remains a challenge. Classic syntax-aware methods depend on intricate model architectures and loss functions, rendering their integration into LLM training resource-intensive. This paper employs in-context learning (ICL), which directly integrates task exemplars into the input context, to post-incorporate code structural knowledge into pre-trained LLMs. We revisit exemplar selection in ICL from an information-theoretic perspective, proposing that list-wise selection based on information coverage is more precise and general objective than traditional methods based on combine similarity and diversity. To address the challenges of quantifying information coverage, we introduce a surrogate measure, Coverage of Abstract Syntax Tree (CAST), measuring maximum subtree coverage between ASTs of test source code and exemplars. Furthermore, we formulate the NP-hard CAST maximization for exemplar selection and prove that it is a standard submodular maximization problem. Therefore, we propose a greedy algorithm for CAST submodular maximization, which theoretically guarantees a $(1 - 1/e)$-approximate solution in polynomial time complexity. Our method is the first training-free and model-agnostic approach to post-incorporate code structural knowledge into existing LLMs at test time. Experimental results show that our method significantly improves LLMs performance in code translation and reveals two meaningful insights: 1) Code structural knowledge can be effectively post-incorporated into pre-trained LLMs during inference, despite being overlooked during training; 2) Scaling up model size or training data does not lead to the emergence of code structural knowledge, underscoring the necessity of explicitly considering code syntactic structure.**

***Index Terms*—Code Translation, Syntactic Validity, Large Language Model, In-context Learning.**

I. Introduction

CODE translation migrates codebases from one programming language to another, avoiding the time- and labor-intensive process of redeveloping software from scratch. Automated code translation significantly enhances coding productivity in various practical scenarios, such as: 1) modernizing legacy systems (e.g., COBOL or FORTRAN) with contemporary languages (e.g., Python or Java) to extend functionality; 2) ensuring cross-platform compatibility or using language-specific advantages, e.g., integrating high-performance C++ code with Python-based machine learning frameworks.

The authors are with the National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China, and also with the School of Artificial Intelligence, Nanjing University, Nanjing 210023, China. Email: {duyl, sunh, lim}@lamda.nju.edu.cn

† These authors contributed equally. Corresponding author: Ming Li.

Manuscript received xxx xx, 20xx.

Recently, large language models (LLMs) have demonstrated exceptional performance across various software mining tasks [1]–[7]. However, effectively handling the syntactic structure of source code remains a significant challenge. Unlike the grammatical structure of natural language, the syntactic structure of source code is stricter and more complex, as it directly reflects program behavior. Previous studies have shown that explicitly incorporating code syntactic knowledge can significantly enhance performance in software mining tasks [8]–[15]. However, traditional methods often depend on complex model architectures and customized loss functions, making their integration with LLMs both time-consuming and resource-intensive, even infeasible for proprietary models.

In-context learning (ICL) provides an effective and efficient solution of integrating new domain-specific knowledge into off-the-shelf LLMs without additional training by directly incorporating task exemplars into the input context [16]. For instance, in Python-to-C++ code translation, ICL enhances LLMs performance by incorporating a few paired exemplars (e.g., Python-to-C++ translation pairs) into the input prompt. This training-free approach offers flexibility across diverse tasks, avoids catastrophic forgetting, and ensures efficient resource utilization. Selecting appropriate contextual exemplars is critical for ICL. However, research on exemplar selection strategies for ICL in code translation is still limited.

Traditional ICL exemplar selection strategies often overlook code translation tasks and fail to consider the critical syntactic information inherent in source code [16]. These methods primarily rely on sample-wise similarity [17]–[19], ignoring compositional relationships among exemplars, which often leads to redundancy. Furthermore, syntax-aware similarity remains underexplored. Some approaches attempt to ensure diversity through clustering and additional regularization losses [20]–[24]. However, these methods often hinder the optimization of relevance and overlook compositional relationships in the syntactic structure of code. Moreover, balancing similarity and diversity poses a significant challenge.

This paper revisits exemplar selection for ICL from an information-theoretic perspective, emphasizing that information coverage serves as a more precise and general objective than the traditional similarity and diversity. However, directly quantifying information coverage in a combinatorial manner using information theory is challenging. To address this, we propose a surrogate measure for information coverage in code translation, based on Abstract Syntax Tree (AST), called the Coverage of AST (CAST). For a given test source code

and exemplar set, we extract ASTs of them while retaining only node type attributes and discarding value attributes. The exemplar ASTs are then used to cover subtrees of the test source code AST. CAST quantifies the proportion of nodes in the test source code AST included in the *maximum covered subtrees*. Maximizing CAST supports list-wise compositional exemplar selection, achieving "two birds with one stone": it explicitly considers syntactic structure knowledge in ICL and approximates the information coverage between the context and the test source code. However, two new challenges arise: 1) How to efficiently perform subtree covering across trees, and 2) How to solve CAST maximization for exemplar selection, as it is an NP-hard problem that cannot be optimally solved within polynomial time.

To address these challenges, we register a unique fingerprint to each AST subtree. Fingerprints for all subtrees in an AST are efficiently extracted via a single post-order traversal, and the exemplar database can be pre-processed efficiently. Next, we construct a co-occurrence matrix M between the test source code and candidate exemplars, where M_{ij} denotes whether the j-th AST subtree of the source code exists in the i-th candidate exemplar. Consequently, exemplar selection can be formulated as selecting rows from the co-occurrence matrix to maximize the ℓ_1-norm of the *OR* vector across selected rows. We prove this is a standard submodular maximization problem, which has been widely studied in subset selection area. Although this is a general NP-hard problem, we can employ a greedy algorithm that starts with an empty selection set and iteratively adds elements to maximize the marginal gain of CAST at each step until the size limit k is reached. This algorithm theoretically guarantees a $(1-1/e)$-approximate solution with polynomial time complexity, i.e., its performance is at least $(1-1/e)$ of the optimal solution.

To evaluate CAST, we integrate it as a plug-and-play module with state-of-the-art (SOTA) LLMs of varying scales during inference and compare it with traditional exemplar selection strategies based on similarity and diversity. Experimental results demonstrate that our method significantly improves performance over baseline methods and outperforms other exemplar selection strategies.

Our main contributions are summarized as follows:

1) **Revealing meaningful insights:** This paper presents two key findings: First, code syntactic knowledge, often overlooked during training, can be effectively post-incorporated into pre-trained LLMs at test time. Second, scaling model size or training data alone does not ensure the sufficient emergence of code syntactic knowledge, emphasizing the need to explicitly consider code syntactic structures into LLMs.
2) **New objective for ICL exemplar selection, CAST:** We propose the first training-free and LLM-agnostic approach to incorporate code syntactic knowledge into existing LLMs during inference using ICL. We revisit exemplar selection in ICL from an information-theoretic perspective, showing that information coverage is a more precise and general objective. We introduce a surrogate measure that incorporates syntactic structure knowledge into LLMs compositionally while approximating the quantify-challenging information coverage efficiently.
3) **Practical solution for CAST maximization:** Although CAST maximization in exemplar selection is NP-hard, we prove it is a submodular maximization problem and propose a greedy algorithm that theoretically guarantees a $(1-1/e)$-approximate solution with practical polynomial time complexity. Experimental results show that CAST significantly improves the code translation performance of LLMs across various scales.

II. Revisiting Exemplar Selection via Information Theory for ICL in Code Translation

A. Preliminaries of Code Translation and ICL

Code translation can be formulated as $\mathcal{P}_{\text{trans}} : X \mapsto Y$, where X represents the source code to be translated, Y represents the target code, and $\mathcal{P}_{\text{trans}}$ denotes a translation model, such as an LLM. For a given source code instance X_{test}, the translation is formally expressed as:

$$\widehat{Y}_{\text{test}} = \mathcal{P}_{\text{trans}}(\widehat{Y}_{\text{test}}|X_{\text{test}})\,. \tag{1}$$

In ICL, the database $\mathcal{D} = \{(X_i, Y_i)\}_{i=1}^{n}$ contains n source and target code pairs. The indices of the database are denoted by $\mathbb{N}_n = \{1, 2, \ldots, n\}$, and $S \subseteq \mathbb{N}_n$ represents a subset of these indices. Thus, ICL-based code translation with k exemplars as context is expressed as:

$$\widehat{Y}_{\text{test}} = \mathcal{P}_{\text{trans}}(\widehat{Y}_{\text{test}}|\underbrace{X_{S[1]}, Y_{S[1]}, \ldots, X_{S[k]}, Y_{S[k]}}_{\text{context of } k \text{ paired exemplars}}, X_{\text{test}})\,, \tag{2}$$

where $S[i]$ represents the index of the i-th selected exemplar.

Therefore, the primary challenge is to select an effective context for the input source code X_{test}. This requires finding a subset of indices S that meets the context size constraint $|S| = k$, defined as:

$$S^* = \arg\max_{S \subseteq \mathbb{N}_n, |S|=k} \text{Score}(S, X_{\text{test}})\,. \tag{3}$$

Traditional ICL exemplar selection relies on two main criteria: similarity and diversity [16]. Similarity-based methods select the top-k most similar exemplars [17], [25]–[28], which are simple and fast but often yield homogeneous exemplars and sub-optimal results. To address redundancy, some studies incorporate diversity [20]–[22], but these methods typically depend on instance-wise distance metrics and neglect the compositional relationships among exemplars. Moreover, current ICL exemplar selection methods often struggle to balance similarity and diversity, as the trade-off is task-sensitive, limiting their robustness and generalizability.

B. Revisiting Exemplar Selection with Information Theory

We revisit exemplar selection in ICL from an information-theoretic perspective. In code translation, information coverage refers to *the overlapping information between the source code of test sample and task exemplars*, quantifying the useful knowledge provided by exemplars for translating the current source code. Formally, information coverage is defined as:

$$\text{IC}(S, X_{\text{test}}) = H(X_{\text{test}}) - H(X_{\text{test}}|X_{S[1]}, \ldots, X_{S[k]})\,, \tag{4}$$

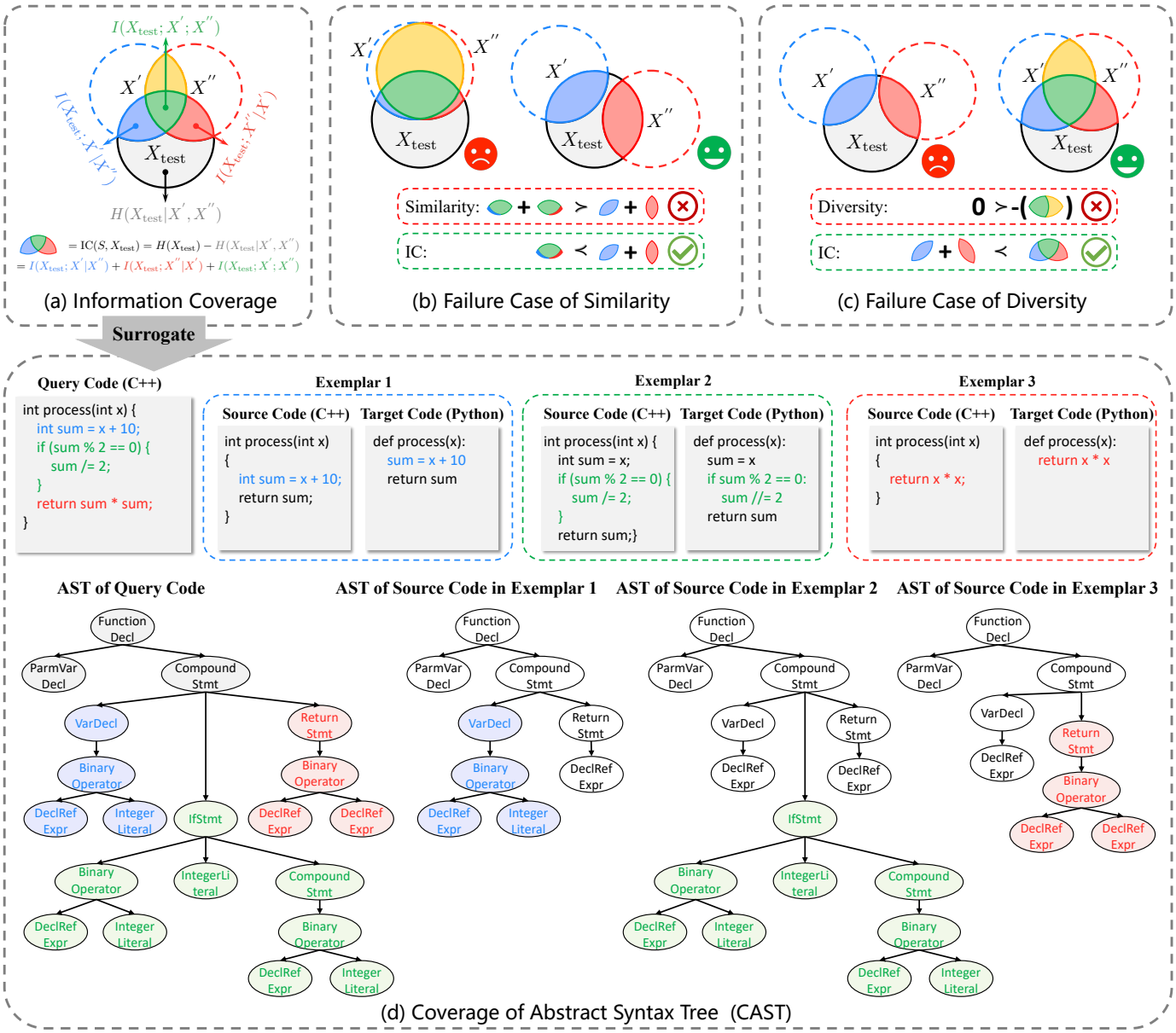


Fig. 1. Revisiting ICL exemplar selection from an information-theoretic perspective; CAST for code translation.

where $H()$ represents information entropy. For instance, as shown in Fig. 1 (a), if two exemplars ($k = 2$) with source codes $X^{'}$ and $X^{''}$ are selected for the test source code X_{test}, the information coverage is:

$$\begin{aligned}\text{IC}(S, X_{\text{test}}) &= H(X_{\text{test}}) - H(X_{\text{test}}|X^{'}, X^{''}) \\ &= I(X_{\text{test}}; X^{'}|X^{''}) + I(X_{\text{test}}; X^{''}|X^{'}) + I(X_{\text{test}}; X^{'}; X^{''}),\end{aligned} \tag{5}$$

where $I()$ denotes general mutual information, such as interaction information or conditional mutual information. Here, the required knowledge to translate X_{test} ($H(X_{\text{test}})$) is divided into four parts: $I(X_{\text{test}}; X^{'}|X^{''})$ and $I(X_{\text{test}}; X^{''}|X^{'})$ measure the individual contributions of $X^{'}$ and $X^{''}$, respectively. $I(X_{\text{test}}; X^{'}; X^{''})$ represents the common knowledge provided by both $X^{'}$ and $X^{''}$, and $H(X_{\text{test}}|X^{'}; X^{''})$ reflects the knowledge not provided by these exemplars.

