TIGER: A Generating-Then-Ranking Framework for Practical Python Type Inference

Chong Wang*, Jian Zhang*†, Yiling Lou‡, Mingwei Liu§, Weisong Sun*, Yang Liu*, and Xin Peng‡

*College of Computing and Data Science, Nanyang Technological University, Singapore
{chong.wang, jian_zhang, weisong.sun, yangliu}@ntu.edu.sg
‡School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China
{yilinglou, pengxin}@fudan.edu.cn
§School of Software Engineering, Sun Yat-sen University, China
liumw26@mail.sysu.edu.cn

Abstract—Python's dynamic typing system offers flexibility and expressiveness but can lead to type-related errors, prompting the need for automated type inference to enhance type hinting. While existing learning-based approaches show promising inference accuracy, they struggle with practical challenges in comprehensively handling various types, including complex parameterized types and (unseen) user-defined types.

In this paper, we introduce TIGER, a two-stage *generating-then-ranking* (*GTR*) framework, designed to effectively handle Python's diverse type categories. TIGER leverages fine-tuned pre-trained code models to train a generative model with a span masking objective and a similarity model with a contrastive training objective. This approach allows TIGER to generate a wide range of type candidates, including complex parameterized types in the generating stage, and accurately rank them with user-defined types in the ranking stage. Our evaluation on the ManyTypes4Py dataset shows TIGER's advantage over existing methods in various type categories, notably improving accuracy in inferring user-defined and unseen types by 11.2% and 20.1% respectively in Top-5 Exact Match. Moreover, the experimental results not only demonstrate TIGER's superior performance and efficiency, but also underscore the significance of its generating and ranking stages in enhancing automated type inference.

Index Terms—type inference, pre-trained code models, generating-then-ranking, contrastive learning

I. INTRODUCTION

Python's dynamic typing system, while fostering adaptable and expressive code, can pose challenges like code comprehension, maintainability, and type-related errors [1]–[3]. Python Enhancement Proposals (PEPs) [4], [5] were introduced to enhance Python's type hinting system, advocating for type hints to improve code quality and support tools like static type checking [6]–[9]. Despite this, manual type annotation can be labor-intensive and error-prone [10].

To this end, there is a growing trend towards adopting automated approaches for inferring variable types through contextual analysis, encompassing both rule-based and learning-based methodologies. Rule-based approaches [6]–[9] rely on predefined patterns and rules for accurate type hinting but may lack comprehensive coverage for diverse scenarios [3], [11]. Learning-based approaches, gaining prominence, transform

type inference into different tasks. They can be classified into three types: 1) *Classification-based*. Utilize classification models trained on contextual features to predict variable types [11]–[13]. 2) *Similarity-based*: Utilize deep learning similarity calculation models to assess candidate types based on their similarity scores, ranking them accordingly. 3) *Generation-based*: Utilize generative models to produce types based on given inputs [3], [14]–[16].

However, these learning-based approaches *face limitations in considering various visible types, notably parameterized types and (unseen) user-defined types.* Note that user-defined types in this paper also include third-party types. Classification-based approaches [11]–[13] rely on fixed-sized type vocabularies (*e.g.,* 1,000 types for TypeWriter [12]), leading to numerous out-of-vocabulary (OOV) types, such as complex parameterized types (*e.g.,* "Dict[str, List[int]]") or unseen user-defined types. Similarity-based approaches [1], [2], [17] use import information to gather user-defined types and rank candidate types based on contextual similarities. However, these approaches struggle to cover the myriad parameterized types (*e.g.,* "Union[str, int]") formed by elementary types (*e.g.,* "Union", "str", "int"). Generation-based approaches [3], [14]–[16] can generate creative types, including complex parameterized types, but most struggle with recognizing and predicting unseen user-defined types not in the training set.

Large language models (LLMs) such as ChatGPT are increasingly utilized for type inference tasks. For instance, Peng et al. [3] introduced TypeGen, a generation-based approach that utilizes parameter-frozen LLMs and incorporates visible user-defined types in prompt construction. However, it faces accuracy challenges for function arguments as discussed in their paper [3], due to intricacies and robustness issues with prompt engineering on non-tunable, parameter-frozen LLMs [18]. *Furthermore, ChatGPT-based approaches, including TypeGen, face practical limitations regarding inference cost and scalability.* TypeGen relies on external APIs [19] for interactions with ChatGPT and employs voting mechanisms [20] for accurate inference, resulting in significant time and financial overhead.

