AN INTERPRETABLE ERROR CORRECTION METHOD FOR ENHANCING CODE-TO-CODE TRANSLATION

Min Xue **Artur Andrzejak*** **Marla Leuther**
Heidelberg University, Germany
{min.xue, artur.andrzejak, Leuther}@uni-heidelberg.de

ABSTRACT

Transformer-based machine translation techniques currently dominate the field of program translation. However, these models pose challenges in explaining program translations. Moreover, researchers frequently invest substantial time and computational resources in retraining models, yet the improvement in translation accuracy is quite limited. To address these issues, we introduce a novel approach, kNN-ECD, which combines k-nearest-neighbor search with a key-value error correction datastore to overwrite the wrong translations of TransCoder-ST (Roziere et al., 2022). This provides a decision-making basis for interpreting the corrected translations. Building upon this, we further propose kNN-ECS$_m$, a methodology that employs a distributed structure with m sub-datastores connected in series, utilizing m diverse experts for multi-round error correction. Additionally, we put forward a *unified name rule*, encouraging the datastore to focus more on code logic and structure rather than diverse rare identifiers. Our experimental results show that our approach improves the translation accuracy from 68.9% to 89.9% of TransCoder-ST (for translation from Java to Python). This error correction method augments program translation, overcoming the inherent limitations of Transformer-based code translation models, such as resource-intensive retraining requirements and uninterpretable outcomes.

1 INTRODUCTION

Transformer-based large language models (LLMs), such as BERT (Devlin et al., 2018), XLM (Lample & Conneau, 2019) and XLNet (Yang et al., 2019), have been widely used in the field of program translation. In the latest work, researchers have attempted to enhance Transformer-based program translation by incorporating compiler intermediate representations, but the improvements remain limited (Szafraniec et al., 2023; Rozière et al., 2021). These methods leverage large amounts of data collected from public repositories such as GitHub and GitLab, combining unsupervised and self-supervised learning to overcome the need for parallel corpora. Nevertheless, these techniques often require substantial investments in computational resources for retraining translation models, while yielding only marginal improvements in return. Compared to retraining the LLMs, **Error Correction** emerges as a more efficient alternative, capable of enhancing accuracy through repairing wrong translations produced by the program translation model. This approach holds significant practical value, making it possible to enhance program translation accuracy at a low cost.

The internal workings of Transformer-based models are relatively complex, making it challenging to track and identify which snippets in the training dataset contribute to each output token. In other words, the Transformer-based program translation model cannot provide an intuitive interpretation for the output. A common approach is to exploit knowledge neurons and causal effects to explain the output of the transformer model (Dai et al., 2022; Vig et al., 2020). In the latest research, Meng et al. (2023) revealed the crucial role of middle-layer feed-forward modules in storing factual associations, but it has been so far applied only to simple subject-predicate-object sentences. In contrast, the k-nearest-neighbor machine translation (kNN-MT) combined with a large-scale datastore has recently demonstrated a remarkable capacity in optimizing and interpreting natural language translations (Khandelwal et al., 2020; Zhang et al., 2018; Tu et al., 2017). Compared with traditional

*Corresponding author.

Transformer-based models (Vaswani et al., 2017), datastore possesses inherent interpretability. The kNN retrieval method can provide a clear inference path and decision-making basis through tracking and identifying which snippet in the training dataset contribute to each generated token, without the need to analyze complex neural network hierarchies. Based on this, we consider integrating the kNN-MT method with error correction, which exhibits great potential in providing an interpretable code correction analysis.

In modern source code datasets, among millions of unique identifiers, only less than 1% of the identifiers appear frequently (Karampatsis et al., 2020). Diverse rare identifiers, such as function names, variable names, and parameter names, often cause perturbations during the model training and inference phases (Chirkova & Troshin, 2020b). This phenomenon is akin to introducing noise into the process of information transmission, making it difficult to focus on the key information. Notably, although these different rare identifiers tend to introduce noise in program translation and error correction, only a few researchers have paid attention to this issue. For this problem, we propose a *unified name rule*, which replaces diverse rare identifiers according to the homogeneity rule (as shown in Figure 2) during the training and testing phases, minimizing the emphasis on rare identifiers and focusing more on code logic and structure.

Our work builds upon the Transformer-based code translation models proposed in the TransCoder (Roziere et al., 2020) and TransCoder-ST (Roziere et al., 2022) projects. TransCoder employed self-supervised learning across multiple programming languages (between Java, C++, and Python), and then TransCoder-ST extended it by introducing a rigorously tested parallel corpus. In this paper, we propose to extract error correction information from TransCoder-ST to guide the error correction model in learning repair knowledge. More specifically, we create unit tests for the Java source dataset, and then use TransCoder-ST to generate multiple Python functions with unit tests for each Java function. Subsequently, we conduct unit testing on the Python functions, and then extract the first failed Python function and the first successful Python function to form an error correction language pair for each Java function. Based on this, we establish two alternative error correction models, kNN-ECD and kNN-ECS$_m$, to improve the translation accuracy of TransCoder-ST by correcting wrong translations. Overall, our contributions are as follows:

- We propose kNN-ECD, which leverages kNN search to retrieve correction information from the error correction datastore, thereby repairing wrong translations in TransCoder-ST. Notably, the error correction process is interpretable, overcoming the opaque defects of Transformer-based program translation.
- We introduce kNN-ECS$_m$, an approach that employs a distributed structure with m small datastore units, capturing comprehensive repair information by using multiple datastore variants for multi-round error correction. This approach promotes the traditional kNN-MT technique, effectively improving the data retrieval capabilities by trying diverse data flows.
- We present a *unified name rule*, which ignores the differences in rare identifiers and focuses more on the logic and structure of the code, thus avoiding interference caused by irrelevant identifiers on similarity retrieval.
- We evaluate our approach, showing that the translation accuracy of TransCoder-ST (Java→Python) significantly improves from 68.9% to 89.9% after employing kNN-ECD/ kNN-ECS$_m$. Without the necessity of retraining Transformer-based models, our approach still achieves great advancements in program translation.

2 Related Work

Transformer-based Program Translation. In recent years, Transformer-based unsupervised learning methods have become the mainstream approach in the field of program translation (Shi et al., 2022; Zhang et al., 2023). TransCoder (Roziere et al., 2020), as the first work that combined an unsupervised model with programming translation, pioneered automatic program translation in the field of software development. Benefitting from the structural characteristics of programming languages, Rozière et al. (2021) introduced a new pre-training objective, DOBF, to recover the original version of the obfuscated source code by pre-training a model. Based on this, Roziere et al. (2022) developed TransCoder-ST, which utilizes an automated unit testing system to filter out invalid translations, and then uses a well-tested parallel corpus to fine-tune the unsupervised model (Radford

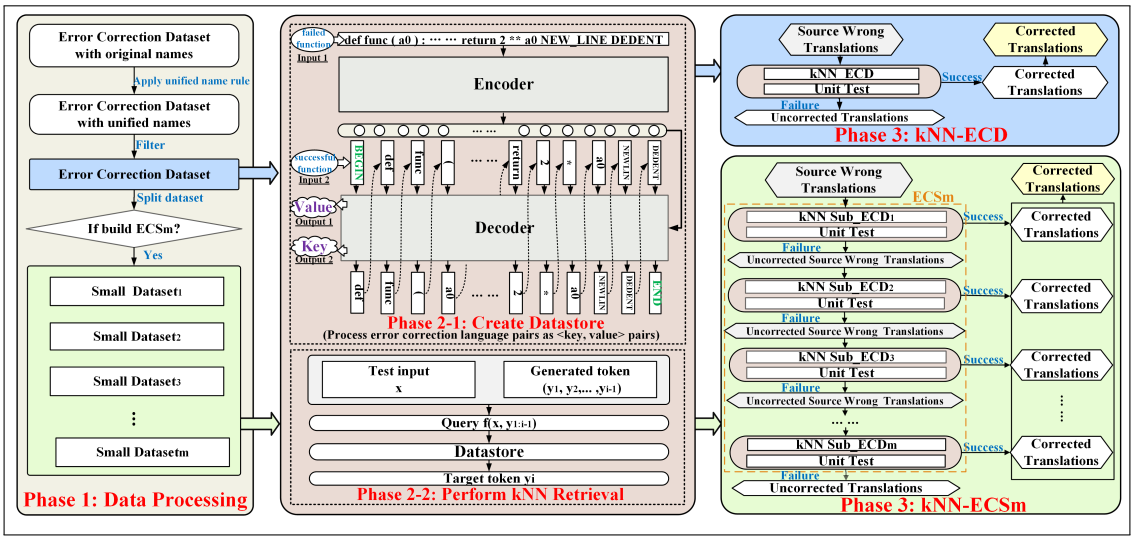


Figure 1: **Training process: Construction workflow of kNN-ECD and kNN-ECS$_m$.** The workflow consists of three phases. (1) After deduplication, we can get the error correction dataset. If we intend to build kNN-ECD, we use the error correction dataset directly; if we intend to build kNN-ECS$_m$, we divide the error correction dataset into m small datasets. (2) For the error correction dataset/small_dataset$_{i\in[1,m]}$, we process it into corresponding datastore and introduce the kNN retrieval. Here, we can get the corresponding kNN-ECD/kNN-sub_ECD$_{i\in[1,m]}$, respectively. (3) For kNN-ECD, we directly conduct unit testing on the output; for kNN-ECS$_m$, we connect sub_ECD$_1, \ldots,$ sub_ECD$_i, \ldots,$ sub_ECD$_m$ in series and carry out unit testing on the output of each sub_ECD$_{i\in[1,m]}$.