Revisiting the traditional notions of similarity and diversity with the information theory, similarity can be interpreted as:

$$\text{Similarity}(S, X_{\text{test}}) \propto \sum_{i=1}^{n} I(X_{\text{test}}; X_{S[i]}), \tag{6}$$

which measures the total mutual information between the test sample X_{test} and each selected exemplar $X_{S[i]}$. On the other hand, pairwise diversity can be interpreted as:

$$\text{Diversity}(S, X_{\text{test}}) \propto -\sum_{i,j=1;i\neq j}^{n} I(X_{S[i]}; X_{S[j]}), \tag{7}$$

which accounts for the negative mutual information between selected exemplars, ensuring reduced redundancy among them.

The Venn diagram examples in Fig. 1 illustrate the failure cases of maximizing similarity or diversity individually. Focusing solely on similarity overemphasizes common knowledge

among exemplars (as shown in Fig. 1 (b)), resulting in redundant selections with minimal information gain. Conversely, optimizing diversity alone ignores the relevance between the test sample and selected exemplars and unjustly penalizes irrelevant mutual information $I(X^{'}; X^{''}|X_{\text{test}})$ (yellow part in Fig. 1 (c)). Some methods arbitrarily combine similarity and diversity as a weighted objective. To compare it with information coverage (Eq. (5)), consider selecting two exemplars, $X^{'}$ and $X^{''}$. The combined objective, weighted by λ, is given by:

$$\begin{aligned}&\text{Similarity}(S, X_{\text{test}}) + \lambda \cdot \text{Diversity}(S, X_{\text{test}})\\ &\quad\propto I(X_{\text{test}}; X^{'}) + I(X_{\text{test}}; X^{''}) - \lambda I(X^{'}; X^{''})\\ &\quad= I(X_{\text{test}}; X^{'}|X^{''}) + I(X_{\text{test}}; X^{''}|X^{'})\\ &\qquad+ (2-\lambda) I(X_{\text{test}}; X^{'}; X^{''}) - \lambda I(X^{'}; X^{''}|X_{\text{test}})\,.\end{aligned} \tag{8}$$

It is similar to Eq. (5), but there are two key differences: 1) It arbitrarily assigns different weights to the components of required knowledge. While setting $\lambda = 1$ ensures fairness, increasing the selection size k complicates partitioning and weighting, making the balance harder to maintain. 2) The term $I(X^{'}; X^{''}|X_{\text{test}})$ is unnecessary, as it represents shared knowledge of $X^{'}$ and $X^{''}$. Minimizing this term is redundant and can interfere with optimizing other valuable terms.

Consequently, information coverage serves as a more precise and general objective than similarity and diversity. However, directly computing it, as defined in Eq. (4), is challenging due to the complexity of estimating the conditional distribution $\mathcal{P}(X_{\text{test}}|X_{S[1]}, \ldots, X_{S[k]})$. To overcome this challenge, we propose a surrogate measure called CAST, which is computationally efficient and considers code syntactic knowledge. The details are elaborated in the next section (Sec. III).

III. Proposed Method

To enhance code translation in LLMs via ICL, we propose CAST, a surrogate measure for information coverage, and present a practical solution for NP-hard CAST maximization with exemplar set selection. In Sec. III-A, we define CAST, a list-wise Score function based on the code's AST. CAST accounts for both the relevance of exemplars to the source code and the compositional relationships among multiple selected exemplars. In Sec. III-B, we present a greedy algorithm for CAST maximization, proving that it guarantees a $(1-1/e)$-approximation for the NP-hard subset selection problem with polynomial time complexity. Finally, Sec. III-C provides an overview of the CAST framework.

A. Surrogate Scoring Function: CAST

We aim to integrate code structural knowledge into LLMs via ICL. As discussed in Sec. II, effective ICL exemplar selection relies on a well-designed Score function, with information coverage as a precise and general scoring objective. However, directly quantifying information coverage is challenging, and limited research has explored considering code structural knowledge in ICL. We propose CAST, a surrogate measure that approximates information coverage by quantifying the ratio of the *maximum subtrees covered by exemplars* in the test source code AST. Maximizing CAST is the first training-free and model-agnostic method for explicitly post-incorporating code structural knowledge into LLMs at test time.

Algorithm 1: $\mathrm{Fingerprint}(T, r)$

Input: A tree T and its root node r.
Output: The fingerprint of the tree T.

```
1 fp ← 0;
2 for c ∈ children(r) do
3 |   fp ← hash(fp + Fingerprint(T, c));
4 end
5 fp ← hash(fp + hash(r.type));
6 return fp;
```

We employ tree-sitter[1] to extract the AST of the source code. Each node contains two attributes: type and value, e.g.,

$$\begin{aligned}\text{node}_1 &= \{\text{“type”: “variable name”, “value”: “count”}\}\,,\\ \text{node}_2 &= \{\text{“type”: “function name”, “value”: “sort’‘}\}\,.\end{aligned} \tag{9}$$

The type attribute reveals syntactic information, while the value attribute captures lexical details. To improve generalization, we retain only the type attribute (i.e., syntactic information) and discard the value attribute. Let T_{test} denote the AST of the current test source code (X_{test}), and let T_i denote the AST of the i-th sample's source code (X_i) in the database. CAST serves as the Score function, a surrogate measure for information coverage with definition: **A maximum covered subtree is a subtree of T_{test} that exists in the AST of exemplars and contains no smaller covered subtrees. CAST is computed as the ratio of nodes in the maximum covered subtrees to the total nodes in T_{test}.** The number of nodes in a tree equals the total number of its subtrees. Consequently, let $\mathrm{Sub}(T_S) = \mathrm{Sub}(\bigcup_{i=1}^{|S|} T_{S[i]})$ denote the subtrees of ASTs for all exemplars in the context. CAST is then defined as:

$$\mathrm{CAST}(S, X_{\text{test}}) = \frac{|\mathrm{Sub}(T_{\text{test}}) \cap \mathrm{Sub}(T_S)|}{|\mathrm{Sub}(T_{\text{test}})|}\,. \tag{10}$$

To compute CAST efficiently, we assign a fingerprint (fp) for each subtree via hashing, as detailed in Algorithm 1. The fingerprint encodes the node type (line 5) and recursively calculates the subtree fingerprint using post-order traversal and hashing. The set of fingerprints for the subtrees of a tree T_i is denoted as $\Phi_i = \{\mathrm{fp}_j^{(i)}\}_{j=1}^{|T_i|}$, where all $\mathrm{fp}_j^{(i)}$ are computed from line 5 in Algorithm 1 by calling $\mathrm{Fingerprint}(T, r)$ only once. The fingerprint set Φ_i for all samples $X_i \in \mathcal{D}$ in the database can be precomputed. To translate a new source code X_{test}, we first compute the fingerprint set for subtrees in its AST $\Phi_{\text{test}} = \{\mathrm{fp}_j^{(\text{test})}\}_{j=1}^{|T_{\text{test}}|}$, and then construct a co-occurrence matrix $M \in \{0,1\}^{n \times |\Phi_{\text{test}}|}$, with each element M_{ij} defined as:

$$M_{ij} = \begin{cases} 1, & \text{if } \mathrm{fp}_j^{(\text{test})} \in \Phi_i \\ 0, & \text{otherwise} \end{cases}\,. \tag{11}$$

This matrix indicates whether the j-th subtree of the test source code's AST (T_{test}) is present in the AST of the i-th source code sample (T_i) in the database.

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

To apply CAST as the Score function for list-wise exemplar selection in ICL (Eq. (2)), we need to maximize CAST (Eq. (10)), which is equivalent to maximizing $|\mathrm{Sub}(T_{\text{test}}) \cap \mathrm{Sub}(T_S)|$. Based on the definition of M, CAST maximization is formally expressed as:

$$\begin{aligned} \max_{S \subseteq \mathbb{N}_n} \quad & \left\| \bigvee_{s \in S} M[s,:] \right\|_1 \\ \text{s.t.} \quad & |S| = k\,. \end{aligned} \tag{12}$$

Here, $\|\cdot\|_1$ denotes the ℓ_1-norm of a vector, which counts the number of ones in the union vector $\bigvee_{s\in S} M[s,:] \in \{0,1\}^{|\Phi_{\text{test}}|}$.

This optimization problem is a subset selection task with a list-wise compositional objective, which is NP-hard that cannot be optimally resolve in polynomial time complexity.

B. Practical Solution: Greedy Submodular Maximization

To address the NP-hard challenge of CAST maximization, we propose a greedy submodular maximization algorithm. In exemplar selection for ICL, the law of diminishing marginal utility states that as more additional exemplars are selected, the marginal (incremental) information gain is non-increasing. This property holds for both information coverage and its surrogate measure, CAST. Formally, this property is known as submodularity in mathematics and is defined as:

Definition 1 (Submodularity [29])**.** *Let Ω denote a finite set, and let $f : 2^\Omega \to \mathbb{R}_{\geq 0}$ be a set function, where 2^Ω represents the power set of Ω. The function f is defined as submodular if it satisfies: For every $A \subseteq B \subseteq \Omega$, and for all $x \in \Omega \setminus B$,*

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)\,, \tag{13}$$

Additionally, f is monotone *if and only if* $f(A) \leq f(B)$ for all $A \subseteq B \subseteq \Omega$.

Maximizing a non-negative, monotone, and submodular function in subset selection referred to *submodular maximization*. Although generally NP-hard, a greedy algorithm that iteratively maximizes the marginal gain offers a theoretical $(1 - 1/e)$-approximate solution in polynomial time. The surrogate CAST (Eq. (10)) corresponds to the value function $f(S) = \left|\bigvee_{s\in S} M[s,:]\right|_1$ in the optimization problem (Eq. (12)). This value function is non-negative, monotone, and submodular. We present a lemma to prove these properties:

Lemma 2 (Properties of ℓ_1-norm in boolean vector operations)**.** *Let $A, B \in \{0,1\}^m$. Then, the following holds:*

$$\|B \wedge \neg A\|_1 = \|A \vee B\|_1 - \|A\|_1 = \|B\|_1 - \|B \wedge A\|_1\,, \tag{14}$$

where $\|\cdot\|_1$ denotes the ℓ_1-norm.

Proof. The ℓ_1-norm of a binary vector $A \in \{0,1\}^m$ counts the number of elements equal to "1":

$$\|A\|_1 = \sum_{i=1}^{m} A_i\,. \tag{15}$$

Boolean operations on binary vectors are performed elementwise, as follows:

$$\begin{aligned} (B \wedge \neg A)_i &= B_i(1 - A_i) = B_i - (B \wedge A)_i\,, \\ (A \vee B)_i &= A_i + B_i - (A_i \wedge B_i) = A_i + (B \wedge \neg A)_i\,. \end{aligned} \tag{16}$$

Therefore,

$$\begin{aligned} \|B \wedge \neg A\|_1 &= \sum_{i=1}^{m} B_i - (B \wedge A)_i = \|B\|_1 - \|B \wedge A\|_1\,, \\ \|B \wedge \neg A\|_1 &= \sum_{i=1}^{m} (A \vee B)_i - A_i = \|A \vee B\|_1 - \|A\|_1 \end{aligned} \tag{17}$$

Consequently, the proof is complete. □

Proposition 3 (The value function for CAST maximization is non-negative, monotone, and submodular.)**.** *Given a binary matrix $M \in \{0,1\}^{n\times m}$, the value function for an index set $S \subseteq \mathbb{N}_n$,*

$$f(S) = \left\| \bigvee_{s\in S} M[s,:] \right\|_1\,, \tag{18}$$

is non-negative, monotone, and submodular.