To overcome these limitations, we introduce a two-stage framework named *generating-then-ranking* (*GTR* framework),

---

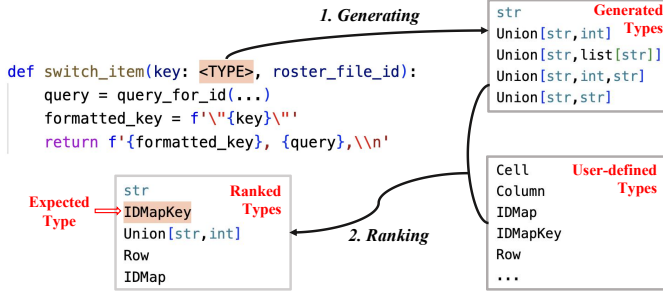† Corresponding author: Jian Zhang (jian_zhang@ntu.edu.sg).

Fig. 1. The *Generating-Then-Ranking* Framework

as depicted in Figure 1. *This framework combines the strengths of both generation-based and similarity-based approaches, allowing for the consideration of comprehensive visible types*. In the initial *generating* stage, our objective is to produce diverse candidate types, including commonly used elementary types like "UUID" and parameterized types like "Union[str,int]", to fill the type placeholders (*i.e.*, "<TYPE>" in Figure 1). Subsequently, in the *ranking* stage, visible user-defined types, potentially unseen during training, such as "IDMap" and "IDMapKey", are effectively ranked alongside the generated candidates based on similarities. Furthermore, for efficiency and cost considerations, we opt for relatively lightweight pre-trained code models like the 220M-parameter CodeT5 [14], specifically fine-tuned for type inference, diverging from the use of LLMs like ChatGPT.

We present TIGER, a novel <u>T</u>ype <u>I</u>nference approach that implements the <u>GE</u>nerating-Then-<u>R</u>anking framework. To facilitate the online *GTR* inference depicted in Figure 1, TIGER initiates with offline training of a generation model and a similarity model with collected annotated Python functions. During the generation model training, TIGER masks annotated types with type placeholders to create type-missed functions. These functions are employed with the span masking objective [21] to train the model in generating types to fill the placeholders. For training the similarity model, TIGER analyzes import statements to identify visible user-defined types, which are combined with the types generated by the generation model to form a set of candidate types. Subsequently, a contrastive training objective [22]–[25] is utilized to train the model to learn robust representations for both type-missed functions and candidate types [26], facilitating effective measurement of contextual similarities between them. During online inference, armed with the trained models, TIGER performs the two-stage generating-then-ranking process. Given a type-missed function containing a type placeholder, this process generates type candidates using the generation model. These are then ranked based on their similarity to user-defined types visible in the scope, using the similarity model.

We extensively evaluate TIGER's inference effectiveness and efficiency on the ManyTypes4Py dataset [27], a widely adopted benchmark in previous studies. Our results demonstrate TIGER's superiority in inference accuracy across all type categories compared to six baseline models. Specifically, TIGER achieves an impressive accuracy rate of 94.1% and 85.0% in Top-5 Exact Match for predicting user-defined and unseen types, respectively, marking significant improvements of 11.2% and 20.1% over the best baseline approach (RQ1). Furthermore, TIGER exhibits enhanced robustness across various variable categories, encompassing local variables, function arguments, and return values (RQ2). We also assess the efficiency of TIGER, highlighting its potential for optimization in large-scale inference scenarios (RQ3). Finally, our ablation study validates the significant contributions of both the generating and ranking stages to TIGER's overall performance (RQ4).

To summarize, this paper makes the following contributions:

- A novel two-stage framework, "Generating-Then-Ranking", which systematically generates and ranks candidate types alongside user-defined types, facilitating a comprehensive consideration of visible candidate types.
- TIGER, a practical Python type inference approach that implements the two-stage framework through fine-tuning pre-trained code models with specific training objectives.
- Extensive experiments on the ManyTypes4Py benchmark demonstrate the effectiveness and robustness of TIGER across diverse type and variable categories, particularly for (unseen) user-defined types prevalent in real-world scenarios. Additionally, efficiency evaluations of TIGER showcase its potential for optimization in large-scale inference scenarios.