et al., 2018; Yang et al., 2019; Raffel et al., 2019; Pan et al., 2023). Subsequently, Szafraniec et al. (2023) proposed TransCoder-IR, which leverages lower-level compiler intermediate representations to advance code translation. Due to the success of pre-trained language model, Feng et al. (2020) introduced CodeBERT, a neural architecture combining BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2019) and a bidirectional Transformer (Vaswani et al., 2023), trained with a hybrid objective function that incorporates both natural language and programming language. Further, in order to accommodate multilingual representations, Ahmad et al. (2021) employed bidirectional and autoregressive transformer (Lewis et al., 2019) for pre-training on unlabeled natural language and programming language data. Li et al. (2022) proposed AlphaCode, which uses a Transformer model to generate a large number of potential solutions, and then extracts suitable candidates through filtering and clustering operations. Currently, almost all program translation models are built based on retraining the Transformer model. However, the complex internal workings of Transformer models almost leads to uninterpretable translations. Furthermore, this process usually consumes a significant amount of computational resources, yet the gains in translation accuracy are quite limited.

kNN-MT Method. As a method for providing interpretable results, the datastore-based kNN retrieval approach is a promising alternative to Transformer-based translation models. Khandelwal et al. (2020) pioneered the concept of kNN-MT, which augments translation by retrieving (key, value) pairs from an external datastore without updating the model. However, the large-scale datastore often suffers from high-latency data retrieval. To address this problem, Wang et al. (2021b) introduced a hierarchical clustering strategy (Kanungo et al., 2002), aiming to improve retrieval efficiency by approximately querying the distance between data points in the datastore. On this basis, Wang et al. (2022) proposed to use the cluster-based compact network and cluster-based pruning solution to compress the datastore, thereby reducing the kNN retrieval latency. Based on the traditional kNN-MT method, researchers have initiated the exploration of more adaptive kNN-MT paradigms. Zheng et al. (2021) introduced the concept of adaptive kNN-MT, dynamically customizing the number of nearest neighbors for each target token. Furthermore, in order to achieve a more interactive and efficient learning method, Wang et al. (2021a) proposed the kNN-over-kNN (KoK) method as a plug-and-play solution for online learning combined with human feedback. However, existing

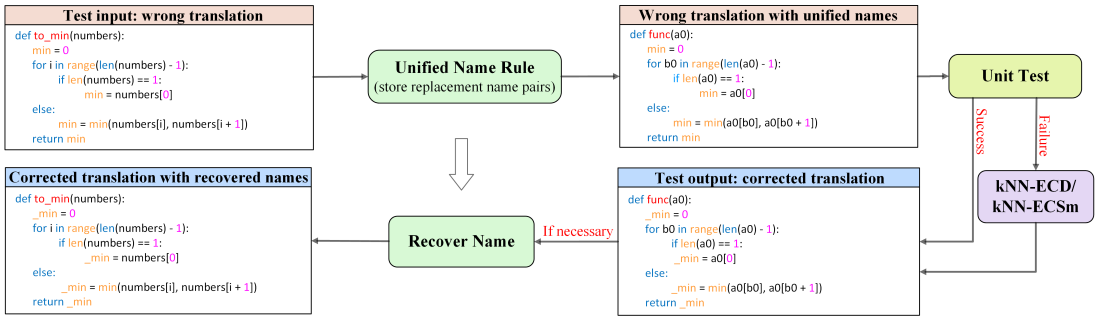


Figure 2: **Testing process: Implementation workflow for kNN-ECD and kNN-ECS$_m$.** First, we apply the *unified name rule* on wrong translations and store replacement name pairs. Subsequently, we perform the unit testing on the processed wrong translations, and then feed the failed translations to the error correction model. If necessary, we can recover the masked names in the corrected translations using the replacement name pairs.

works only focus on the retrieval speed and context adaptation while overlooking the limited search capability, failing to capture comprehensive global information in a large datastore space.

Diverse Rare Identifiers. Various rare identifiers often introduce noise in program translation, distracting attention from critical information such as code structure and logic. Modern source code dataset contains millions of unique identifiers, with less than 1% of the identifiers appearing more than 5 times (Karampatsis et al., 2020). To address this issue, Chirkova & Troshin (2020b) presented an identifier-anonymization-based approach to handle Out-of-Vocabulary identifiers, which significantly improves Transformer's performance in code completion (Svyatkovskiy et al., 2020) and bug fixing (Gupta et al., 2017; Li et al., 2020). Additionally, Xu et al. (2019) also recognized the limitation of Out-of-Vocabulary terms, proposing to replace all class/variable names with corresponding placeholders. For unfamiliar identifiers, Ahmed et al. (2018) proposed the TRACER method, which handles these cases by replacing all identifiers with their recommended abstract types. Furthermore, by incorporating a more versatile value anonymization process, Chirkova & Troshin (2020a) explored the effectiveness of anonymization across a broader range of tasks based on the Transformer architecture. In previous works, researchers have ignored the possibility of unifying diverse identifiers in both training and test datasets. Also, existing methods are extremely simple, often breaking the code structure and neglecting to avoid replacing special identifiers such as built-in functions.

3 APPROACH

In this paper, we aim to enhance code translation through error correction, rather than retraining the program translation model. As shown in Figure 1, following data processing, we create the corresponding datastore and perform kNN retrieval on two variants of error correction dataset, constructing two alternative error correction models to enhance TransCoder-ST: kNN-ECD and kNN-ECS$_m$.

3.1 CREATION OF ERROR CORRECTION DATASET

Generating error correction language pairs. We intend to extract error correction language pairs from TransCoder-ST, guiding kNN-ECD and kNN-ECS$_m$ in learning correction knowledge. First, we download the Java source dataset from Google BigQuery and process it using the TransCoder-ST preprocessing pipeline. Subsequently, we employ EvoSuite (Dinella et al., 2022) to create high-quality Java unit test cases, and then feed the processed Java functions with the test cases into TransCoder-ST (Java $\rightarrow$ Python, beam_size $= N$). For each Java function, we can get N Python functions with their corresponding unit test cases, followed by executing the Python unit tests. If both 'success' and 'failure' occur in N test results, we combine the first failed Python function with the first successful Python function to form an error correction language pair.

Unified name rule. Inspired by OOV anonymization method (Chirkova & Troshin, 2020b), we propose a *unified name rule* to standardize the diverse rare identifiers in the training dataset and test dataset, thus reducing noise in the retrieval process. First, we categorize non-built-in identifiers into three groups: variable names, function name, and parameter names, where each function has

only one function name. Then, we replace the identifiers initially used in the code with identifiers following a homogeneous schema of naming, as follows:

Unified name rule: For non-built-in identifiers in a function, we replace parameter names by $a_0, a_1, \ldots, a_{j\in\mathbb{N}}$, function name by $func$, and variable names by $b_0, b_1, \ldots, b_{j\in\mathbb{N}}$ in order. During this process, the replacement name pairs of each function are recorded. If needed, the unified names can be recovered.

In Figure 2, we give an example of a *unified name rule* used for identifier replacement. By implementing the *unified name rule*, we can filter out extraneous information from rare identifiers, thereby channeling attention toward essential code elements such as logic and structure. Besides, this rule facilitates preliminary error correction within the test dataset, repairing errors resulting from identifier confusion in the code, such as the overlap between function names and variable names.