Proof. The function $f(S) = \left\|\bigvee_{s\in S} M[s,:]\right\|_1$ computes the number of ones resulting from the bitwise *OR* operation applied across the rows indexed by S.

1) **Non-negativity**: Since $f(S)$ counts the number of ones in the bitwise *OR* operation, it is always non-negative. Thus, we have $f(S) \geq 0$ for all $S \subseteq \mathbb{N}_n$.

2) **Monotonicity**: For any subsets $A \subseteq B \subseteq \mathbb{N}_n$, adding more rows to the bitwise *OR* operation cannot decrease the count of ones. Hence, $f(A) \leq f(B)$, i.e., $f(S)$ is monotone.

3) **Submodularity**: Given a subset $S \subseteq \mathbb{N}_n$, when adding a new index $x \in \mathbb{N}_n \setminus S$, the marginal gain is given by:

$$\begin{aligned} \Delta_S(x) &= f(S \cup \{x\}) - f(S) \\ &= \left\| \bigvee_{s\in S\cup\{x\}} M[s,:] \right\|_1 - \left\| \bigvee_{x\in S} M[s,:] \right\|_1 \\ &= \left\| M[x,:] \wedge \neg \bigvee_{s\in S} M[s,:] \right\|_1\,. \end{aligned} \tag{19}$$

This follows from $\|A\vee B\|_1 - \|A\|_1 = \|B\wedge\neg A\|_1$ in Lemma 2. Hence, for any subsets $A \subseteq B \subseteq \mathbb{N}_n$:

$$\begin{aligned} \Delta_B(x) &= \left\| M[x,:] \wedge \neg \bigvee_{s\in B} M[s,:] \right\|_1 \\ &= \left\| M[x,:] \wedge \neg \bigvee_{s\in A} M[s,:] \wedge \neg \bigvee_{s\in B\setminus A} M[s,:] \right\|_1 \\ &= \underbrace{\left\| M[x,:] \wedge \neg \bigvee_{s\in A} M[s,:] \right\|_1}_{\Delta_A(x)} \\ &\quad - \underbrace{\left\| M[x,:] \wedge \neg \bigvee_{s\in A} M[s,:] \wedge \bigvee_{s\in B\setminus A} M[s,:] \right\|_1}_{\geq 0}\,, \end{aligned} \tag{20}$$

where the last equation follows from Lemma 2, which states that $\|B\wedge\neg A\|_1 = \|B\|_1 - \|B\wedge A\|_1$. So, $\Delta_A(x) - \Delta_B(x) \geq 0$,

thereby the value function f is submodular, $\forall A \subseteq B \subseteq \mathbb{N}_n$ and $\forall x \in \mathbb{N}_n \setminus B$, the following holds:

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B). \tag{21}$$

Hence, the function $f(S)$ is non-negative, monotone, and submodular. The proof is complete. □

Consequently, the list-wise exemplar set selection problem in Eq. (12) is a *submodular maximization* problem.

Submodular maximization is generally NP-hard. However, when the value function is non-negative, monotone, and submodular, a greedy algorithm can be used. It starts with an empty set and iteratively adds the element with the maximum marginal gain ($\Delta_S(x)$):

$$x^* = \arg\max_{x \in \Omega \setminus S} f(S \cup \{x\}) - f(S)\,, \tag{22}$$

continuing until k elements are selected. It provides a solution within polynomial time and guarantees a near-optimal approximation, as detailed in the following theorem [29], [30]:

Theorem 4 ($(1-1/e)$-approximation of greedy submodular maximization [29])**.** *For any non-negative, monotone submodular function f, the greedy heuristic produces a solution $\hat{S}$ of size $|\hat{S}| = k$ with the following approximation guarantee:*

$$\begin{aligned} f(\hat{S}) &\geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) f(S^*) \\ &\geq \lim_{k \to +\infty} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) f(S^*) \\ &= \left(1 - \frac{1}{e}\right) f(S^*)\,, \end{aligned} \tag{23}$$

where S^* represents the optimal solution to the submodular maximization, defined as $S^* = \arg\max_{S \subseteq \Omega, |S| \leq k} f(S)$.

Therefore, we propose a greedy algorithm to solve the CAST maximization problem, guided by Eq. (22). The algorithm starts with an empty selected set, $S \leftarrow \emptyset$. In each iteration, the algorithm selects a new exemplar that maximizes the marginal gain $\Delta_S(f)$ of Eq. (12):

$$\begin{aligned} j &= \arg\max_{i \in \mathbb{N}_n \setminus S} \left\| \bigvee_{s \in S \cup \{i\}} M[s,:] \right\|_1 - \left\| \bigvee_{s \in S} M[s,:] \right\|_1\,, \\ &\Downarrow \textbf{Lemma 2} \\ j &= \arg\max_{i \in \mathbb{N}_n \setminus S} \left\| M[i,:] \wedge \neg \bigvee_{s \in S} M[s,:] \right\|_1\,. \end{aligned} \tag{24}$$

The selected index j is added to S in each iteration until the subset reaches the predefined size limit k. The pseudocode of this greedy CAST maximization is provided in Algorithm 2.

As shown in Theorem 4, our CAST maximization method in Eq. (24) and Algorithm 2 provides a $(1-1/e)$-approximation, ensuring sufficient effectiveness. For instance, if $k = 3$ exemplars are selected, the approximation ratio reaches at least $1 - (1-1/3)^3 \approx 70.37\%$.

The time complexity of Algorithm 2 includes three parts: $\mathcal{O}(k)$ for the loop in line 3, $\mathcal{O}(n)$ for the loop in line 5, and $\mathcal{O}(|\Phi_{\text{test}}|)$ for bitwise boolean computations in line 7. Thus, the total time complexity of the CAST algorithm is $\mathcal{O}(nk \cdot |\Phi_{\text{test}}|)$. Let $L_{X_{\text{test}}}$ denote the length of the test source code. The size of the subtree fingerprint set is upper-bounded by $|\Phi_{\text{test}}| \leq L_{X_{\text{test}}} \log(L_{X_{\text{test}}})$. Moreover, the selection size k and the subtree scale $|\Phi_{\text{test}}|$ are much smaller than the database size n, i.e., $k \ll n$ and $|\Phi_{\text{test}}| \ll n$. As a result, the greedy CAST maximization algorithm in Algorithm 2 is computationally efficient. Furthermore, the iterations in lines 5-13 are independent, allowing the loop in line 5 to be parallelized for faster computation. With sufficient resources in industrial applications, this optimization reduces the time complexity to $\mathcal{O}(k \cdot |\Phi_{\text{test}}|)$, further improving efficiency.

Constructing the co-occurrence matrix takes $\mathcal{O}(n \cdot |\Phi_{\text{test}}|)$ time, based on its size ($M^{n \times |\Phi_{\text{test}}|}$). Extracting the subtree fingerprint set Φ_{test} from the test source code takes $\mathcal{O}(|\Phi_{\text{test}}|)$ time using a single post-order traversal on T_{test}. Moreover, the subtree fingerprint sets for database samples can be precomputed, avoiding extra computations at test time. An inverted index links each subtree fingerprint fp to the corresponding database samples. For a new test source code, the inverted index (e.g., using ElasticSearch) efficiently retrieves co-occurrence coordinates for each fp $\in \Phi_{\text{test}}$ in $\mathcal{O}(|\Phi_{\text{test}}|)$ time. Each fp retrieves a sparse column (in $\{0,1\}^n$) for the co-occurrence matrix $M \in \{0,1\}^{n \times |\Phi_{\text{test}}|}$. Combining sparse columns to form the full matrix M takes $\mathcal{O}(n \cdot |\Phi_{\text{test}}|)$.

In summary, constructing the co-occurrence matrix for CAST takes $\mathcal{O}(n \cdot |\Phi_{\text{test}}|)$, while the greedy CAST algorithm in Algorithm 2 has a polynomial time complexity of $\mathcal{O}(nk \cdot |\Phi_{\text{test}}|)$. Using distributed and parallel computation, this can be further reduced to $\mathcal{O}(k \cdot |\Phi_{\text{test}}|)$. Thus, our approach ensures both theoretical effectiveness with a near-optimal $(1-1/e)$-approximation and practical efficiency.

Algorithm 2: Greedy CAST Submodular Maximization

Input: Co-occurrence matrix $M \in \{0,1\}^{n \times |\Phi_{\text{test}}|}$ and selection capacity limit $k \ll n$.
Output: Subset of exemplar indices $S \subseteq \mathbb{N}_n$ such that $|S| = k$.

1 Initialize the selected set: $S \leftarrow \emptyset$;
2 Initialize the coverage mask: $c \leftarrow \mathbf{0}^{|\Phi_{\text{test}}|}$;
3 **for** $i \leftarrow 1$ ***to*** k **do**
4 $s \leftarrow -1$; max_gain $\leftarrow -1$;
5 **for** $j \leftarrow 1$ ***to*** n **do**
6 **if** $j \notin S$ **then**
7 $g \leftarrow \|M[j,:] \wedge \neg c\|_1$; `// marginal gain`
8 **if** $g >$ *max_gain* **then**
9 max_gain $\leftarrow g$;
10 $s \leftarrow j$;
11 **end**
12 **end**
13 **end**
14 Update coverage mask: $c \leftarrow c \vee M[s,:]$;
15 Update selected set: $S \leftarrow S \cup \{s\}$;
16 **end**
17 **return** S;

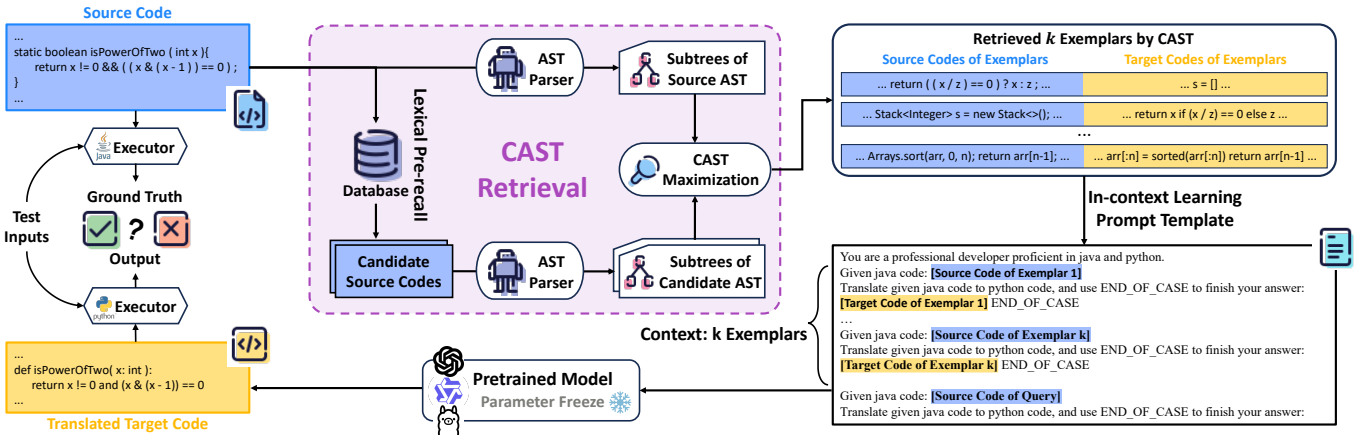


Fig. 2. Overview of post-incorporating code syntactic knowledge into pre-trained models using ICL based on CAST retrieval.

C. Overall Framework of CAST

Figure 2 illustrates the overall pipeline of our approach, which post-incorporates code syntactic structure knowledge into off-the-shelf LLMs through ICL based on Covering Abstract Syntax Tree (CAST) retrieval. To accelerate the overall process, lexical similarity is used to coarsely pre-recall $\lfloor t \cdot k \rfloor$ candidate samples, where $t \geq 1$ is a hyperparameter. Specifically, the Levenshtein Distance is employed to recall the top-$\lfloor t \cdot k \rfloor$ nearest candidates. The fingerprint set of the test source code Φ_{test} is extracted using Algorithm 1, and a co-occurrence matrix M is constructed with all pre-recalled candidate samples. Then, Algorithm 2 selects k exemplars as context. The selected k exemplar translation pairs are directly concatenated with the test source code to create the prompt, as shown in Fig. 2. This prompt is then input into pre-trained models, such as LLMs, to translate the test source code.

Besides, our CAST retrieval method includes two variants: 1) **CAST-F**: Selects a fixed number of exemplars based on the predefined capacity limit k. 2) **CAST-A**: Automatically determines the number of exemplars using a CAST (defined in Eq. (10)) threshold, allowing early stop when the coverage rate reaches the threshold.

IV. Experiments

To evaluate CAST, we integrate it as a plug-and-play module with SOTA LLMs of varying scales during inference and compare it with traditional exemplar selection strategies based on similarity and diversity. Extensive experiments are conducted to evaluate the effectiveness and efficiency of CAST on thousands of translation tasks.