II. RELATED WORK

A. Rule-based Type Inference

Rule-based methods in type inference rely on predefined rules for inferring variable types. A rule can only be activated when all the underlying premises are known, after which it determines the type of the variable based on a set conclusion [3]. To address the imperative for static type hints within dynamically typed programming languages, numerous techniques have emerged for the purpose of type inference and verification. Notable examples include Pyright and Pylance (Microsoft) [8], Pyre (Meta) [7], pytype (Google) [9], and Python's official type checker, Mypy [6]. Apart from industry tools, several academic methodologies have emerged for type inference across various programming languages like Python and JavaScript [28]–[33]. While these approaches excel in terms of accuracy, they confront a significant challenge stemming from dynamic language features and external function calls [17]. This challenge manifests as a limitation in their coverage, hindering their ability to provide comprehensive type inference scenarios.

B. Learning-based Type Inference

Classification-based Approaches. In addressing the type inference problem, supervised learning has been employed by researchers with large-scale training data to train classification-based models. Pradel *et al.* [12] introduced TypeWriter, which utilizes probabilistic type prediction and search-based refinement to infer function types from partially annotated code and validate them using a gradual type checker. Wei *et*

al. [34] introduced a probabilistic type inference approach for TypeScript that employs a graph neural network to analyze a program's type dependency graph, facilitating predictions of both standard and user-defined types. Yan *et al.* [13] proposed DLInfer, an approach that collects slice statements for variables through static analysis and employs a bi-directional gated recurrent unit (GRU) model to learn type propagation information for inference. Additionally, TypeBert [11] demonstrated that by harnessing the token-sequence inductive bias found in BERT-style models and having access to ample data, it is feasible to surpass the type-annotation performance of even the most advanced models. These approaches encounter practical challenges when dealing with OOV types, including unseen user or library-defined types. As a result, they struggle to accurately predict types for a significant portion of variables in real-world projects.

Similarity-based Approaches. Allamanis *et al.* [1] presented a graph neural network model that leverages probabilistic reasoning and deep similarity learning to predict types, including rare and user-defined types, based on a program's structure, names, and patterns. Mir *et al.* [2] proposed Type4Py, a hierarchical neural network model employing deep similarity learning to infer likely types for program elements by distinguishing between similar and dissimilar types in a high-dimensional space. Peng *et al.* introduced HiTyper, a hybrid type inference approach combining static inference and similarity-based models by leveraging type dependency graphs (TDGs) to record and integrate type dependencies among variables, facilitating iterative static inference and neural predictions until the complete inference of TDGs. While these approaches demonstrate the capability to handle arbitrary types, they require pre-determined candidates, limiting their effectiveness in adequately considering numerous parameterized types.

Generation-based Approaches. Following the type annotation conventions recommended in PEPs, the Python type inference problem can be transformed into a conditional generation problem. For instance, it can be treated as a cloze-style fill-in-blank task where a type placeholder is inserted after a variable that requires annotation. Pre-trained code models [14]–[16], [21] are then employed to generate types fill in the placeholder with types based on their learned code naturalness. To fully leverage their generation capability, some studies [35] have fine-tuned these pre-trained models to improve inference accuracy. These generation-based approaches, with fill-in-blank task formulation, lack awareness of unseen user or library-defined types. In a recent study, Peng *et al.* [3] introduced a generation-based approach known as TypeGen for Python type inference, which relies on parameter-forzen LLMs like `gpt-3.5` and `gpt-4`. TypeGen integrates type dependencies derived from lightweight static analysis with in-context learning [36] to construct few-shot Chain-of-Thought (CoT) prompts. These prompts are employed to generate both type predictions and accompanying explanations. While this LLM-based generative approach demonstrates notable effectiveness in type inference, it faces practical challenges,

```
def switch_item(key: <TYPE>②, roster_file_id) -> <TYPE>③:
    query: <TYPE>① = query_for_id(...)
    formatted_key = f'\"{key}\"'
    return f'{formatted_key}, {query},\\n'
```

Fig. 2. Type Placeholders for Three Categories of Variables

particularly with regard to the scalability concerns arising from the resource-intensive interactions with expensive LLMs.

C. Pre-trained Code Models in SE

Recent research [14], [37], [38] indicates that pre-trained code models encapsulate valuable information about code syntax and semantics, identifier and namespace concepts, and natural language naming. To harness this rich code information for downstream software engineering tasks, researchers commonly fine-tune these models on task-specific data. Examples of such tasks include code generation/completion [39]–[42], code search [43], [44], defect detection [45], vulnerability detection/location [46], and program repair [47], [48]. Researchers have explored the application of prompt-based approaches with pre-trained code models in various software engineering tasks. For instance, Wang et al. [49] investigated the effectiveness of prompt learning in code intelligence tasks, such as clone detection and code summarization. Another study by Wang et al. [50] utilized pre-trained code models as knowledge bases and employed prompt learning for variable explanation. In this study, we strategically fine-tune pre-trained code models to facilitate practical type inference.