3.2 ERROR CORRECTION USING kNN-ECD

Creating error correction datastore (ECD). The ECD is generated from the error correction dataset, which mainly includes two primary storage components. The first component is designed to store error correction knowledge in the format of (key, value) pairs, where the generation of (key, value) pair follows kNN-MT method (Khandelwal et al., 2020). The second component is used to store source functions and target function prefixes $\langle \texttt{src_func}, \texttt{tgt_func_pref} \rangle$, as well as their corresponding ground truth target tokens `tgt_tok`. More specifically, we use the pre-trained Transformer model as a coding tool, where key $= f(\langle \text{src_func}, \text{tgt_func_pref} \rangle)$ is the representation of $\langle \text{src_func}, \text{tgt_func_pref} \rangle$ obtained from the last hidden layer of the decoder, and value $= h(\text{tgt_tok})$ is the tokenization of tgt_tok (Ferrando et al., 2022). For each token generated from (key, value) pairs, users can obtain a detailed explanation by accessing the corresponding $\langle \text{src_func}, \text{tgt_func_pref} \rangle \rightarrow$ tgt_tok, where $\langle \text{src_func}, \text{tgt_func_pref} \rangle$ serves as the *decision-making basis* for the generated token tgt_tok.

Performing kNN retrieval on ECD. Combined with the ECD, we utilize kNN retrieval to correct the wrong translations of TransCoder-ST. The kNN method conducts the similarity search on a large-scale datastore, retrieving relevant (key, value) pairs to generate error correction results. In the code correction process, when generating the next token y_i, at each step, we utilize the representation $f(x, \hat{y}_{1:i-1})$ as a *query*, based on the test input x. Following this, we retrieve the k nearest neighbors to the *query* from the error correction datastore, where $\hat{y}$ represents the generated token. Then, we calculate the distance $d()$ between the key and the *query* as a weight to regularize the probability of the value. On the basis of kNN-MT (Khandelwal et al., 2020), we define the probability distribution of the next token as follows:

$$p\left(y_i|x, \hat{y}_{1:i-1}\right) = \sum_{(k_j,v_j)\in N} \mathbb{1}_{y_i=v_j} exp\left(\frac{-d(k_j, f(x, \hat{y}_{1:i-1}))}{T}\right)$$

where T is the temperature, used to prevent overfitting in the retrieval context. When the temperature value $T > 1$, it tends to make the distribution more uniform, preventing deviations towards the most similar retrieval results and ensuring diversity in the retrieved data.

3.3 ERROR CORRECTION USING kNN-ECS$_m$

Creating error correction system (ECS$_m$). As shown in Figure 1, ECS$_m$ consists of m sub-datastores, which are connected in sequential order. To build ECS$_m$, we first randomly divide the error correction dataset into m equal parts $\{\text{small_dataset}_1, \ldots, \text{small_dataset}_i, \ldots, \text{small_dataset}_m\}$, where each $\text{small_dataset}_{i\in[1,m]}$ contains the same number of error correction language pairs. That is to say, ECD and ECS$_m$ are generated from the same error correction dataset. Then, we follow the ECD creation process to generate the corresponding $\{\text{sub_ECD}_1, \ldots, \text{sub_ECD}_i, \ldots, \text{sub_ECD}_m\}$, where these sub_ECDs are linked sequentially. Here, each $\text{sub_ECD}_{i\in[1,m]}$ contains two primary storage components. The first components is dedicated to storing (key, value) pairs, while the second components is designed for storing $\langle \text{src_func}, \text{tgt_func_pref} \rangle \rightarrow$ tgt_tok records.

Performing kNN retrieval on ECS$_m$. Within the framework of ECS$_m$, we repeatedly feed incorrected wrong translations into subsequent $\text{sub_ECD}_{i\in[1,m]}$ and perform kNN retrieval, using diverse

data flows for multiple rounds of error correction. Specifically, for each sub_ECD$_{i\in[1,m]}$, we employ kNN retrieval to generate outputs and conduct unit testing on the generated results. If the unit testing result is successful, it means that the wrong translation has been corrected. In this case, we output the corrected code. However, if the unit testing result is failed, it means that the wrong translation has not been adequately corrected. In response, we re-enter the source wrong translation into the next sub-datastore. By adopting a distributed structure with multiple sub-datastores, kNN retrieval can capture more comprehensive repair information, effectively enhancing the kNN search capability.

4 EXPERIMENTS

4.1 EXPERIMENTAL DETAILS

Model architecture. The implementation of kNN-ECD/kNN-ECS$_m$ is conducted under the guidance of Oracle (Loney & McClain, 2004), where Oracle represents a group of manual unit testers. By reviewing the output and providing feedback, Oracle assists us in determining whether the corrected results make sense. As shown in Figure 2, we implement the *unified name rule* on the test input for preliminary error correction before feeding it into kNN-ECD/kNN-ECS$_m$. Then, we execute unit testing on the processed test dataset. If successful, it means that the wrong translation is due to identifier confusion. If failed, we feed the failed test samples into ECD/ECS$_m$.

Training dataset & test dataset. We download the Java source code from Google BigQuery and implement the TransCoder-ST preprocessing pipeline for dataset filtering. Next, we set the maximum runtime of 20 seconds for each process and employ EvoSuite to create Java test cases. Here, the unit test cases are created based on two criteria: mutation score over 0.9 and at least two assertions. In this process, we collect 82,665 Java functions with unit test cases.

Subsequently, we feed the above processed Java function into TransCoder-ST (beam_size = 10), and set the target language as Python. In this step, $82,665 \times 10$ Python functions with unit test cases are generated. For each Java function, we perform unit testing on the corresponding 10 Python functions. If 'success' and 'failure' both appear in 10 test results, we merge the first failed Python function and the first successful Python function into an error correction language pair. Ultimately, we can get a preliminary error correction dataset with 26,230 language pairs for TransCoder-ST.

After that, we employ the *unified name rule* to standardize diverse rare identifiers within the preliminary error correction dataset. By removing duplicate entries from the processed dataset, an error correction dataset with 21,385 correction pairs is built. Then, we divide the error correction dataset into a training dataset and a test dataset with a ratio of $9 : 1$. Among them, the training dataset consists of 19,368 error correction language pairs, and the test dataset consists of 2,017 wrong translations, each with unit test cases and replacement name pairs[1].

Training details & fine-tuning details. During the training phase, we introduce the previous version of TransCoder-ST as a coding tool for error correction language pairs, mainly because its encoder and decoder can simultaneously process codes in the same programming language, which conforms to the data properties of error correction language pairs (Qi et al., 2018). Based on this, we feed the error correction language pair into the coding tool, where we treat the correct translation as *target* and the wrong translation as *source*. Then, we extract the cross-attention output as *key*, and the tokenization of the ground truth target token as *value*.

During the fine-tuning phase, for ECD and ECS$_m, m \in \{3, 6, 9, 12, 15, 18\}$, we consider the following parameters: neighbor (p_0) and temperature (p_1) . When more neighbors are retrieved, noise may be introduced, which can lead to worse results. Meanwhile, properly adjusting the temperature parameters can prevent excessive bias towards a single neighbor, thus ensuring the diversity of results. In this case, we test the performance of ECD and ECS$_m$ under $p_0 \in \{1, 4, 8, 12, 16, 32\}$, $p_1 \in \{1, 10, 100, 1000\}$, and select the optimal parameter combination. Experiments show that for ECD, when $p_0, p_1 = \{4, 10\}$, the best error correction rate can be achieved. For ECS$_{m=3,12,15}$, the optimal parameter combination is $p_0, p_1 = \{8, 10\}$. ECS$_{m=6,18}$ attain the highest effectiveness with the parameter combination $p_0, p_1 = \{16, 10\}$. As for ECS$_{m=9}$, the most effective parameter combination is $p_0, p_1 = \{2, 10\}$.

[1]https://github.com/minxue29031/Error_Correction

4.2 RESULTS AND DISCUSSION

Translation performance. In Table 1, we compare the translation performance of TransCoder-ST after combining ECD/ECS$_m$ or ECD*/ECS$^*_m$. In the comparison, we focus on the translation from Java to Python, where the performance of our models is measured against j2py[2], TransCoder, DOBF, TransCoder-ST (TC-ST). As shown in Table 1, it is clear that when TransCoder-ST utilizes the error correction model, the translation performance improves significantly, increasing from 68.9% to a range of 82.4% $\sim$ 89.9%. This improvement stems from the datastore, which stores a large amount of error correction information generated from TransCoder-ST. By systematically retrieving relevant (key, value) pairs, we can correct relative errors in the wrong translations. Moreover, comparing kNN-ECD/kNN-ECS$_m$ with kNN-ECD*/kNN-ECS$^*_m$, the former shows superior performance. This difference is attributed to the *unified name rule*, which reduces the interference caused by rare identifiers during datastore construction and implementation.