A. Dataset and Database

We use the dataset from [31], containing 948 parallel functions in Python, Java, and C++, widely used in code translation [31]–[33]. Of these, 568 include unit tests for at least one programming language: 464 for Python, 482 for Java, and 467 for C++. Following the previous work [34], our experiments focus on this subset. To further validate our approach, we evaluate various methods on AVATAR [35], a more realistic benchmark constructed from competitive programming sites, online platforms, and open-source repositories. We also extend our evaluation beyond code translation to code summarization using benchmarks from CodeXGLUE [36]. ICL requires a separate database. For each pre-trained LLM, we use it to translate the entire test dataset and verify the predictions using test input cases. Samples that pass all test input cases are added to the database for this LLM. Models are evaluated on the remaining data, and final metrics are computed using the entire dataset, treating the database samples as exactly correct. The replication package is publicly available at https://doi.org/10.5281/zenodo.16920815.

B. Baselines

We apply CAST-F and CAST-A to SOTA LLMs and compare them with LLMs using various ICL exemplar selection strategies, as well as several learning-based translation models.

1) Baseline Code Translation Models: We compare our method with recent LLM families across various parameter scales, consisting of 14 advanced LLMs:

- **CodeGen** [2]: An LLM trained in natural language and programming data for conversation-based program synthesis. We employ CodeGen-6B-Multi in the experiment.
- **Qwen** [37], [38]: A suite of models with parameter sizes ranging from 0.5B to 72B, including dense and Mixture-of-Experts variants, excelling in language, multilingual tasks, coding, math, and reasoning.
- **Gemma** [39]: A lightweight LLMs, ranging in scale from 2B to 27B parameters.
- **DeepSeek-Coder** [1], [40]: A series of open-source code LLMs, pre-trained on 2 trillion tokens using the fill-in-the-blank task to enhance code generation and infilling.
- **LLaMA** [41], [42]: Multilingual LLMs with parameter sizes from 8B to 70B, are designed to enhance inference, code generation, and instruction compliance.
- **CodeLLaMA** [3]: A series of code-centric LLMs derived from LLaMA2 [42], further trained on 500B code tokens.
- **GPT-4o** [43]: Advanced generative AI models developed by OpenAI. Although not explicitly trained for code generation, they demonstrate notable performance in this domain. Their effectiveness in handling code generation tasks is primarily attributed to their massive scale.

TABLE I
RESULTS OF PRE-TRAINED MODELS, INCLUDING LLMS AND CODE TRANSLATION MODELS, WITH ONE-SHOT FIXED EXEMPLAR STRATEGY.

Models	C++ to Python		Python to C++		Java to C++		C++ to Java		Average	
	CA	EM	CA	EM	CA	EM	CA	EM	CA	EM
Off-the-shelf Large Language Models										
CodeGen-6B	30.17%	4.40%	22.70%	1.94%	35.12%	7.57%	11.20%	2.46%	24.80%	4.09%
CodeLLaMA-13B	23.24%	2.29%	16.90%	2.11%	11.62%	3.35%	16.73%	3.52%	14.48%	2.82%
CodeLLaMA-34B	45.07%	6.16%	39.08%	4.23%	23.59%	7.57%	38.38%	14.96%	36.53%	8.23%
CodeLLaMA-70B	44.01%	4.58%	50.00%	3.70%	49.65%	16.37%	43.66%	11.80%	46.83%	9.11%
DeepSeek-Coder-6.7B	44.97%	7.22%	19.75%	3.14%	34.04%	13.73%	53.17%	17.78%	37.98%	10.47%
DeepSeek-Coder-33B	53.70%	7.75%	28.87%	4.01%	39.44%	14.96%	54.06%	18.31%	44.02%	11.26%
Gemma2-2B	47.36%	6.51%	39.61%	2.64%	54.05%	12.15%	22.18%	5.28%	40.80%	6.65%
Gemma2-9B	60.39%	8.45%	59.26%	4.75%	65.61%	**31.34%**	20.81%	23.06%	51.52%	**16.90%**
Gemma2-27B	63.38%	8.45%	63.91%	5.33%	68.49%	24.96%	**68.31%**	21.83%	66.02%	15.14%
Qwen2-7B	58.27%	6.16%	50.70%	4.40%	60.74%	13.20%	50.88%	11.62%	55.15%	8.85%
Qwen2.5-72B	69.01%	8.63%	68.49%	6.34%	71.30%	23.42%	64.61%	24.30%	68.36%	15.67%
LLaMA3-70B	69.37%	5.63%	60.74%	**10.03%**	67.96%	23.39%	64.61%	**27.99%**	65.67%	16.76%
GPT-4o-mini	70.95%	9.33%	**69.01%**	6.16%	70.95%	6.69%	54.58%	22.01%	66.37%	11.05%
GPT-4o	**71.65%**	**9.49%**	68.66%	5.99%	**72.71%**	23.02%	65.85%	26.41%	**69.72%**	16.23%
Learning-based Code Translation Models										
TransCoder	36.64%	2.29%	30.40%	0.88%	27.84%	5.81%	49.77%	14.39%	36.16%	5.84%
TransCoder-IR	/	/	/	/	40.99%	14.26%	50.53%	18.28%	45.76%	16.27%
TransCoder-ST	46.34%	2.64%	47.85%	1.06%	49.68%	9.68%	64.73%	17.43%	52.15%	7.70%

$^{\Phi}$ For clarification, we adopt "-" to concatenate LLM and corresponding parameter size for discrimination. For example, Qwen2 with 7B parameters is dubbed Qwen2-7B.

Moreover, three state-of-the-art learning-based code translation models serve as baselines.

- **TransCoder** [31]: An unsupervised code translation model using monolingual samples, pre-trained with cross-lingual modeling, denoising, and back-translation.
- **TransCoder-IR** [32]: A model extended on TransCoder that incorporates compiler intermediate representation (IR) to enhance code translation.
- **TransCoder-ST** [33]: A model extended on TransCoder that utilizes automatically generated test cases to filter out invalid translation results.

2) Baseline ICL Exemplar Selection Strategies: We evaluate several baseline ICL exemplar selection strategies to compare with our proposed CAST variants.

These strategies encompass simple heuristic methods, similarity-based approaches (e.g., LD, CodeBERT, and BM25), and diversity-oriented techniques (e.g., Diversity), which are essentially different implicit surrogates of Eq. (8). Considering only similarity is equivalent to setting $\lambda = 0$ in Eq. (8). Each exemplar selection method can be defined as a specific instantiation of the general formulation Eq. (3).

Fixed Exemplars (Fixed): A static exemplar set is shared across all test samples:

$$S^* = S_{\text{fixed}}, \quad \forall X_{\text{test}}\,. \tag{25}$$

Random Sampling (Random): k exemplars are uniformly sampled from the candidate pool, independent of X_{test}:

$$S^* \sim \text{Uniform}(\mathbb{N}_n), \quad |S^*| = k\,. \tag{26}$$

Retrieval by Levenshtein Distance (LD): The score is defined as the negative sum of Levenshtein distances between each exemplar and the test sample:

$$S^* = \arg \max_{S \subseteq \mathbb{N}_n,\, |S|=k} -\sum_{i \in S} \text{LD}(X_i, X_{\text{test}})\,. \tag{27}$$

Similarly, the method **Retrieval by AST Edit Distance (AST-ED)** is defined based on the Tree Edit Distance [44] between each exemplar and the test sample, where only the type information of AST nodes is retained for distance calculation to improve generalization, aligning with the setting of CAST.

Retrieval by CodeBERT Similarity (CodeBERT): Each sample is encoded using a CodeBERT embedding function $f_{\text{CB}}(\cdot)$, and cosine similarity is used:

$$S^* = \arg \max_{S \subseteq \mathbb{N}_n,\, |S|=k} \sum_{i \in S} \cos\left(f_{\text{CB}}(X_i), f_{\text{CB}}(X_{\text{test}})\right)\,. \tag{28}$$

Retrieval by BM25 (BM25): The score is the sum of BM25 relevance scores between each candidate and the test input:

$$S^* = \arg \max_{S \subseteq \mathbb{N}_n,\, |S|=k} \sum_{i \in S} \text{BM25}(X_i, X_{\text{test}})\,. \tag{29}$$

Diversity with Clustering (Diversity): The candidate pool is clustered into k groups using CodeBERT embeddings, and the most similar sample in each cluster is selected:

$$\mathbb{N}_n \xrightarrow{\text{k-means}(f_{\text{CB}}(X_i))} \{C_1, \ldots, C_k\}\,, \tag{30}$$

where C_j represents the index set of the j-th cluster.

$$S^*_{[i]} = \arg \max_{j \in C_i} \cos\left(f_{\text{CB}}(X_j), f_{\text{CB}}(X_{\text{test}})\right)\,, \tag{31}$$

for $i \in \{1, \ldots, k\}$, and $S^* = S^*_{[1:k]}$.

C. Evaluation Metrics

We employ Computational Accuracy (CA) and Exact Match Accuracy (EM) to evaluate model performance, following the prior works [31]–[34].

Traditional studies primarily use the Bi-Lingual Evaluation Understudy (BLEU) [45] metric for evaluating code translation as a natural language translation task. However, in code

TABLE II
RESULTS OF LLMS OF VARIOUS PARAMETER SCALES FOR 5-SHOT CODE TRANSLATION UPON DIFFERENT ICL STRATEGIES.

Models	Strategy	C++ to Py	Py to C++	Java to C++	C++ to Java	Py to Java	Java to Py	Average
Qwen2-7B	Random	63.38%	59.31%	66.54%	61.27%	49.74%	67.43%	61.28%$^\bullet$
	LD	64.96%	57.69%	67.96%	63.73%	52.03%	67.61%	62.33%$^\bullet$
	AST-ED	64.96%	61.09%	68.13%	64.44%	52.99%	67.08%	63.12%$^\bullet$
	BM25	64.61%	58.98%	67.25%	63.73%	50.18%	67.78%	62.09%$^\bullet$
	CodeBERT	64.79%	57.02%	66.02%	63.91%	53.35%	66.37%	61.91%$^\bullet$
	Diversity	64.96%	59.51%	67.42%	65.15%	51.41%	67.61%	62.68%$^\circ$
	CAST-F	**65.32%**	**61.62%**	**68.84%**	**65.15%**	**55.77%**	**69.01%**	**64.28%**
Gemma2-27B	Random	67.25%	67.08%	70.77%	71.83%	66.02%	69.89%	68.81%$^\bullet$
	LD	67.43%	66.73%	70.95%	69.54%	65.49%	69.54%	68.28%$^\bullet$
	AST-ED	67.25%	67.96%	70.77%	69.72%	66.55%	69.19%	68.57%$^\bullet$
	BM25	68.13%	67.43%	70.95%	71.65%	65.32%	69.54%	68.84%$^\bullet$
	CodeBERT	67.25%	66.55%	70.42%	70.25%	64.79%	69.72%	68.16%$^\bullet$
	Diversity	68.13%	66.55%	70.77%	69.37%	64.08%	69.89%	68.13%$^\bullet$
	CAST-F	**70.60%**	**68.66%**	**71.13%**	**72.49%**	**66.90%**	**70.40%**	**70.03%**
Qwen2.5-72B	Random	69.54%	70.07%	73.94%	66.73%	61.80%	72.01%	69.02%$^\circ$
	LD	70.25%	70.42%	71.48%	70.95%	59.15%	72.18%	69.07%$^\circ$
	AST-ED	70.60%	70.07%	73.24%	66.90%	63.03%	71.65%	69.25%$^\circ$
	BM25	70.42%	70.07%	70.95%	68.31%	64.96%	71.13%	69.31%$^\circ$
	CodeBERT	70.42%	69.72%	71.65%	70.42%	60.56%	72.01%	69.13%$^\bullet$
	Diversity	70.25%	69.89%	73.06%	71.30%	64.96%	71.65%	70.19%$^\bullet$
	CAST-F	**70.77%**	**70.77%**	**75.53%**	**73.06%**	**65.31%**	**72.54%**	**71.33%**
GPT-4o	Random	72.01%	69.19%	73.42%	70.95%	69.01%	71.48%	71.01%$^\bullet$
	LD	72.36%	70.25%	73.59%	71.48%	69.54%	71.48%	71.45%$^\bullet$
	AST-ED	72.71%	70.42%	73.59%	71.30%	69.54%	71.48%	71.51%
	BM25	72.36%	70.60%	73.06%	71.83%	69.19%	71.30%	71.39%$^\bullet$
	CodeBERT	73.06%	69.89%	73.94%	71.48%	69.37%	71.48%	71.54%$^\bullet$
	Diversity	72.36%	69.54%	73.59%	71.13%	69.37%	71.65%	71.27%$^\bullet$
	CAST-F	**73.24%**	**70.77%**	**74.12%**	**72.01%**	**70.07%**	**71.65%**	**71.98%**

TABLE III
RESULTS OF VARIOUS ICL STRATEGIES FOR 5-SHOT TRANSLATION ON AVATAR [35] OF QWEN2-7B.