III. PROBLEM DEFINITION

We define the type inference problem within a Python function as a cloze-style *fill-in-blank* task [14], [16], [21], [51], focusing on three categories of variables: local variables, function arguments, and return types. To annotate the type for a target variable, we insert a type placeholder based on its category, adhering to the annotation guidelines outlined in the PEPs. In Figure 2, the three numbered "`<TYPE>`" represent the type placeholders inserted for local variables, function arguments, and return types, respectively. In practical applications, we address one target variable in a given Python function at a time, following the approach adopted in previous studies [2], [3], [12]. Functions containing type placeholders are referred to as *type-missed functions*, where suitable types need to be predicted to fill the placeholders with difference categories of types, including builtin types, parameterized types, and use-defined or third-party types.

IV. APPROACH

The overview of TIGER is presented in Figure 3. TIGER first offline trains a generation model and a similarity model by fine-tuning existing pre-trained code models (*i.e.,* base models) using annotated Python functions (*i.e.,*① and ②). The trained models are used to support the online two-stage *generating-then-ranking* inference (*i.e.,*③).
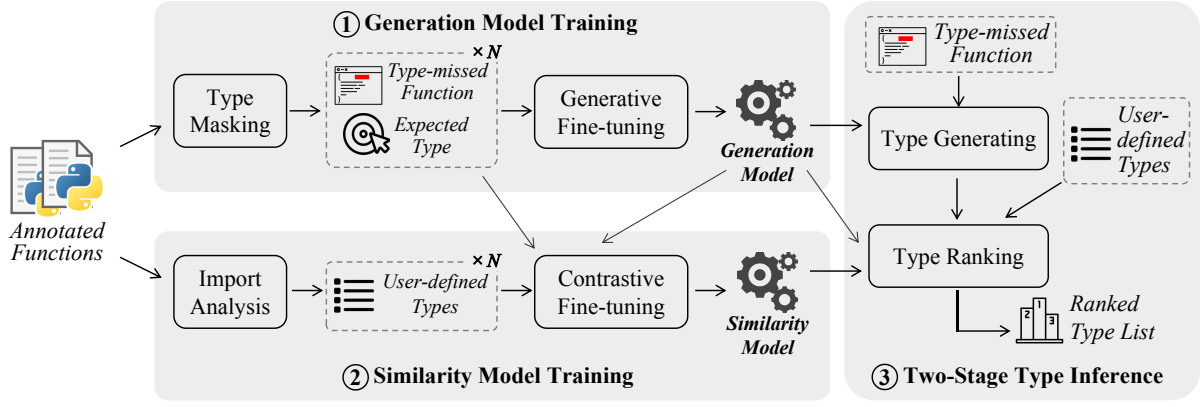
Fig. 3. Overview of TIGER

① When training the generation model, TIGER employs a process where it masks variable types in annotated functions, replacing them with type placeholders to create type-missed functions. These type-missed functions, along with the corresponding expected types, are then utilized for fine-tuning a base model. The fine-tuning process is guided by a generative span masking objective [14], which directs the model to generate appropriate types to fill the placeholders.

② For the training of the similarity model, TIGER conducts an analysis of import statements to identify user-defined types available for the type-missed functions. These identified types are then combined with the types generated by the generation model to form a comprehensive set of candidate types. TIGER employs a contrastive training objective [22]–[25] to fine-tune a base model, instructing it to differentiate expected types from all candidate types based on their contextual similarities with the type-missed functions.

③ With the trained generation model and similarity model, TIGER executes type inference for a given type-missed function that contains a type placeholder. This process adheres to the *GTR* inference framework, which encompasses both the generation of candidate types and their subsequent ranking alongside available user-defined types. In the ranking stage, candidates are ranked by combining the generative likelihood and the contextual similarity yeilded by the generation model and similarity model, respectively.

A. Model Architecture Selection

We select encoder-decoder architecture models, such as CodeT5 [14], as the base model for the *GTR* framework and provide a concise overview of its encoding-decoding process.

1) Encoder-Decoder Model: Encoder-decoder models typically undergo specific pre-training on a code corpus with the masked span prediction task, compelling the model to fill in masked span (token sequence) placeholders within the code. Moreover, with the encoder-decoder architecture, such models can capture bidirectional context within code while generating token sequences of arbitrary length and combination to fill the placeholders [50]. This dual capability perfectly aligns with our fill-in-blank type inference task, distinguishing it from encoder-only models (*e.g.,* CodeBERT [21] and GraphCodeBERT [51]) and decoder-only models (*e.g.,* InCoder [16]).