Table 1: Translation performance of TransCoder-ST with kNN-ECD/kNN-ECS$_m$ (Java $\rightarrow$ Python).

	kNN-ECS$_{18}$/ kNN-ECS$^*_{18}$	kNN-ECS$_{15}$/ kNN-ECS$^*_{15}$	kNN-ECS$_{12}$/ kNN-ECS$^*_{12}$	kNN-ECS$_9$/ kNN-ECS$^*_9$	kNN-ECS$_6$/ kNN-ECS$^*_6$	kNN-ECS$_3$/ kNN-ECS$^*_3$	kNN-ECD/ kNN-ECD*
j2py	38.3%	38.3%	38.3%	38.3%	38.3%	38.3%	38.3%
TransCoder	49.0%	49.0%	49.0%	49.0%	49.0%	49.0%	49.0%
DOBF	52.7%	52.7%	52.7%	52.7%	52.7%	52.7%	52.7%
Pure TC-ST (Online)	68.9%	68.9%	68.9%	68.9%	68.9%	68.9%	68.9%
TC-ST+ECD*/ECS$^*_m$	89.4%	89.1%	88.7%	87.9%	87.1%	85.1%	82.4%
TC-ST + ECD/ECS$_m$	**89.9%**	89.6%	89.3%	89.1%	87.9%	86.5%	84.5%

[*] For kNN-ECD/kNN-ECS$_m$, the *unified name rule* was employed during the training and testing phases.
[*] For kNN-ECD*/kNN-ECS$^*_m$, the *unified name rule* was not employed during the training and testing phases.
[*] We cannot directly compare our results with the latest research, TransCoder-IR, because it is not suitable for translating from Java to Python.

Interpretability of error correction model. During the testing phase, the traditional Transformer model cannot track and identify which snippets in the training dataset contributed to each generated token. However, kNN-ECD/kNN-ECS$_m$ can address this issue, providing an intuitive and readable decision-making basis for each output token. When building the datastore, we store the (key, value) pairs and the corresponding snippets $\langle$src_func, tgt_func_pref$\rangle$ $\rightarrow$ tgt_tok from the training dataset, where the coding of $\langle$src_func, tgt_func_pref$\rangle$ is key, the coding of tgt_tok is value, and $\langle$src_func, tgt_func_pref$\rangle$ serves as the *decision-making basis* for the generated token tgt_tok. Consequently, when generating each output token, we will return both (key, value) pair and $\langle$src_func, tgt_func_pref$\rangle$ $\rightarrow$ tgt_tok. In APPENDIX Table 7, we show a detailed decision-making process for generating a corrected translation. For clarity, we only focus on the correction process of the wrong token '/' $\rightarrow$ '//' in Table 2, where the (key, value) pair used to correct the wrong token '/' $\rightarrow$ '//' is a string of numbers, making it challenging to extract valid intuitive information. In such cases, the uncoded form $\langle\cdot,\cdot\rangle$ $\rightarrow$ '//' of (key, value) pair can provide more intuitive information.

Table 2: Interpretable error correction process for repairing incorrect tokens in wrong translations

Wrong translation	Corrected translation	Decision-making basis
`def func ( a0 ) :` `  return 9 * a0 / 5 + 32`	`def func ( a0 ) :` `  return 9 * a0 // 5 + 32`	$\langle$ 'def func (a0) : NEW_LINE INDENT return a0 / 6 NEW_LINE DEDENT', 'def func (a0) : NEW_LINE INDENT return a0'$\rangle$ $\rightarrow$ '//'

`def func ( a0 ) :`
`  b = list ( a0 )`
`  b . sort ( )`
`  return b [ len ( b ) / 2 ]`
	`def func ( a0 ) :`
	`  b = list ( a0 )`
	`  b . sort ( )`
	`  return b [ len ( b ) // 2 ]`
		$\langle$ 'def func (a0) : NEW_LINE INDENT a0 . sort ( ) NEW_LINE return a0 [len (a0) / 2] NEW_LINE DEDENT', 'def func (a0) : NEW_LINE INDENT a0 . sort ( ) NEW_LINE return a0 [len (a0)'$\rangle$ $\rightarrow$ '//'

`def func ( a0 ) :`
`  b = 0`
`  while a0 > 0 :`
`  if a0 % 10 == 2 :`
`  b += 1`
`  a0 = a0 / 10`
`  return b`
	`def func ( a0 ) :`
	`  b = 0`
	`  while a0 > 0 :`
	`  if a0 % 10 == 2 :`
	`  b += 1`
	`  a0 = a0 // 10`
	`  return b`
		$\langle$ 'def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE while a0 $>$ 0 : NEW_LINE INDENT sum += (a0 % 10) ** 2 NEW_LINE a0 = a0 / 10 NEW_LINE DEDENT return sum NEW_LINE DEDENT', 'def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE while a0 $>$ 0 : NEW_LINE INDENT sum += (a0 % 10) ** 2 NEW_LINE a0 = a0'$\rangle$ $\rightarrow$ '//'

[*] $\langle\cdot,\cdot\rangle$ $\rightarrow$ '//' : $\langle\cdot,\cdot\rangle$ represents the decision-making basis of the correction process '/' $\rightarrow$ '//', $\langle\cdot,\cdot\rangle$ indicates $\langle$src_func, tgt_func_pref$\rangle$, and '//' indicates tgt_tok.

Generalizability analysis. To explore the generalization of the error correction model, first, we analyze the overlap of code text between the test dataset and the training dataset. Following data preprocessing, we observe that there are no duplicated codes between the two datasets. Additionally, comparing the wrong code fragments (i.e., code fragment containing the wrong token) in the test

[2]https://github.com/natural/java2python

dataset with those in the training dataset, we still cannot find any identical wrong code fragments. It means that the kNN-ECD/kNN-ECS$_m$ can learn how to correct wrong tokens, rather than accidentally correcting wrong tokens due to the test dataset containing duplicate 'codes' or 'wrong code fragments' with the training dataset. Second, we show the interpretable decision-making basis for wrong token fixing in Table 2, where $\langle$src_func, tgt_func_pref$\rangle \rightarrow$ tgt_tok is retrieved to fix the wrong token '/' $\rightarrow$ '//'. We find that the text of the wrong translation is quite different from the text of $\langle$src_func, tgt_func_pref$\rangle$. It indicates that the error correction model can well apply the knowledge learned from the error correction dataset to new samples. In summary, the above error correction behavior demonstrates that the error correction model can extend the repair knowledge acquired from the training dataset to new wrong translations, rather than merely relying on straightforward comparisons of similar code texts.

Error correction performance of kNN-ECD* and kNN-ECS$^*_m$. In Table 3, we compare the error correction performance of kNN-ECD* and kNN-ECS$^*_m$. It is worth noting that kNN-ECD* and kNN-ECS$^*_m$ are generated from the same error correction dataset, which means that both contain the same error correction information. Surprisingly, in this case, kNN-ECS$^*_m$ achieves 65.8% error correction rate, showing a substantial 22.3% improvement compared to kNN-ECD*. Besides, as shown in Figure 4(a), with the increase in the number of sub-datastores under the error correction system, we observe an improvement in error correction performance. The main reason is that traditional kNN methods usually suffer from insufficient retrieval capabilities in a large datastore, which tends to focus on high-density regions while failing to capture potential correlations in low-density regions. In contrast to kNN-ECD*, the improvement of kNN-ECS$^*_m$ is primarily attributed to its distributed structure, which includes diverse datastore variants. By employing different data flows for multiple rounds of error correction, it can capture more comprehensive error correction information.

Table 3: Correction performance of kNN-ECD*/kNN-ECS$^*_m$ on TransCoder-ST wrong translations

	kNN-ECS$^*_{18}$	kNN-ECS$^*_{15}$	kNN-ECS$^*_{12}$	kNN-ECS$^*_9$	kNN-ECS$^*_6$	kNN-ECS$^*_3$	kNN-ECD*
sub_ECD$^*_1$	25.4%	26.4%	27.3%	28.5%	30.7%	33.9%	43.5%
sub_ECD$^*_2$	26.5%	27.2%	28.6%	30.0%	33.0%	36.3%	-
sub_ECD$^*_3$	26.5%	27.4%	28.0%	28.7%	32.0%	35.9%	-
sub_ECD$^*_4$	26.1%	27.9%	30.4%	29.3%	30.9%	-	-
sub_ECD$^*_5$	25.0%	26.2%	29.5%	28.0%	31.4%	-	-
sub_ECD$^*_6$	26.0%	26.6%	28.6%	29.5%	32.1%	-	-
sub_ECD$^*_7$	25.4%	29.1%	28.7%	28.4%	-	-	-
sub_ECD$^*_8$	24.1%	26.7%	26.8%	26.9%	-	-	-
sub_ECD$^*_9$	25.9%	27.0%	26.8%	30.2%	-	-	-
sub_ECD$^*_{19}$	27.9%	27.6%	27.3%	-	-	-	-
sub_ECD$^*_{11}$	26.9%	26.8%	28.5%	-	-	-	-
sub_ECD$^*_{12}$	25.7%	24.9%	27.8%	-	-	-	-
sub_ECD$^*_{13}$	27.7%	26.0%	-	-	-	-	-
sub_ECD$^*_{14}$	27.1%	27.0%	-	-	-	-	-
sub_ECD$^*_{15}$	26.0%	27.6%	-	-	-	-	-
sub_ECD$^*_{16}$	26.2%	-	-	-	-	-	-
sub_ECD$^*_{17}$	26.0%	-	-	-	-	-	-
sub_ECD$^*_{18}$	26.3%	-	-	-	-	-	-
sub_ECD$^*_{avg}$	26.1%	27.0%	28.2%	28.8%	31.7%	35.4%	43.5%
ECD*/ECS$^*_m$	**65.8%**	64.9%	63.8%	61.1%	58.4%	52.1%	43.5%

* For kNN-ECD*/kNN-ECS$^*_m$, the *unified name rule* was not employed during the training and testing phases.