Strategy	Java to Python						Python to Java					
	CA	CodeBLEU	BLEU	WNG	Syntax.	Dataflow.	CA	CodeBLEU	BLEU	WNG	Syntax.	Dataflow.
Random	39.36%	20.06	1.56	2.29	**36.39**	35.40	46.40%	20.88	2.01	3.10	43.88	29.56
LD	40.56%	19.73	1.56	2.23	34.77	33.40	47.59%	21.30	1.90	2.90	42.48	29.96
BM25	46.59%	19.85	1.61	2.26	36.03	35.50	44.00%	21.39	1.94	2.98	43.27	30.16
CodeBERT	46.59%	20.36	1.55	2.18	35.16	**36.11**	49.20%	20.88	1.99	3.14	44.08	30.30
Diversity	39.76%	19.72	1.51	2.14	34.69	33.70	44.40%	20.80	1.99	3.14	44.08	29.97
CAST-F	**47.79%**	**22.04**	**1.73**	**2.55**	35.79	33.80	**50.40%**	**21.55**	**2.17**	**3.38**	**45.60**	**31.54**

translation, correctness of code execution is more critical. CA measures the percentage of translated code that produces identical execution results to the ground truth code under the same inputs. This metric assesses execution consistency, assuming the provided inputs adequately cover all boundary conditions, and is defined as:

$$\mathrm{CA} \triangleq \mathop{\mathbb{E}}_{X_{\text{test}}} \left[\mathbb{I}\left(\mathrm{Exec}(X_{\text{test}}, \mathcal{D}_X^{\text{test}}) = \mathrm{Exec}(\widehat{Y}_{\text{test}}, \mathcal{D}_X^{\text{test}}) \right) \right] \quad (32)$$

$\mathcal{D}_X^{\text{test}}$ denotes the input set for the current source code X_{test}. $\mathrm{Exec}(X_{\text{test}}, \mathcal{D}_X^{\text{test}})$ and $\mathrm{Exec}(\widehat{Y}_{\text{test}}, \mathcal{D}_X^{\text{test}})$ indicate the outputs produced by executing the source code X_{test} and the translated target code $\widehat{Y}_{\text{test}}$ with the input cases $\mathcal{D}_X^{\text{test}}$, respectively.

Additionally, since the datasets provide ground truth code in the target programming languages, we evaluate EM accuracy, defined as the proportion of target translations that exactly match the ground truth target code.

D. Empirical Results and Analysis

RQ1: How do SOTA LLMs of various scales perform in the code translation task?

In this research question, we evaluate SOTA LLMs across various scales for code translation. As shown in Tab. I, the results reveal several meaningful observations: 1) The LLMs demonstrate impressive performance in the code translation task, highlighting their potential for practical application. Specifically, the proprietary LLMs, such as GPT-4o and GPT-4o-mini, achieve an average CA of 66.37% and 69.72%, respectively, indicating strong potential for LLM-based code translation. 2) A high EM score indicates that the translated code is identical to the ground truth, often used as a metric to assess the extent of training data leakage. Compared to learning-based methods, which ensure no data leakage during training, the EM scores of LLMs are comparable, indicating minimal data leakage. This shows that LLMs can effectively understand the task through prompts and produce semantically

correct translated code. 3) Most LLMs outperform learning-based models. Within the same model series, larger models, such as the Gemma2 series, perform better than smaller ones, indicating that scaling up parameters enhances the code translation capabilities of LLMs.

Finding 1: LLMs demonstrate remarkable performance in code translation, surpassing learning-based methods and showcasing high effectiveness and generalization. Scaling up the parameters of LLMs further enhances their code translation capabilities, highlighting their strong potential for industrial applications.

RQ2: How does CAST perform with SOTA LLMs compared to different ICL exemplar selection strategies? Referring to the results in Tab. I, we choose three open-source LLMs as primary baselines: Qwen2-7B, the SOTA LLM under 10B parameters; Gemma2-27B, the SOTA LLM under 40B parameters; and Qwen2.5-72B, the SOTA LLM exceeding 70B parameters. CAST-F is used to select exemplars for ICL, and its performance is compared against other exemplar selection strategies. Results in Tab. II demonstrate the effectiveness of CAST-F across LLMs with various parameter scales in the 5-shot code translation task. We conduct t-tests comparing CAST-F with each other ICL retrieval strategies. The resulting p-values indicate that CAST-F significantly outperforms alternative strategies in most settings, with particularly strong statistical advantages observed on GPT-4o and Gemma2-27B. Methods that are highly significantly inferior to our approach ($p < 0.05$) are marked with "•", and those that are significantly inferior ($0.05 \leq p < 0.1$) are marked with "○". There are several key observations: 1) CAST-F consistently outperforms all other ICL exemplar selection strategies across LLMs and translation tasks. Specifically, it surpasses the second-best strategy, Diversity, by 1.6% for Qwen2-7B, 1.9% for Gemma2-27B, and 1.14% for Qwen2.5-72B. This shows that information coverage is a more precise and general measure for ICL, and CAST-F effectively integrates code structural knowledge into LLMs to enhance code translation. Compared with other models, since GPT-4o, as a commercial-closed model, inherently possesses strong capabilities, the performance improvement space brought by structural information injection is not significant, but CAST still maintains a certain advantage compared with baselines. 2) For Gemma2-27B, exemplar selection using LD, CodeBERT, and Diversity performs worse than random sampling. This highlights the importance of effective exemplar selection strategies in ICL. CAST-F, the only list-wise method that considers both information coverage and code structural knowledge, consistently achieves superior results compared to other strategies. 3) AST-ED focuses more on measuring the similarity of the overall structure of the code. In contrast, CAST is based on subtree coverage, placing greater emphasis on matching local structural modules. It provides module-by-module references for translation, effectively reducing translation deviations in local logic and thereby improving the overall performance.

In real-world scenarios, the quality and structure of code repositories vary considerably. To more accurately assess the practical effectiveness of our method, we further evaluated its code translation performance on the AVATAR [35] dataset, which includes 249 Java samples, 250 Python samples, and 6,255 test cases. In addition to computation accuracy, we also computed a range of metrics, including BLEU, Weighted N-Gram (WNG) Match Score, Syntactic Match Score, Dataflow Match Score, and CodeBLEU (weighted combination of the above four parts) [46], offering a more comprehensive evaluation of translation quality. The experimental results, presented in Tab. III, reveal significant performance differences across various ICL strategies in 5-shot code translation using Qwen2-7B on AVATAR. Among the evaluated methods, CAST-F consistently outperforms others in both Java-to-Python and Python-to-Java translation tasks, achieving the highest computational accuracy at 47.79% and 50.40%, respectively, as well as the best overall CodeBLEU scores. These results underscore CAST's strong generalization ability and its practical effectiveness in real-world code translation tasks. Specifically, CAST-F excels not only in BLEU and WNG but also in Syntactic and dataflow alignment, demonstrating its ability to capture both surface-level and structural aspects of code. In contrast, baseline strategies like Random and Diversity yield lower scores across most metrics, highlighting the importance of informed example selection in ICL-based code translation.

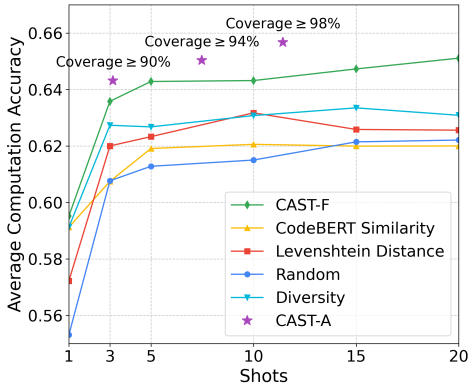
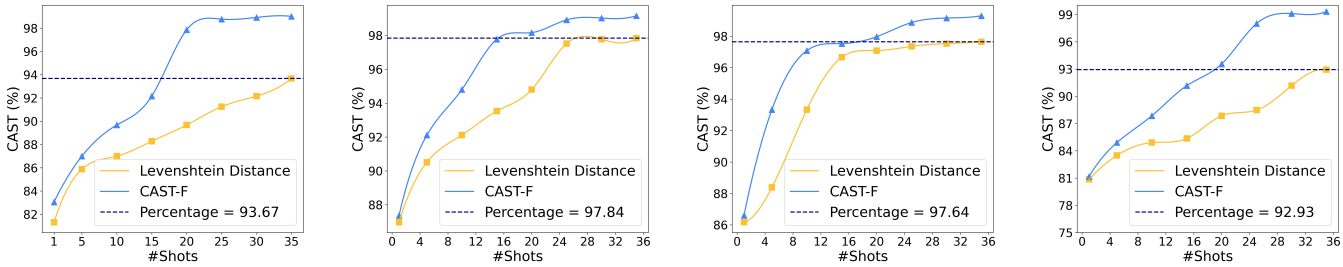


Fig. 3. The performance of Qwen2-7B with various strategies and shot count.

Finding 2: CAST-F demonstrates robust performance across various LLMs, translation tasks, real-world scenarios, and multiple metrics, emphasizing its ability to capture both surface-level and structural features of code. Consequently, information coverage provides a more precise and generalized measure for ICL, with CAST-F effectively incorporating code structural knowledge into LLMs to improve code translation.

RQ3: How does ICL perform using different exemplar selection strategies as the number of exemplars increases? Results in Tab. IV and Fig. 3 show the performance of Qwen2-7B in code translation using various ICL exemplar selection strategies, with the number of exemplars increasing from 1-shot to 20-shots. Several key observations arise from this analysis: 1) CAST-F consistently achieves the highest average performance across all strategies, with an accuracy of 65.11% at 20-shots. 2) Increasing the number of exemplars generally improves performance across most strategies. CAST-F demonstrates superior scalability, particularly at higher shot counts, underscoring its robustness in improving code translation. 3) Compared to baselines, CAST-F consistently delivers in-



(a) C++ to Java (b) Python to C++ (c) Python to Java (d) C++ to Python

Fig. 4. Comparison of the average coverage of abstract syntax tree (CAST) between the CAST-F and Levenshtein distance (LD) strategies.

cremental gains by leveraging information coverage and code structural knowledge. 4) The Diversity strategy ranks second, performing well by considering both point-wise similarity and diversity, but it lacks the ability to account for code structural knowledge or perform selection in a compositional manner. 5) CAST-F achieves the highest accuracy across challenging language pairs, such as Python to Java (58.10%) and Java to Python (69.37%), demonstrating its effectiveness in translating between syntactically diverse languages. 6) A law of diminishing marginal utility is evident in Tab. IV and Fig. 3 as the number of exemplars increases. While performance improves significantly with the first few exemplars, additional gains become minimal beyond 5-shots due to redundancy or information saturation. However, Qwen2-7B can remit this issue effectively by considering information coverage as a more precise and general measure.

Finding 3: Despite diminishing marginal utility with more exemplars in all ICL strategies, CAST-F shows superior performance and scalability. It achieves the highest accuracy at higher shot counts by considering information coverage and code structure.

RQ4: How does CAST-A perform when dynamically determining the selection size?

Results of CAST-A, as presented in Tab. IV and Fig. 3, reveal several key observations: 1) CAST-A achieves competitive performance compared to fixed-size exemplar selection strategies with fewer exemplars. It dynamically adjusts the selection size based on a CAST threshold (e.g., CAST$\geq$ 98%), surpassing all baselines at 20-shots while using an average of only 11.43 exemplars. 2) Compared to fixed-shot strategies, CAST-A mitigates the issue of diminishing returns observed in static exemplar selection by dynamically adjusting the selection size to balance information saturation and redundancy. 3) CAST-A demonstrates superior adaptability to varying input complexity. As shown in Fig. 3, CAST-F forms the Pareto frontier between input complexity (exemplar count) and performance (average CA) across all fixed-shot strategies. Surprisingly, CAST-A with different thresholds occupies the front of the Pareto frontier of fixed-strategies, showcasing the high effectiveness, efficiency, and robustness of both CAST-F and CAST-A.

Finding 4: CAST-A effectively adapts the selection size based on the CAST threshold, outperforming fixed-size strategies while using fewer exemplars. This dynamic adjustment mitigates diminishing returns and improves scalability, showcasing its robustness in handling varying input complexities.

RQ5: How does the CAST convergence of CAST-F compare with that of similarity-only strategy (e.g., LD)?

CAST serves as a surrogate measure of information, indicating the ratio of necessary knowledge gained from exemplars during code translation. As shown in Fig. 4, CAST is computed from exemplars retrieved by CAST-F and LD across varying shot counts, revealing several key observations: 1) CAST-F consistently converges faster than LD as the number of shots increases, demonstrating greater efficiency in acquiring necessary knowledge from exemplars for translation tasks. 2) CAST-F achieves a higher final CAST ratio than LD across all translation tasks, showcasing its robustness in optimizing exemplar selection. This emphasizes the advantage of CAST-F in handling complex translations.

Finding 5: CAST-F accelerates the acquisition of necessary knowledge and ensures higher final gains across all tasks. These advantages demonstrate its effectiveness in optimizing exemplar selection for diverse and complex code translation scenarios.

RQ6: Does scaling up the model size automatically lead to sufficient code structural knowledge?