Impact of the unified name rule. In Figure 4, we show the error correction performance of kNN-ECD and kNN-ECS$_m$ after applying the *unified name rule*. Comparing Table 3 and Table 4, we observe that implementing the *unified name rule* can effectively enhance the error correction rate, ranging from 1.7% $\sim$ 6.5%. The main reason is that, during the training and testing phases, using the *unified name rule* can ignore the diversity of function names, parameter names, and variable names as much as possible, while paying more attention to the logic and structure of the code. By adopting the *unified name rule*, we can minimize the emphasis on rare identifiers, thereby eliminating the interference produced by diverse rare identifiers during the process of datastore construction and implementation. Furthermore, it is worth mentioning that employing the *unified name rule* alone can achieve 5.1% preliminary error corrections before feeding the wrong translation into ECD/ECS$_m$, where the errors mainly arise from identifier confusion within the code.

Ablation analysis. In Table 5, we investigate the importance of adjacent tokens in the process of identifying and correcting wrong tokens within the input. Our approach is to progressively remove

Table 4: Correction performance of kNN-ECD/kNN-ECS$_m$ on TransCoder-ST wrong translations

	kNN-ECS$_{18}$	kNN-ECS$_{15}$	kNN-ECS$_{12}$	kNN-ECS$_9$	kNN-ECS$_6$	kNN-ECS$_3$	kNN-ECD
sub_ECD$_1$	29.0%	29.4%	30.4%	32.1%	35.5%	42.5%	48.7%
sub_ECD$_2$	26.3%	27.1%	29.9%	32.8%	37.1%	37.8%	-
sub_ECD$_3$	26.0%	26.1%	27.3%	33.8%	33.1%	37.1%	-
sub_ECD$_4$	25.3%	28.4%	27.9%	29.0%	33.7%	-	-
sub_ECD$_5$	25.3%	29.7%	30.9%	30.4%	33.2%	-	-
sub_ECD$_6$	28.1%	25.8%	32.0%	30.0%	32.5%	-	-
sub_ECD$_7$	28.3%	26.9%	31.0%	28.4%	-	-	-
sub_ECD$_8$	26.3%	29.9%	28.0%	30.9%	-	-	-
sub_ECD$_9$	25.0%	29.4%	29.8%	29.2%	-	-	-
sub_ECD$_{10}$	25.7%	28.5%	28.1%	-	-	-	-
sub_ECD$_{11}$	26.7%	28.0%	28.1%	-	-	-	-
sub_ECD$_{12}$	26.5%	25.8%	26.7%	-	-	-	-
sub_ECD$_{13}$	29.1%	27.1%	-	-	-	-	-
sub_ECD$_{14}$	27.8%	29.3%	-	-	-	-	-
sub_ECD$_{15}$	27.3%	27.2%	-	-	-	-	-
sub_ECD$_{16}$	25.2%	-	-	-	-	-	-
sub_ECD$_{17}$	25.8%	-	-	-	-	-	-
sub_ECD$_{18}$	28.0%	-	-	-	-	-	-
sub_ECD$_{avg}$	26.8%	27.9%	29.2%	30.7%	34.2%	39.2%	48.7%
Unified name rule	5.1%	5.1%	5.1%	5.1%	5.1%	5.1%	5.1%
ECD/ECS$_m$	**67.5%**	66.7%	65.7%	64.8%	61.2%	56.5%	50.0%

* For kNN-ECD/kNN-ECS$_m$, the *unified name rule* was employed during the training and testing phases.

the adjacent tokens near the target wrong token in the input. This systematic approach provides a deeper understanding of how adjacent tokens contribute to the identification and correction of wrong tokens, depending on whether the wrong token is successfully corrected in the output. In Table 5, the wrong token '/' in the wrong translation should be corrected to '//'. Initially, when we feed the original input to the error correction model, it can effectively correct the wrong token '/'. However, as we systematically remove the adjacent tokens one by one, we encounter challenges in rectifying the wrong token. It means that adjacent tokens are key to identifying and correcting wrong tokens. The identification and correction of wrong tokens relies on the internal relationships among adjacent tokens, rather than directly pinpointing the wrong tokens. In essence, the error correction method corrects wrong translations by learning and understanding the internal relationships between tokens within a function.

Table 5: The impact of adjacent tokens in the error correction process

	Input (original format)	Input (remove len ( ))	Input (remove b)	Input (remove return)

Input	`def func(a0):`
	`  b = 0`
	`  for c in a0:`
	`  b += c`
	`  return b / len(a0)`
		`def func(a0):`
		`  b = 0`
		`  for c in a0:`
		`  b += c`
		`  return b / a0`
			`def func(a0):`
			`  b = 0`
			`  for c in a0:`
			`  b += c`
			`  return / len(a0)`
				`def func(a0):`
				`  b = 0`
				`  for c in a0:`
				`  b += c`
				`  b / len(a0)`

Output	`def func(a0):`
	`  b = 0`
	`  for c in a0:`
	`  b += c`
	`  return b // len(a0)`
		`def func(a0):`
		`  b = 0`
		`  for c in a0:`
		`  b += c`
		`  return b / a0`
			`def func(a0):`
			`  b = 0`
			`  for c in a0:`
			`  b += c`
			`  return b / len(a0)`
				`def func(a0):`
				`  b = 0`
				`  for c in a0:`
				`  b += c`
				`  b / len(a0)`
				`  return b`

5 Conclusion

In real-world scenarios, the interpretability of outputs plays a crucial role in gaining users' trust. Currently, Transformer-based models are widely used for program translation. However, even with researchers investing significant time and computational resources in retraining models, the improvement in translation accuracy remains relatively limited. Furthermore, due to the complex internal workflow of the Transformer model, it is difficult to track and identify which snippet in the training dataset contribute to each output token. In this paper, we employ the kNN retrieval on an error correction datastore to enhance the translation capability of TransCoder-ST through code correction. This approach provides a decision-making basis for each generated token, laying a solid research foundation for the subsequent improvement of error correction. Importantly, by simply integrating additional error correction datastore, the datastore-based kNN retrieval approach significantly enhances the translation performance of TransCoder-ST, without the need to consume significant computational resources to retrain the Transformer-based model.

REFERENCES

Hindle A. 2015 Aggarwal K, Salameh M. Using machine translation for converting python 2 to python 3 code. *PeerJ PrePrints 3:e1459v1*, 2015. URL https://doi.org/10.7287/peerj.preprints.1459v1.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *CoRR*, abs/2103.06333, 2021. URL https://arxiv.org/abs/2103.06333.

Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. Compilation error repair: For the student programs, from the student programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, pp. 78–87, 2018.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/d759175de8ea5b1d9a2660e45554894f-Paper.pdf.

Nadezhda Chirkova and Sergey Troshin. Empirical study of transformers for source code. *CoRR*, abs/2010.07987, 2020a. URL https://arxiv.org/abs/2010.07987.

Nadezhda Chirkova and Sergey Troshin. A simple approach for handling out-of-vocabulary identifiers in deep learning for source code. *CoRR*, abs/2010.12663, 2020b. URL https://arxiv.org/abs/2010.12663.

Damai Dai, Li Dong, Yaru Hao, Zhifang Sui, Baobao Chang, and Furu Wei. Knowledge neurons in pretrained transformers. pp. 8493–8502, 01 2022. doi: 10.18653/v1/2022.acl-long.581.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. TOGA. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, may 2022. doi: 10.1145/3510003.3510141. URL https://doi.org/10.1145%2F3510003.3510141.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020. URL https://arxiv.org/abs/2002.08155.

Javier Ferrando, Gerard I. Gállego, Belen Alastruey, Carlos Escolano, and Marta R. Costa-jussà. Towards opening the black box of neural machine translation: Source and target interpretations of the transformer, 2022.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common C language errors by deep learning. In Satinder Singh and Shaul Markovitch (eds.), *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pp. 1345–1351. AAAI Press, 2017. URL http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603.

Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, 2002. doi: 10.1109/TPAMI.2002.1017616. URL https://doi.org/10.1109/TPAMI.2002.1017616.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: Open-vocabulary models for source code. *CoRR*, abs/2003.07914, 2020. URL https://arxiv.org/abs/2003.07914.

Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Nearest neighbor machine translation. *CoRR*, abs/2010.00710, 2020. URL https://arxiv.org/abs/2010.00710.

Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *CoRR*, abs/1901.07291, 2019. URL http://arxiv.org/abs/1901.07291.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *CoRR*, abs/1910.13461, 2019. URL http://arxiv.org/abs/1910.13461.

Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 602–614, 2020.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Eccles, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

Kevin Loney and Lisa McClain. *Oracle Database 10g The Complete Reference*. McGraw-Hill, Inc., USA, 1 edition, 2004. ISBN 0072253517.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in gpt, 2023.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Understanding the effectiveness of large language models in code translation, 2023.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*. Association for Computational Linguistics, 2001. doi: 10.3115/1073083.1073135. URL https://doi.org/10.3115/1073083.1073135.

Ye Qi, Devendra Singh Sachan, Matthieu Felix, Sarguna Janani Padmanabhan, and Graham Neubig. When and why are pre-trained word embeddings useful for neural machine translation? *CoRR*, abs/1804.06323, 2018. URL http://arxiv.org/abs/1804.06323.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019. URL http://arxiv.org/abs/1910.10683.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 20601–20611. Curran Associates, Inc., 2020.

Baptiste Rozière, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. *CoRR*, abs/2102.07492, 2021. URL https://arxiv.org/abs/2102.07492.

Baptiste Roziere, Jie M. Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. *arXiv:2110.06773 [cs]*, Feb 2022. URL http://arxiv.org/abs/2110.06773. arXiv: 2110.06773.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural language to code translation with execution, 2022.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

Marc Szafraniec, Baptiste Roziere, Hugh Leather, Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. 2023.

Zhaopeng Tu, Yang Liu, Shuming Shi, and Tong Zhang. Learning to remember translation history with a continuous cache. *CoRR*, abs/1711.09367, 2017. URL http://arxiv.org/abs/1711.09367.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Yaron Singer, and Stuart Shieber. Investigating gender bias in language models using causal mediation analysis. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 12388–12401. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/92650b2e92217715fe312e6fa7b90d82-Paper.pdf.

Dexin Wang, Kai Fan, Boxing Chen, and Deyi Xiong. Efficient cluster-based k-nearest-neighbor machine translation, 2022.

Dongqi Wang, Haoran Wei, Zhirui Zhang, Shujian Huang, Jun Xie, Weihua Luo, and Jiajun Chen. Non-parametric online learning from human feedback for neural machine translation. *CoRR*, abs/2109.11136, 2021a. URL https://arxiv.org/abs/2109.11136.

Shuhe Wang, Jiwei Li, Yuxian Meng, Rongbin Ouyang, Guoyin Wang, Xiaoya Li, Tianwei Zhang, and Shi Zong. Faster nearest neighbor machine translation. *CoRR*, abs/2112.08152, 2021b. URL https://arxiv.org/abs/2112.08152.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 3975–3981. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/552. URL https://doi.org/10.24963/ijcai.2019/552.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237.

Jingyi Zhang, Masao Utiyama, Eiichiro Sumita, Graham Neubig, and Satoshi Nakamura. Guiding neural machine translation with retrieved translation pieces. *CoRR*, abs/1804.02559, 2018. URL http://arxiv.org/abs/1804.02559.

Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. Multilingual code co-evolution using large language models, 2023.

Xin Zheng, Zhirui Zhang, Junliang Guo, Shujian Huang, Boxing Chen, Weihua Luo, and Jiajun Chen. Adaptive nearest neighbor machine translation. *CoRR*, abs/2105.13022, 2021. URL https://arxiv.org/abs/2105.13022.

A APPENDIX

A.1 EVALUATION

Most program translation studies employ BLEU scores to assess the quality of the results (Papineni et al., 2001; Chen et al., 2018; Aggarwal K, 2015). However, a wrong translation typically involves only a few wrong tokens, which means that the BLEU score tends to remain relatively high, regardless of the correctness of the translation. Inspired by computational accuracy in TransCoder (Roziere et al., 2020), we introduce *functional equivalence* to measure the quality of code correction.

Functional equivalence: Given the same input, the source code and the corrected code produce the same output, i.e., both the source code and the corrected code succeed under the same unit test cases.

This metric emphasizes the correctness and functionality of the code, rendering it better suited for evaluating the quality of corrected translations. For example, some wrong translations occur not due to bugs in the code, but because the source code and the translated code fail to produce the same output when given the same input.

A.2 GENERATION OF PYTHON UNIT TEST DATASET

Why do we need to use translated Python unit tests? Firstly, in the process of generating the error correction training dataset, we translate a Java function to multiple Python functions (beam_size $>$ 1), and then combine the first failed Python function with the first successful Python function to form an error correction language pair. Therefore, it is crucial to ensure that the Java function and its corresponding multiple Python functions have equivalent unit test dataset, as well as multiple Python functions under the same Java function have the same unit test dataset. Secondly, during the testing phase, we need to verify that the wrong Python function, the corrected Python function, and its source Java function have an equivalent unit test dataset, as well as the wrong Python function and the corrected Python function have the same unit test dataset. Based on the above two cases, directly translating Java unit tests into Python unit tests is a simple and effective approach.

How to ensure the feasibility of the translated unit test dataset? For the translation process[3] from Java unit test dataset to Python unit test dataset and the screening process of source Java functions, we refer to the work of TransCoder-ST (Roziere et al., 2020). In Table 6, we show the process of using the error correction model to rectify the wrong Python function generated by TransCoder-ST, where Python unit tests are the translations of Java unit tests. In our training dataset and test dataset, each wrong Python function has a matching correct function (ground truth) that passes the corresponding unit test dataset, and almost every unit test dataset contains 3$\sim$6 unit tests. This ensures that in the training dataset, the wrong function and the correct function under the same error correction language pair use the same unit test dataset. It also guarantees that during the testing phase, if the wrong function can be corrected, the corrected function will successfully pass all unit tests. More importantly, multiple unit tests under each unit test dataset assure the reliability of the final unit test result.

Table 6: Translation process from Java unit tests to Python unit tests

Type	Details

Source Java function	`public static double clamp (double value, double min, double max) {`
	`  if (value < min) {`
	`  return min;`
	`  }`
	`  if (value > max) {`
	`  return max;`
	`  }`
	`  return value;`
	`}`

Continued on next page

[3] https://github.com/facebookresearch/CodeGen/blob/main/codegen_sources/test_generation/evosuite_tests_translators/evosuite_to_python.py

Type	Details

Wrong Python function	`def func (a0, a1, a2):`
	`  if a0 < a1:`
	`  return min`
	`  if a0 > a2:`
	`  return max`
	`  return a0`

Corrected Python function	`def func (a0, a1, a2):`
	`  if a0 < a1:`
	`  return a1`
	`  if a0 > a2:`
	`  return a2`
	`  return a0`

Java unit test dataset	`# This file was automatically generated by EvoSuite`
	`# Thu Jan 26 21:55:01 GMT 2023`
	
	`import org.junit.Test;`
	`import static org.junit.Assert.*;`
	`import org.evosuite.runtime.EvoRunner;`
	`import org.evosuite.runtime.EvoRunnerParameters;`
	`import org.junit.runner.RunWith;`
	
	`@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS =`
	`  true, useVNET = true, resetStaticState = true, separateClassLoader = true)`
	`public class CLASS_64ef580337f1_ESTest extends CLASS_64ef580337f1_ESTest_scaffolding {`
	
	`  @Test(timeout = 4000)`
	`  public void test0()  throws Throwable  {`
	`  double double0 = CLASS_64ef580337f1.clamp((-1275.777374486698),`
	`  (-72868.4857075), 42848.99);`
	`  assertEquals((-1275.777374486698), double0, 1.0E-4);`
	`  }`
	
	`  @Test(timeout = 4000)`
	`  public void test1()  throws Throwable  {`
	`  double double0 = CLASS_64ef580337f1.clamp(44640.610466346, (-42604.8), (-1.0));`
	`  assertEquals((-1.0), double0, 1.0E-4);`
	`  }`
	
	`  @Test(timeout = 4000)`
	`  public void test2()  throws Throwable  {`
	`  double double0 = CLASS_64ef580337f1.clamp((-2306.1415894012503),`
	`  44640.610466346, (-1.0));`
	`  assertEquals(44640.610466346, double0, 1.0E-4);`
	`  }`
	