As shown in Tab. II, although translation performance improves with larger models, the improvement from explicitly incorporating the code structural knowledge by CAST-F is much more substantial. Additionally, for a fairer comparison, we evaluate CAST-F with 5-shots on Gemma2-(2B/9B/27B), all of which are from the same family. The results, shown in Tab. V, indicate that the improvement from CAST-F is not significantly related to model scale. This confirms that the additional code structural knowledge from CAST-F is independent of model scale. Therefore, simply scaling up the model or training data does not result in the sufficient emergence of code structural knowledge, highlighting the necessity of explicitly incorporating code syntax.

Finding 6: Scaling up model size or training data does not lead to the sufficient emergence of code structural knowledge, underscoring the necessity of explicitly considering code syntactic structure in LLMs.

RQ7: What is the hyperparameter sensitivity?

In CAST-F, there is only one hyperparameter, i.e., the factor t for adjusting the size of lexical pre-recall ($\lfloor t \times k \rfloor$). We evaluate CAST-F across varying values of the factor t and the selection size k. Fig. 5 shows the results, which indicate that regardless of the selection size k, $t = 2$ yields the best performance, and

TABLE IV
PERFORMANCE OF QWEN2-7B ON CODE TRANSLATION WITH VARIOUS ICL EXEMPLAR SELECTION STRATEGIES AND NUMBERS OF EXEMPLARS.

Strategy	#Shots	C++ to Py	Py to C++	Java to C++	C++ to Java	Py to Java	Java to Py	Average
Fixed	1	58.27%	50.70%	60.74%	50.88%	20.07%	63.20%	50.64%
Random	1	60.39%	54.93%	66.90%	52.11%	29.93%	67.60%	55.31%
	3	63.02%	57.02%	65.14%	61.90%	49.74%	67.78%	60.77%
	5	63.38%	59.31%	66.55%	61.27%	49.74%	67.43%	61.28%
	10	63.73%	57.57%	66.55%	63.73%	49.65%	67.78%	61.50%
	15	63.20%	59.86%	66.73%	64.96%	50.70%	67.42%	62.15%
	20	63.56%	59.15%	66.02%	64.08%	51.94%	68.49%	62.21%
LD	1	59.51%	59.51%	67.43%	54.57%	34.51%	67.78%	57.22%
	3	61.44%	60.92%	67.78%	63.38%	51.06%	67.43%	62.00%
	5	64.96%	57.69%	67.96%	63.73%	52.03%	67.61%	62.33%
	10	64.26%	60.73%	67.43%	65.67%	52.46%	68.49%	63.17%
	15	63.73%	59.15%	66.90%	65.49%	52.99%	67.25%	62.59%
	20	64.44%	59.15%	67.43%	64.43%	52.64%	67.25%	62.56%
BM25	1	63.20%	57.04%	66.37%	57.04%	40.32%	67.61%	58.60%
	3	64.26%	58.45%	67.78%	62.68%	49.65%	67.78%	61.77%
	5	64.61%	58.98%	67.25%	63.73%	50.18%	67.78%	62.09%
	10	64.44%	59.15%	67.25%	63.56%	53.17%	68.13%	62.62%
	15	65.14%	60.56%	67.08%	64.08%	52.46%	68.13%	62.91%
	20	65.84%	60.74%	67.78%	65.32%	53.35%	68.31%	63.56%
CodeBERT	1	64.61%	56.69%	64.61%	54.92%	46.13%	67.78%	59.12%
	3	63.91%	57.39%	66.73%	62.15%	47.05%	67.25%	60.75%
	5	64.79%	57.02%	66.02%	63.91%	53.35%	66.37%	61.91%
	10	64.79%	56.16%	65.49%	64.08%	53.17%	68.66%	62.06%
	15	65.32%	55.96%	65.32%	64.43%	53.17%	67.78%	62.00%
	20	64.44%	57.75%	65.49%	61.62%	54.23%	68.49%	62.00%
Diversity	1	64.61%	56.69%	64.61%	54.92%	46.13%	67.78%	59.12%
	3	64.08%	58.63%	67.42%	64.26%	54.05%	67.96%	62.73%
	5	64.96%	59.51%	67.42%	65.15%	51.41%	67.61%	62.68%
	10	64.79%	59.33%	67.42%	65.49%	53.12%	68.31%	63.08%
	15	64.96%	60.39%	67.42%	65.85%	53.52%	67.96%	63.35%
	20	65.14%	60.74%	66.02%	65.49%	52.29%	68.84%	63.09%
CAST-F	1	64.08%	61.09%	67.96%	59.96%	35.72%	68.13%	59.52%
	3	64.61%	61.09%	68.31%	64.44%	54.74%	68.13%	63.58%
	5	65.32%	61.62%	68.84%	65.15%	55.77%	69.01%	64.28%
	10	65.14%	60.92%	68.49%	66.19%	56.16%	69.01%	64.32%
	15	65.84%	61.09%	68.66%	66.37%	57.22%	69.19%	64.73%
	20	66.02%	61.27%	69.37%	66.55%	**58.10%**	**69.37%**	65.11%
Average #Shots	11.43	15.65	8.08	13.98	9.57	7.37	13.94	11.43
CAST-A ($\geq$ 98%)		**67.25%**	**61.80%**	**71.13%**	**68.31%**	56.16%	**69.37%**	**65.67%**

TABLE V
RESULTS OF THE GAIN OF CAST-F AMONG DIFFERENT SIZES OF LLMS, CONSISTING OF 2B, 9B, AND 27B OF GEMMA2.

Models	Strategy	C++ to Python		Python to C++		Java to C++		C++ to Java		Average	
		CA	EM	CA	EM	CA	EM	CA	EM	CA	EM
Gemma2-2B	Fixed	47.36%	6.51%	39.61%	2.64%	54.05%	12.15%	22.18%	5.28%	40.80%	6.65%
	CAST-F	51.06%	8.10%	48.77%	3.17%	63.56%	17.43%	30.81%	10.39%	48.55%	9.77%
	Improv.	7.81%	24.42%	**23.13%**	20.08%	**17.59%**	**43.46%**	38.91%	**96.78%**	21.86%	46.18%
Gemma2-9B	Fixed	60.39%	8.45%	59.26%	4.75%	65.61%	31.34%	20.81%	23.06%	51.52%	16.90%
	CAST-F	60.74%	14.08%	64.08%	9.51%	69.19%	36.27%	65.14%	32.22%	64.79%	23.02%
	Improv.	0.58%	**66.63%**	8.13%	**100.21%**	5.46%	15.73%	**213.02%**	39.72%	**56.80%**	**55.57%**
Gemma2-27B	Fixed	63.38%	8.45%	63.91%	5.33%	68.49%	24.96%	68.31%	21.83%	66.02%	15.14%
	CAST-F	70.60%	11.62%	68.66%	6.87%	71.13%	34.86%	72.49%	28.87%	70.72%	20.56%
	Improv.	**11.39%**	37.51%	7.43%	28.89%	3.85%	39.66%	6.12%	32.25%	7.20%	34.58%

performance remains stable across all values of k for varying t. For CAST-A, there is an additional threshold τ. Results in Fig. 3 indicate that CAST-A achieves Pareto optimality when τ varies between 90% and 98%, demonstrating stable and effective performance across different thresholds.

Finding 7: Both CAST-F and CAST-A demonstrate strong robustness and stability across varying hyperparameter settings.

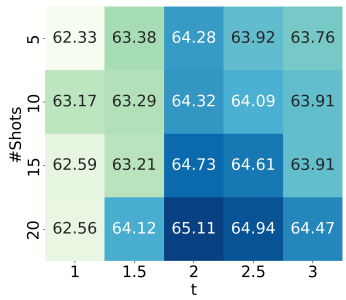


Fig. 5. Sensitivity analysis of hyper-parameters in the average CA of Qwen2-7B for the selection size k and the pre-recalled candidate size $\lfloor t \cdot k \rfloor$.

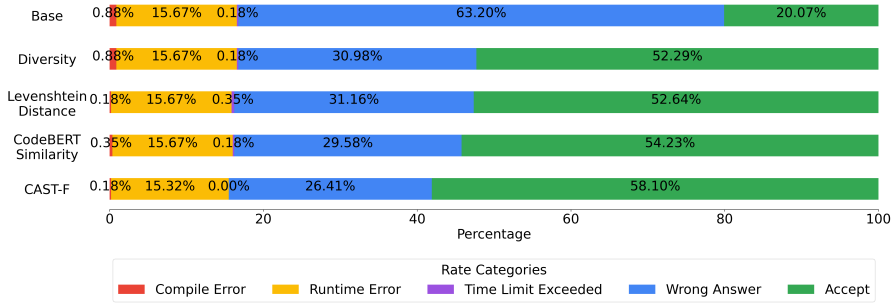


Fig. 6. The distribution of result types in Python-to-Java 20-shots translation by Qwen2-7B across various exemplar selection strategies in ICL. Outcomes are categorized as Compile Error, Runtime Error, Time Limit Exceeded, Wrong Answer, and Accept.

TABLE VI
RESULTS OF QWEN2-7B FOR 5-SHOT CODE SUMMARIZATION UPON DIFFERENT ICL STRATEGIES.

Strategy	Java		Python		PHP		Go		Javascript		Ruby	
	BLEU	Rouge-L	BLEU	Rouge-L	BLEU	Rouge-L	BLEU	Rouge-L	BLEU	Rouge-L	BLEU	Rouge-L
Random	2.25	9.01	5.29	14.80	12.30	10.78	4.60	14.93	2.33	5.45	5.02	13.13
BM25	2.27	9.60	5.92	14.74	**17.65**	16.55	4.48	14.24	2.86	6.03	5.49	12.50
LD	2.06	9.39	4.47	13.45	16.27	16.14	5.43	15.63	3.11	7.83	4.82	13.00
CodeBERT	**2.43**	9.56	5.92	15.07	17.41	15.05	4.34	13.35	2.80	5.87	5.99	13.14
Diversity	2.30	9.39	5.23	13.00	14.52	13.40	3.88	12.80	2.43	5.86	5.25	13.34
CAST-F	2.42	**9.90**	**6.73**	**16.57**	17.58	**18.65**	**5.52**	**17.68**	**4.81**	**9.12**	**6.71**	**14.47**

RQ8: What is the impact of our method on failure frequencies?

This study evaluates how exemplar selection strategies influence test code failures across various input cases. Results in Fig. 6 highlight key observations: 1) CAST-F achieves the lowest proportion of "Wrong Answer" outcomes (26.41%) and the highest "Accept" rate (58.10%), surpassing Levenshtein Distance (52.64%) and CodeBERT Similarity (54.23%). This underscores its superior effectiveness in selecting exemplars to enhance translation accuracy. 2) ICL yields limited improvements for "Compile Error" and "Runtime Error." These limitations likely stem from cross-language differences and the lack of comprehensive third-party library knowledge, which hinder LLMs' ability to resolve such errors.

Finding 8: CAST-F effectively reduces failure frequencies, achieving higher accuracy and robustness compared to baseline strategies, particularly in minimizing "Wrong Answer" outcomes and maximizing "Accept" rates.

RQ9: Could CAST be extended to support programming languages and tasks beyond code translation?

To showcase the broader applicability of CAST beyond code translation, we conduct an additional experiment on code summarization. Code summarization generates natural language descriptions from source code snippets, a crucial task in software maintenance and comprehension. We use the widely used dataset provided by the CodeXGLUE [36] team for this task. We evaluate the performance of our method, using the widely adopted Qwen2-7B model across six programming languages: Java, Python, PHP, Go, JavaScript, and Ruby. The evaluation metrics include BLEU [45] and Rouge-L scores, assessing the quality of generated summaries against reference descriptions. Rouge-L computes an F-score based on the longest common subsequence, comparing the similarity between two texts.

Tab. VI presents results comparing CAST-F with baseline exemplar selection strategies. CAST-F achieves the highest BLEU and Rouge-L scores across nearly all languages and metrics, demonstrating its superior ability to select exemplars that enhance the model's summarization quality. This consistent improvement underscores that CAST's structural coverage principle captures critical code characteristics essential for generating accurate and informative summaries. These findings emphasize CAST's generality and robustness as an exemplar selection metric. By capturing structural coverage effectively, CAST facilitates improved task-specific knowledge transfer for LLMs in code-related tasks.

Finding 9: The experiment on code summarization broadens the scope of our approach beyond code translation, establishing CAST as a versatile tool for enhancing code retrieval and generation tasks in various software engineering scenarios.

RQ10: How do different database sizes and noisy data affect the method's performance?

We evaluated the impact of database size variations and noise injection on CAST performance using the TransCoder dataset Java-Python task under Qwen-7B. To assess database size effects, we randomly sampled subsets at 75%, 50%, and 25% of the full database as ICL demonstration exemplars, simulating scenarios with limited available data. For noise injection, we implemented three perturbation types to simulate real-world code imperfections:

- *Syntax Error*: Random deletion or replacement of punctuation marks (semicolons, parentheses, quotes), introducing minor cross-language syntax errors.

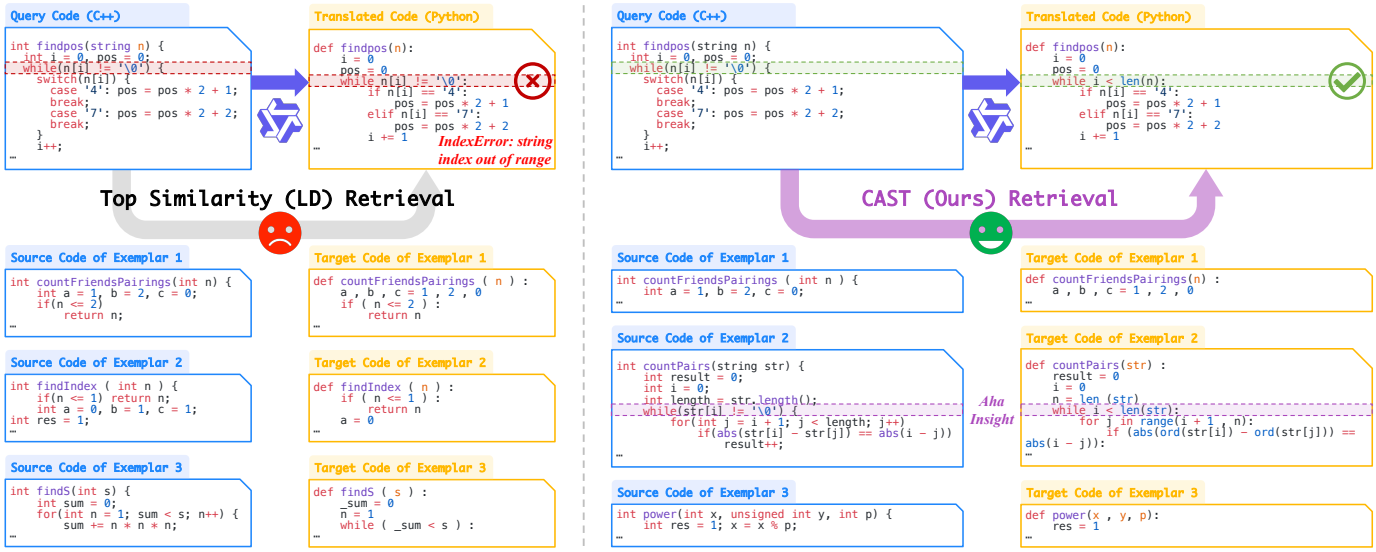


Fig. 7. A case study (qualitative analysis) of code translation comparing similarity-only method (i.e., LD) and our CAST-F, based on Qwen2-7B.

- *Identifier Rename*: Variable name obfuscation or random replacement, breaking identifier consistency and simulating naming errors.
- *Statement Shuffle*: Random reordering of consecutive statements within code snippets, mimicking structural disorder in semantically related blocks.

As shown in Fig. 8a, reducing the database size for exemplar sampling leads to gradual accuracy degradation. Performance decreases from 69.01% (full database) to 67.08% (25% sampling), indicating that while CAST maintains robustness under reduced exemplar availability, performance diminishes as database information decreases. Fig. 8b demonstrates performance degradation under different noise types. Accuracy drops from 69.01% (no noise) to 66.20% (statement shuffle), with syntax errors and identifier renaming showing intermediate impact (67.25% and 67.08% respectively). These results indicate CAST's relative resilience to syntax and naming perturbations but higher sensitivity to structural disruptions.

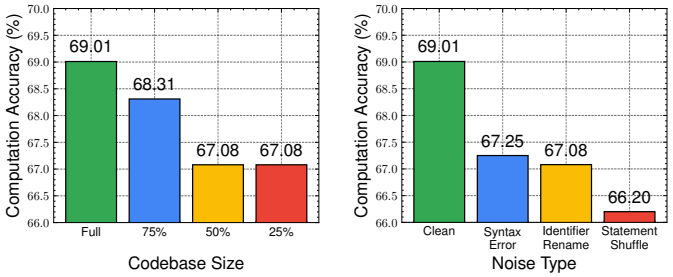


Fig. 8. Impact of database characteristics on computation accuracy. Fig. 8a Database size effect with random sampling. Fig. 8b Performance under different noisy database.

Finding 10: CAST performance degrades with reduced database size (2.93% drop at 25% sampling) and structural noise (2.81% loss with statement shuffle). Adequate database scale and code quality are essential for optimal performance.

V. Case Study

To understand why traditional similarity-based retrieval fails to provide optimal exemplars for code translation, we conduct a case study examining how different retrieval strategies affect translation quality. The fundamental limitation of similarity-only approaches is their tendency to retrieve exemplars with high similarity while overlooking sufficient knowledge required for the current translation task. Fig. 7 shows this through a C++ to Python translation task about `findpos` function that processes string-based position calculations.

The similarity-only retrieval (LD) operates purely on lexical similarity, leading to the selection of highly correlated but redundant exemplars that cannot provide sufficient knowledge for the current translation task. The case shows that the translated code fails in string boundary checking, resulting in an `IndexError: string index out of range`, where correctly translating this construct requires knowledge of the difference between C++ null-terminator-based string handling (`'\0'`) and Python's length-based indexing (`len()`). However, the retrieved exemplars fail to provide this critical cross-language knowledge for properly handling the string termination construct `while (n[i] != '\0')`.

In contrast, CAST retrieval functions as a surrogate objective for information coverage across syntactic structures, actively seeking exemplars that provide the knowledge required for the current translation task. By retrieving exemplars that collectively cover diverse syntactic structural patterns, CAST enables the model to discover the "aha insight" that Python requires length-based indexing (`while i < len(n)`). This demonstrates how CAST's information coverage principle guides the model toward semantically correct translations by ensuring comprehensive knowledge representation in the retrieved exemplar set. The success stems from CAST's ability to prioritize knowledge completeness over superficial similarity,

leading to more robust code translation.

VI. Threat Analysis

Our results are interpreted with three threats to validity.

- The internal validity threat comes from potential LLM data leakage due to possible training/testing set overlaps, but prior work [34] shows this threat is limited.
- The first external validity threat is in the implementation of the compared techniques. To reduce it, we select state-of-the-art pre-trained models and reuse their implementations and weights. We believe this threat is minimal.
- Another external validity threat is the quality of the dataset. It was released by [31] and is widely used due to multi-lingual samples and self-contained test suites [32]–[34]. We use the cleaned dataset from [34], where four authors cross-checked their cleaned results in pairs to reach a consensus for each group of parallel functions.
- The construct threat lies in the evaluation metrics. To reduce this threat, we adopt CA and EM to evaluate the code translation performance of different approaches from both lexical and semantic correctness as most of the relevant papers [31]–[34].

VII. Related Works

A. Code translation

Automated code translation aids in the migration of codebases across programming languages, significantly enhancing productivity in software development. Early rule-based translation methods have been extensively explored [47]–[50]. Furthermore, phrase-based statistical machine translation techniques have shown notable advances [51]–[56]. Due to the success of deep learning, deep code translation methods have extended the SOTA further [57]–[62].

Prior research has shown that, unlike the grammatical structure of natural language, the syntactic structure of source code is stricter and more intricate, directly reflecting the program's behavior [9], [12], [63]–[65]. Previous studies have demonstrated that explicitly incorporating structural knowledge into models enhances their perceptual understanding of code, thereby aiding software mining tasks [14], [15], including code translation [9], [13]. For example, modeling structural knowledge from AST [8], [10], [66], [67], control flow graphs (CFG) [11], [12], and data flow graphs (DFG) [13]. However, these syntax-aware methods rely on meticulously crafted model architectures and loss functions that are tailored through training.

Recently, LLMs, scaled to hundreds of billions of parameters, have exhibited remarkable performance across various natural language processing tasks [68]–[70], such as GPT [43], [71], LLaMA [41], Qwen [38], etc. The LLM-powered code translation has achieved significant progress in recent years [34], [72]–[74]. For example, Yang et al. [34] proposed UniTrans, which introduces auto-generated test cases throughout the whole code translation framework via translation augmentation and repair. Although UniTrans injects test information and achieves certain effectiveness, it overlooks the role of intrinsic code structure information in assisting the model to understand code semantics. In fact, mainstream methods based on LLMs often overlook code structural knowledge, whose effectiveness has been validated in prior studies [11], [12], [65]. Directly combining the carefully designed model architectures and loss functions of classic syntax-aware methods with LLMs is both time- and resource-intensive, and may even be prohibited for proprietary models. Therefore, research on how to post-incorporate code structural knowledge into off-the-shelf LLMs is limited but crucial.

B. In-context Learning

Recent studies have shown that LLMs exhibit strong ICL abilities as model and data sizes scale up [75]–[77]. The essence of ICL lies in learning by analogy, requiring a prompt context composed of a few demonstration exemplars. By concatenating these exemplars with a query, LLMs can make better predictions without updating parameters, differentiating ICL from traditional model tuning. Recently, numerous studies have attempted to leverage ICL for complex tasks, including natural language understanding [17], reasoning [78]–[80], text generation [25], [26], [81]. Additionally, several studies aim to provide theoretical justifications and insights into how LLMs learn from a few in-context demonstration exemplars [82], [83]. Despite significant progress, ICL still faces challenges and requires further exploration. such as its sensitivity to the selection of demonstration exemplars [84].

Various objectives for ICL exemplar selection and tailoring have been explored [16], [19], [85], [86]. Mostly ICL exemplar selection typically relies on point-wise similarity [17]–[19], [66], [67], where a matching score is computed between each candidate exemplar and the input query, followed by the independent selection of the top-k exemplars to construct the context [16]. Although these methods are straightforward and fast, they fail to consider compositional relationships among the selected k exemplars, often leading to redundancy caused by homogeneity. Some studies aim to ensure diversity by clustering exemplars into k groups, then selecting the most similar exemplars from each group for a given query [20]–[22]. Besides, some works [23], [24] introduce an auxiliary loss to regularize diversity. However, additional diversity loss may hinder the optimization of relevance, and accounting for compositional relationships in diversity remains challenging. Some methods attempt to combine weighted similarity and diversity, but adaptively balancing these two objectives is difficult. Moreover, these methods do not focus on code translation and are typically syntax-unaware, neglecting the critical syntactic information inherent in source code. Additionally, [24] attempts to account for local structure coverage in ICL. However, it only considers the structure between a parent node and its child nodes, and the algorithm is rough, needing to work together with complex similarity and diversity regularization losses. In contrast, our method is grounded in information theory, providing greater precision and generality. We consider a more comprehensive subtree structure, and our algorithm can be applied directly, guaranteed by submodular theory, without requiring additional losses.

VIII. Conclusion and Future Work

Code translation aids code migration between languages, boosting productivity in software development. This work focuses on post-incorporating code structural knowledge into pre-trained models via ICL for code translation. We revisit exemplar selection in ICL from an information-theoretic perspective, showing that maximizing information coverage provides a more precise and general solution than traditional methods. We introduce CAST, a surrogate measure based on the maximum subtree coverage ratio of AST. We show that NP-hard CAST maximization is a submodular maximization problem and propose a greedy algorithm with a $(1 - 1/e)$-approximate solution of polynomial time complexity. Our method is the first training-free, model-agnostic approach to integrate code structural knowledge into existing LLMs at test time, and both theory and experiments prove their effectiveness and efficiency. This work provides two key insights: 1) Code structural knowledge can be post-incorporated into pre-trained LLMs during inference despite training neglect. 2) Scaling model size or data doesn't lead to code structural knowledge, highlighting the need to consider code syntax in LLMs.

Although our method shows promise in incorporating code structural knowledge into pre-trained models, several areas for future improvement remain. For instance, considering more advanced code structure beyond AST, e.g., CFG and DFG, could provide richer information for software mining in pretrained models. Besides, introducing a feedback loop during inference, where the model refines exemplar selection based on previous outputs, could further enhance performance.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 62076121 and 61921006, the Major Program (JD) of Hubei Province (2023BAA024), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX24_0301).

References

[1] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, *et al.*, "DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[2] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[3] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, *et al.*, "Code Llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[4] R. Li, L. B. allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. LI, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, J. Lamy-Poirier, J. Monteiro, N. Gontier, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. T. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, U. Bhattacharyya, W. Yu, S. Luccioni, P. Villegas, F. Zhdanov, T. Lee, N. Timor, J. Ding, C. S. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. V. Werra, and H. de Vries, "StarCoder: may the source be with you!," *Transactions on Machine Learning Research*, 2023. Reproducibility Certification.

[5] R.-B. Liu, A. Li, C. Yang, H. Sun, and M. Li, "Revisiting Chain-of-Thought in code generation: Do language models need to learn reasoning before coding?," in *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

[6] X.-Y. Li, Y. Du, and M. Li, "Enhancing LLMs in long code translation through instrumentation and program state alignment," *arXiv preprint arXiv:2504.02017*, 2025.

[7] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, (New York, NY, USA), 2024.

[8] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "TreeBERT: A tree-based pre-trained model for programming language," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, pp. 54–63, 2021.

[9] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," *Advances in neural information processing systems*, vol. 31, 2018.

[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, Jan. 2019.

[11] Y.-F. Ma, Y. Du, and M. Li, "Capturing the long-distance dependency in the control flow graph via structural-guided attention for bug localization," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, pp. 2242–2250, 2023.

[12] Y. Du and Z. Yu, "Pre-training code representation with semantic flow graph for effective bug localization," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (New York, NY, USA), p. 579–591, 2023.

[13] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *9th International Conference on Learning Representations, Virtual Event, Austria*, 2021.

[14] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, "ATOM: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1800–1817, 2022.