	`  @Test(timeout = 4000)`
	`  public void test3()  throws Throwable  {`
	`  double double0 = CLASS_64ef580337f1.clamp(0.0, 0.0, (-1.0));`
	`  assertEquals((-1.0), double0, 1.0E-4);`
	`  }`
	
	`  @Test(timeout = 4000)`
	`  public void test4()  throws Throwable  {`
	`  double double0 = CLASS_64ef580337f1.clamp((-84626.14348206822), 0.0, 0.0);`
	`  assertEquals(0.0, double0, 1.0E-4);`
	`  }`
	
	`  @Test(timeout = 4000)`
	`  public void test5()  throws Throwable  {`
	`  CLASS_64ef580337f1 cLASS_64ef580337f1_0 = new CLASS_64ef580337f1();`
	`  }`
	`}`

Continued on next page

Type	Details
Python unit test dataset	

```
import numpy as np
import math
from math import *
import collections
from collections import *
import heapq
import itertools
import random
import sys
import unittest

#TOFILL
class CLASS_64ef580337f1(unittest.TestCase):
  def test0(self):
      double0 = f_filled((-1275.777374486698), (-72868.4857075), 42848.99)
      assert abs((-1275.777374486698) -  double0) <=  1.0E-4

  def test1(self):
      double0 = f_filled(44640.610466346, (-42604.8), (-1.0))
      assert abs((-1.0) -  double0) <=  1.0E-4

  def test2(self):
      double0 = f_filled((-2306.1415894012503), 44640.610466346, (-1.0))
      assert abs(44640.610466346 -  double0) <=  1.0E-4

  def test3(self):
      double0 = f_filled(0.0, 0.0, (-1.0))
      assert abs((-1.0) -  double0) <=  1.0E-4

  def test4(self):
      double0 = f_filled((-84626.14348206822), 0.0, 0.0)
      assert abs(0.0 -  double0) <=  1.0E-4

if __name__ == '__main__':
    unittest.main()
```

A.3 ITERATIVE CODE CORRECTION

To verify the error correction capability of kNN-ECD, we conduct multi-round experiments on kNN-ECD, trying to perform iterative code correction on the same datastore. We carry out experiments from the following two aspects. On one hand, we iteratively input the source wrong translations into kNN-ECD and compare the overlap of the outputs. The results reveal a significant overlap in corrected translations across multiple trials, with only minor differences in error correction rates ranging from 0.012% to 0.047%. On the other hand, we also attempted to repeatedly feed the wrong output of kNN-ECD back into kNN-ECD for multiple rounds. The experimental results indicate that only 2.5% of wrong functions are re-corrected in the first round. In subsequent rounds, no more than 0.4% of wrong functions are re-corrected each time. The above experiments illustrate that kNN-ECD usually corrects all errors in wrong translation at once, with little additional gain from iterative error correction.

A.4 STABILITY ANALYSIS.

We analyze the stability of the error correction model in terms of both construction and implementation. In the construction phase, we randomly and evenly divide the error correction training dataset into m sub-datasets, and then generate the corresponding $\text{sub_ECD}_{i\in[1,m]}$. In Figure 3, we separately test the independent error correction performance of $\text{sub_ECD}_{i\in[1,m]}$ under the ECS_m, where $\{\text{sub_ECD}_1, \text{sub_ECD}_2, \ldots, \text{sub_ECD}_m\}$ show close error correction rates within the same system. Meanwhile, comparing sub_ECD_{avg} of different kNN-ECS_m in Figure 5, we find that the larger the sub-datastore, the higher the error correction performance. The above observations indicate that the datastore can stably learn correction information from the error correction dataset during the construction process. Moving on to the implementation phase, we repeatedly feed the same test dataset into kNN-ECD, and then compare the overlap of the outputs. The results demonstrate a significant overlap in the corrected translations across multiple trials, with only slight differences in error correction rates ranging from 0.012% to 0.047%. This implies that the error correction model exhibits strong stability during the implementation process, enabling users to trust the output of the model.

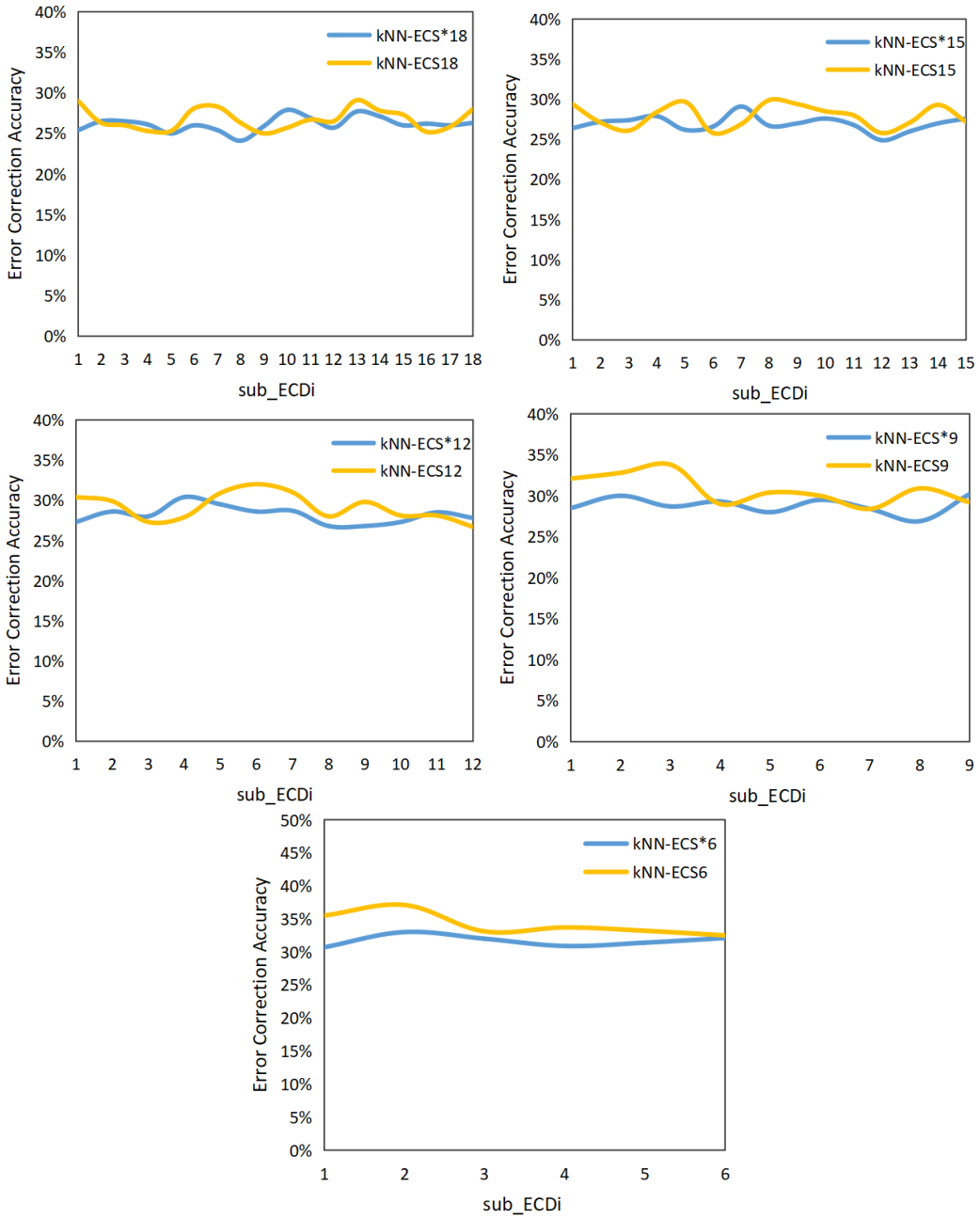


Figure 3: **Independent error correction performance of kNN-sub_ECD$_{i\in[1,m]}$ under kNN-ECS$_m$.** In the same error correction system, where sub_ECDs have similar memory storage, we compare their independent error correction performance respectively. We find that sub_ECDs with approximate information storage capacity exhibit close error correction capabilities. This indicates that datastore can stably learn and apply error correction information from error correction language pairs.