[15] L. Gong, M. Elhoushi, and A. Cheung, "AST-T5: Structure-aware pretraining for code generation and understanding," in *Proceedings of the 41st International Conference on Machine Learning*, pp. 15839–15853, 2024.

[16] M. Luo, X. Xu, Y. Liu, P. Pasupat, and M. Kazemi, "In-context learning with retrieved demonstrations for language models: A survey," *Transactions on Machine Learning Research*, 2024. Survey Certification.

[17] J. Liu, D. Shen, Y. Zhang, B. Dolan, L. Carin, and W. Chen, "What makes good in-context examples for GPT-3?," in *Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures*, (Dublin, Ireland and Online), pp. 100–114, 2022.

[18] F. Gao, Q. Ping, G. Thattai, A. Reganti, Y. N. Wu, and P. Natarajan, "Transform-retrieve-generate: Natural language-centric outside-knowledge visual question answering," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5067–5077, 2022.

[19] M. Luo, Y. Zeng, P. Banerjee, and C. Baral, "Weakly-supervised visual-retriever-reader for knowledge-based question answering," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, (Online and Punta Cana, Dominican Republic), pp. 6417–6431, 2021.

[20] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context llms struggle with long in-context learning," *arXiv preprint arXiv:2404.02060*, 2024.

[21] J. Li, J. Wang, Z. Zhang, and H. Zhao, "Self-prompting large language models for zero-shot open-domain QA," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, (Mexico City, Mexico), pp. 296–310, June 2024.

[22] X. Li and X. Qiu, "Mot: Pre-thinking and recalling enable chatgpt to self-improve with memory-of-thoughts," *CoRR*, 2023.

[23] J. Ye, Z. Wu, J. Feng, T. Yu, and L. Kong, "Compositional exemplars for in-context learning," in *Proceedings of the 40th International Conference on Machine Learning*, pp. 39818–39833, 2023.

[24] I. Levy, B. Bogin, and J. Berant, "Diverse demonstrations improve in-context compositional generalization," in *Proceedings of the 61st Annual*

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), (Toronto, Canada), pp. 1401–1422, 2023.

[25] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," in *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (Seattle, United States), pp. 2655–2671, 2022.

[26] S. Agrawal, C. Zhou, M. Lewis, L. Zettlemoyer, and M. Ghazvininejad, "In-context examples selection for machine translation," in *Findings of the Association for Computational Linguistics: ACL 2023*, (Toronto, Canada), pp. 8857–8873, 2023.

[27] B. Dalvi Mishra, O. Tafjord, and P. Clark, "Towards teachable reasoning systems: Using a dynamic memory of user feedback for continual system improvement," in *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, (Abu Dhabi, United Arab Emirates), pp. 9465–9480, 2022.

[28] B. Wang, S. Min, X. Deng, J. Shen, Y. Wu, L. Zettlemoyer, and H. Sun, "Towards understanding chain-of-thought prompting: An empirical study of what matters," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Toronto, Canada), pp. 2717–2739, July 2023.

[29] G. L. Nemhauser and L. A. Wolsey, "Best algorithms for approximating the maximum of a submodular set function," *Mathematics of operations research*, vol. 3, no. 3, pp. 177–188, 1978.

[30] H. Sun, S. Lu, H. Wang, Q.-G. Chen, Z. Xu, W. Luo, K. Zhang, and M. Li, "MDP3: A training-free approach for list-wise frame selection in video-LLMs," *arXiv preprint arXiv:2501.02885*, 2025.

[31] B. Rozière, M. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *Advances in Neural Information Processing Systems*, 2020.

[32] B. Roziere, J. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," in *International Conference on Learning Representations*, 2022.

[33] M. Szafraniec, B. Roziere, H. J. Leather, P. Labatut, F. Charton, and G. Synnaeve, "Code translation with compiler representations," in *The Eleventh International Conference on Learning Representations*, 2023.

[34] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, 2024.

[35] W. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," in *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 2268–2281, 2023.

[36] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, virtual*, 2021.

[37] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang, G. Dong, H. Wei, H. Lin, J. Tang, J. Wang, J. Yang, J. Tu, J. Zhang, J. Ma, J. Xu, J. Zhou, J. Bai, J. He, J. Lin, K. Dang, K. Lu, K.-Y. Chen, K. Yang, M. Li, M. Xue, N. Ni, P. Zhang, P. Wang, R. Peng, R. Men, R. Gao, R. Lin, S. Wang, S. Bai, S. Tan, T. Zhu, T. Li, T. Liu, W. Ge, X. Deng, X. Zhou, X. Ren, X. Zhang, X. Wei, X. Ren, Y. Fan, Y. Yao, Y. Zhang, Y. Wan, Y. Chu, Z. Cui, Z. Zhang, and Z.-W. Fan, "Qwen2 technical report," *arXiv preprint arXiv:2407.10671*, 2024.

[38] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, *et al.*, "Qwen2.5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

[39] G. Team, M. Riviere, S. Pathak, P. G. Sessa, C. Hardin, S. Bhupatiraju, L. Hussenot, T. Mesnard, B. Shahriari, A. Ramé, *et al.*, "Gemma 2: Improving open language models at a practical size," *arXiv preprint arXiv:2408.00118*, 2024.

[40] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma, *et al.*, "DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence," *arXiv preprint arXiv:2406.11931*, 2024.

[41] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[42] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[43] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[44] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, pp. 1245–1262, 1989.

[45] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, (USA), p. 311–318, 2002.

[46] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[47] K. Alsubhi, F. Alsolami, A. Algarni, E. Albassam, M. Khemakhem, F. Eassa, K. Jambi, and M. U. Ashraf, "A tool for translating sequential source code to parallel code written in c++ and openacc," in *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*, pp. 1–8, 2019.

[48] K. An, N. Meng, and E. Tilevich, "Automatic inference of java-to-swift translation rules for porting mobile applications," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 180–190, 2018.

[49] S. Pino, L. Pollock, and S. Chandrasekaran, "Exploring translation of OpenMP to OpenACC 2.5: lessons learned," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 673–682, 2017.

[50] M. Tauda, Z. Zainuddin, and Z. Tahir, "Programming language translator for integration client application with web apis," in *2021 International Conference on Artificial Intelligence and Mechatronics Systems (AIMS)*, pp. 1–4, 2021.

[51] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, (New York, NY, USA), p. 173–184, 2014.

[52] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-Conquer approach for multi-phase statistical migration for source code (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 585–596, 2015.

[53] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, (New York, NY, USA), p. 651–654, 2013.

[54] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 756–758, 2016.

[55] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[56] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pp. 574–584, 2015.

[57] G. Lample, A. Conneau, L. Denoyer, and M. Ranzato, "Unsupervised machine translation using monolingual corpora only," in *International Conference on Learning Representations*, 2018.

[58] Y. Wen, Q. Guo, Q. Fu, X. Li, J. Xu, Y. Tang, Y. Zhao, X. Hu, Z. Du, L. Li, C. Wang, X. Zhou, and Y. Chen, "BabelTower: Learning to auto-parallelized program translation," in *Proceedings of the 39th International Conference on Machine Learning*, pp. 23685–23700, 2022.

[59] J. D. Weisz, M. Muller, S. Houde, J. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? human-ai partnerships in code translation," in *Proceedings of the 26th International Conference on Intelligent User Interfaces*, pp. 402–412, 2021.

[60] M. Zhu, K. Suresh, and C. K. Reddy, "Multilingual code snippets training for program translation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, pp. 11783–11790, Jun. 2022.

[61] Y. Du, Y.-F. Ma, Z. Xie, and M. Li, "Beyond lexical consistency: Preserving semantic consistency for program translation," in *2023 IEEE International Conference on Data Mining*, pp. 91–100, 2023.

[62] Y. Du, H. Sun, and M. Li, "A joint learning model with variational interaction for multilingual program translation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1907–1918, 2024.

[63] X. Huo, Y. Yang, M. Li, and D.-C. Zhan, "Learning semantic features for software defect prediction by code comments embedding," in *2018 IEEE International Conference on Data Mining*, pp. 1049–1054, 2018.
[64] X. Luo, J. Wu, C. Zhou, X. Zhang, and Y. Wang, "Deep semantic network representation," in *2020 IEEE International Conference on Data Mining*, pp. 1154–1159, 2020.
[65] L. Chai and M. LI, "Pyramid attention for source code summarization," in *Advances in Neural Information Processing Systems*, pp. 20421–20433, 2022.
[66] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.
[67] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, (New York, NY, USA), p. 298–309, 2018.
[68] OpenAI, "ChatGPT: Optimizing language models for dialogue." https://openai.com/blog/chatgpt, 2023.
[69] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
[70] A. Anthropic, "Introducing the next generation of Claude." https://www.anthropic.com/news/claude-3-family, 2024.
[71] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "ChatGPT vs SBST: A comparative assessment of unit test suite generation," *IEEE Trans. Softw. Eng.*, vol. 50, p. 1340–1359, Mar. 2024.
[72] M. Macedo, Y. Tian, F. Cogo, and B. Adams, "Exploring the impact of the output format on the evaluation of large language models for code translation," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, pp. 57–68, 2024.
[73] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, "Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution," in *ECAI 2024*, pp. 4011–4018, IOS Press, 2024.
[74] Y. Luo, R. Yu, F. Zhang, L. Liang, and Y. Xiong, "Bridging gaps in llm code translation: Reducing errors with call graphs and bridged debuggers," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 2448–2449, 2024.
[75] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "PaLM: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
[76] S. Liu, H. Ye, L. Xing, and J. Zou, "In-context vectors: Making in context learning more effective and controllable through latent space steering," *arXiv preprint arXiv:2311.06668*, 2023.
[77] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022. Survey Certification.
[78] J. Wei, X. Wang, D. Schuurmans, M. Bosma, brian ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-Thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, 2022.
[79] P. Lu, L. Qiu, K.-W. Chang, Y. N. Wu, S.-C. Zhu, T. Rajpurohit, P. Clark, and A. Kalyan, "Dynamic prompt learning via policy gradient for semi-structured mathematical reasoning," in *The Eleventh International Conference on Learning Representations*, 2023.
[80] A. Scarlatos and A. Lan, "RetICL: Sequential retrieval of in-context examples with reinforcement learning," *arXiv preprint arXiv:2305.14502*, 2023.
[81] P. Shi, R. Zhang, H. Bai, and J. Lin, "XRICL: Cross-lingual retrieval-augmented in-context learning for cross-lingual text-to-SQL semantic parsing," in *Findings of the Association for Computational Linguistics: EMNLP 2022*, (Abu Dhabi, United Arab Emirates), pp. 5248–5259, 2022.
[82] J. Von Oswald, E. Niklasson, E. Randazzo, J. Sacramento, A. Mordvintsev, A. Zhmoginov, and M. Vladymyrov, "Transformers learn in-context by gradient descent," in *Proceedings of the 40th International Conference on Machine Learning*, pp. 35151–35174, 2023.
[83] S. Garg, D. Tsipras, P. S. Liang, and G. Valiant, "What can transformers learn in-context? a case study of simple function classes," *Advances in Neural Information Processing Systems*, vol. 35, pp. 30583–30598, 2022.
[84] Y. Liu, J. Liu, X. Shi, Q. Cheng, Y. Huang, and W. Lu, "Let's learn step by step: Enhancing in-context learning ability with curriculum learning," *arXiv preprint arXiv:2402.10738*, 2024.
[85] X. Li, K. Lv, H. Yan, T. Lin, W. Zhu, Y. Ni, G. Xie, X. Wang, and X. Qiu, "Unified demonstration retriever for in-context learning," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, (Toronto, Canada), pp. 4644–4668, 2023.
[86] D. Cheng, S. Huang, J. Bi, Y. Zhan, J. Liu, Y. Wang, H. Sun, F. Wei, W. Deng, and Q. Zhang, "UPRISE: Universal prompt retrieval for improving zero-shot evaluation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, (Singapore), pp. 12318–12337, 2023.

Yali Du received the BEng degree from the School of Computer Science and Technology at Shandong University, China, in 2022. She is currently a PhD student at the School of Artificial Intelligence at Nanjing University and a member of the LAMDA Group. Her research interests include machine learning and data mining, especially on software mining.

Hui Sun received the BEng degree from the College of Software Engineering at Jilin University, China, in 2019, and the MS degree from the School of Artificial Intelligence at Nanjing University, China, in 2022. He is currently a PhD student in the School of Artificial Intelligence at Nanjing University and is a member of the LAMDA Group. His research interests include transfer learning, semi-supervised learning, and multimodal large language model.

Ming Li (Member, IEEE) is currently a professor with the LAMDA group, the National Key Laboratory for Novel Software Technology, Nanjing University. His major research interests include machine learning and data mining, especially on software mining. He has served as the area chair of IJCAI, IEEE ICDM, etc, senior PC member of the premium conferences in artificial intelligence such as AAAI, and PC members for other premium conferences such as KDD, NeurIPS, ICML, etc. He is the founding chair of the International Workshop on Software Mining. He has been granted various awards including the PAKDD Early Career Award, the NSFC Excellent Youth Award, the New Century Excellent Talents program of the Education Ministry of China, etc.