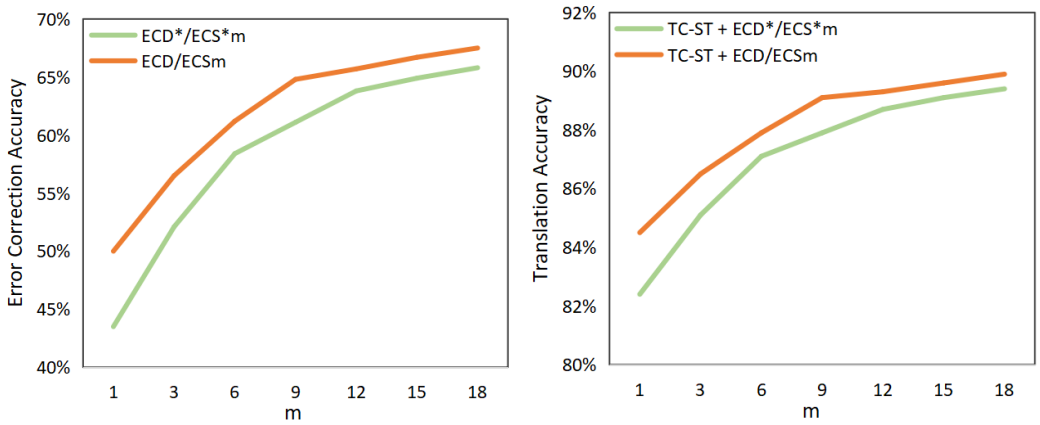
Table 7: The decision-making basis for each generated token in the corrected translation.

Type	Details
Java function	public static void foo (int [ ] buf) for (int i = 0 ; i ¡ buf . length ; i ++) buf [i] = 7 ;
Wrong Python function	def func (a0) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b = 7 NEW_LINE DEDENT DEDENT
Corrected function	def func (a0) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0 [b] = 7 NEW_LINE DEDENT DEDENT
Decision-making basis	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for (b , b) in enumerate (a0) : NEW_LINE INDENT a0 [b] = a1 NEW_LINE DEDENT DEDENT’, ”⟩ → ‘def’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (a1) : NEW_LINE INDENT yield a0 NEW_LINE DEDENT DEDENT’, ‘def’⟩ → ‘func ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (a1) : NEW_LINE INDENT yield a0 NEW_LINE DEDENT DEDENT’, ‘def func’⟩ → ‘(’
	⟨ ‘def func (a0) : NEW_LINE INDENT b = 0 NEW_LINE for c in a0 : NEW_LINE INDENT b = b + 3 NEW_LINE DEDENT return b NEW_LINE DEDENT’, ‘def func (’⟩ → ‘a0 ’
	⟨ ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in a0 : NEW_LINE INDENT sum += b NEW_LINE DEDENT c = d = sum NEW_LINE for b in a0 : NEW_LINE INDENT d -= b NEW_LINE if c == d : NEW_LINE INDENT return b NEW_LINE DEDENT c += b NEW_LINE DEDENT return - 1 NEW_LINE DEDENT’, ‘def func (a0’⟩ → ‘) ’
	⟨ ‘def func (a0) : NEW_LINE INDENT b = 0 NEW_LINE for c in a0 : NEW_LINE INDENT b += c NEW_LINE DEDENT return sum NEW_LINE DEDENT’, ‘def func (a0)’⟩ → ‘: ’
	⟨ ‘def func (a0) : NEW_LINE INDENT for (b , c) in enumerate (a0) : NEW_LINE INDENT if not c . isalnum ( ) : NEW_LINE INDENT return b - 1 NEW_LINE DEDENT DEDENT return len (a0) - 1 NEW_LINE DEDENT’, ‘def func (a0) :’⟩ → ‘NEW_LINE ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE’⟩ → ‘INDENT ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT’⟩ → ‘for ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for’⟩ → ‘b ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b’⟩ → ‘in ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in’⟩ → ‘range ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range’⟩ → ‘(’
	⟨ ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in a0 : NEW_LINE INDENT sum += b NEW_LINE DEDENT c = d = sum NEW_LINE for b in a0 : NEW_LINE INDENT d -= b NEW_LINE if c == d : NEW_LINE INDENT return b NEW_LINE DEDENT c += b NEW_LINE DEDENT return - 1 NEW_LINE DEDENT’, ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in range (’⟩ → ‘len ’
	⟨ ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in a0 : NEW_LINE INDENT sum += b NEW_LINE DEDENT c = d = sum NEW_LINE for b in a0 : NEW_LINE INDENT d -= b NEW_LINE if c == d : NEW_LINE INDENT return b NEW_LINE DEDENT c += b NEW_LINE DEDENT return - 1 NEW_LINE DEDENT’, ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in range (len’⟩ → ‘(’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (’⟩ → ‘a0 ’
	⟨ ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in a0 : NEW_LINE INDENT sum += b NEW_LINE DEDENT c = d = sum NEW_LINE for b in a0 : NEW_LINE INDENT d -= b NEW_LINE if c == d : NEW_LINE INDENT return b NEW_LINE DEDENT c += b NEW_LINE DEDENT return - 1 NEW_LINE DEDENT’, ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in range (len (a0’⟩ → ‘) ’
	⟨ ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in a0 : NEW_LINE INDENT sum += b NEW_LINE DEDENT c = d = sum NEW_LINE for b in a0 : NEW_LINE INDENT d -= b NEW_LINE if c == d : NEW_LINE INDENT return b NEW_LINE DEDENT c += b NEW_LINE DEDENT return - 1 NEW_LINE DEDENT’, ‘def func (a0) : NEW_LINE INDENT sum = 0 NEW_LINE for b in range (len (a0)’⟩ → ‘) ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0))’⟩ → ‘: ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) :’⟩ → ‘NEW_LINE ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE’⟩ → ‘INDENT ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT’⟩ → ‘a0 ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0’⟩ → ‘[’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0 [’⟩ → ‘b ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0 [b’⟩ → ‘] ’
	⟨ ‘def func (a0) : NEW_LINE INDENT [a0] = 0 NEW_LINE DEDENT’, ‘def func (a0) : NEW_LINE INDENT a0 [0]’⟩ → ‘= ’
	⟨ ‘def func (a0) : NEW_LINE INDENT b = a0 . find (b’ ¿ ’) NEW_LINE c = 7 NEW_LINE return a0 [c : b] NEW_LINE DEDENT ¡/s¿”, ”def func (a0) : NEW_LINE INDENT b = a0 . find (’ ¿ ’) NEW_LINE c = ’ ⟩ → ‘7 ’
	⟨ ‘def func (a0 = None) : NEW_LINE INDENT return 8 if a0 is None else 7 NEW_LINE DEDENT’, ‘def func (a0) : NEW_LINE INDENT return 8 if a0 == 256 else 7’⟩ → ‘NEW_LINE ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0 [b] += a1 NEW_LINE’⟩ → ‘DEDENT ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0 [b] += a1 NEW_LINE DEDENT’⟩ → ‘DEDENT ’
	⟨ ‘def func (a0 , a1) : NEW_LINE INDENT for b in a0 : NEW_LINE INDENT b += a1 NEW_LINE DEDENT DEDENT’, ‘def func (a0 , a1) : NEW_LINE INDENT for b in range (len (a0)) : NEW_LINE INDENT a0 [b] += a1 NEW_LINE DEDENT DEDENT’⟩ → ‘EOS ’

* ⟨·, ·⟩ → ‘* ’ : ⟨·, ·⟩ represents the decision-making basis for each generated token ‘* ’ in the corrected Python function.

* NEW_LINE, INDENT, DEDENT represent newline, indentation, dedent in code formatting, respectively.

* EOS: end of sentence.



(a) Correction performance (b) Translation performance

Figure 4: **The influence of diverse datastore distributed structures.** We find that the distributed architecture can effectively enhance the error correction performance by efficiently learning and applying massive error correction information, thereby improving translation accuracy. Meanwhile, we also explore the impact of the *unified name rule* on the error correction model. This approach exhibits a positive gain effect, effectively improving the model's error correction capabilities.

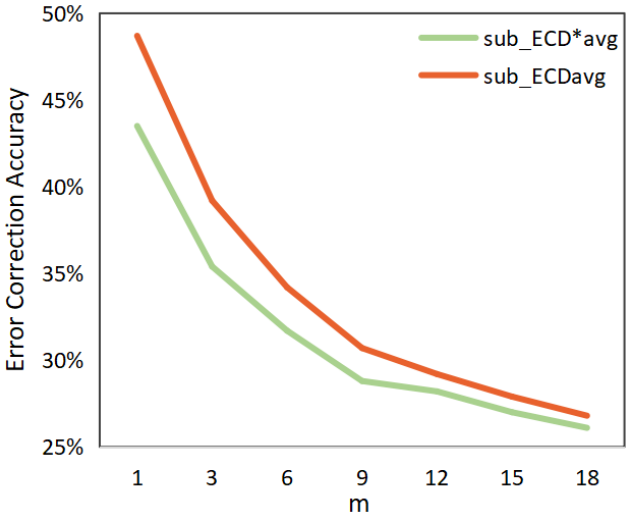


Figure 5: **The influence of datastore capacity on translation accuracy.** We observe a positive correlation between the sub-datastore capacity and error correction performance. The larger the sub-datastore, the higher the error correction capability of the sub-datastore. This finding suggests that the datastore can robustly acquire and integrate correction information from the error correction language pairs.