2) Encoding-Decoding Process: The encoding-decoding process of an encoder-decoder model is a critical aspect for the fill-in-blank task of type inference.

- **Encoding.** After tokenizing a given type-missed function into a token sequence $X = [\texttt{<BOS>}, x_1, x_2, ..., x_M, \texttt{<EOS>}]$ (where "$\texttt{<BOS>}$" and "$\texttt{<EOS>}$" denote the beginning and end of the sequence), the encoder processes X and generates corresponding hidden states $\boldsymbol{H}_X = [\boldsymbol{hx}_B, \boldsymbol{hx}_1, \boldsymbol{hx}_2, ..., \boldsymbol{hx}_M, \boldsymbol{hx}_E]$ (*i.e.,* feature vectors). These vectors capture bidirectional contextual information from X, and the encoding process is mathematically expressed as:

$$\boldsymbol{H}_X \leftarrow \text{Encoder}(X) \tag{1}$$

- **Decoding.** The resulting $\boldsymbol{H}_X$ is then fed into the decoder, tasked with sequentially processing an input partial token sequence and predicting the next token. Specifically, at each decoding step (the t-th step), the decoder takes $\boldsymbol{H}_X$ and the partial sequence $Y^{:t-1} = [\texttt{<BOS>}, y_1, y_2, ..., y_{t-1}]$ as input to compute the corresponding hidden states $\boldsymbol{H}_Y^{:t-1} = [\boldsymbol{hy}_B, \boldsymbol{hy}_1, \boldsymbol{hy}_2, ..., \boldsymbol{hy}_{t-1}]$ for $Y^{:t-1}$:

$$\boldsymbol{H}_Y^{:t-1} \leftarrow \text{Decoder}(\boldsymbol{H}_X, Y^{:t-1}) \tag{2}$$

Subsequently, the decoder then utilizes a classification head (*i.e.,* linear layers) on $\boldsymbol{hy}_{t-1}$ to predict a $|\mathcal{V}|$-dimensional probability distribution $\boldsymbol{p}_t$, corresponding to the token vocabulary $\mathcal{V}$, as expressed by:

$$\boldsymbol{p}_t \leftarrow \text{Head}(\boldsymbol{hy}_{t-1}) \tag{3}$$

Guided by the distribution $\boldsymbol{p}$, the decoder predicts the next token y_t', either by greedily selecting the token with the highest probability or employing specific sampling techniques [52]–[54].

The decoding process concludes when the "$\texttt{<EOS>}$" is fed. After finishing all the decoding steps on the input token sequence $Y = [\texttt{<BOS>}, y_1, y_2, ..., y_N]$, we denote the corresponding hidden states and predicted token sequences

Fig. 4. Type Annotation Masking with Type Placeholder

by the decoder as $\boldsymbol{H}_Y = [\boldsymbol{hy}_B, \boldsymbol{hy}_1, \boldsymbol{hy}_2, ..., \boldsymbol{hy}_N]$ and $Y' = [y'_1, y'_2, ..., y'_N, \texttt{<EOS>}]$, respectively.

B. Generation Model Training

The training of the generation model within TIGER commences with the retrieval of type annotations present in Python functions, adhering to the guidelines outlined in the PEP proposals. Subsequently, fine-tuning of a base model (*i.e.,* a pre-trained encoder-decoder model) is carried out, employing a generative span masking objective [14], [55].

1) Type Annotation Masking: Given a Python function with annotated local variables, function arguments, and return types following PEPs' guidelines, TIGER systematically processes these type annotations. It applies a masking procedure to each annotation, replacing them with the placeholder "<TYPE>". This operation results in the creation of a type-missed function, complemented by the corresponding expected type. For instance, by masking the type annotation "IDMapKey" associated with the function argument "key", TIGER generates a type-missed function containing a type placeholder, as illustrated in Figure 4, accompanied by the expected type "IDMapKey". After processing all annotated Python functions, TIGER obtains a training data that consists of type-missed functions and the corresponding expected types.

2) Generative Fine-tuning: TIGER proceeds with the training of the generation model, involving the fine-tuning of a base model based on acquired type-missed functions and their corresponding expected types. For each pair comprising a type-missed function $func$ and its corresponding expected type $type$, TIGER initiates the process by tokenizing them into their respective token sequences, denoted as X and Y, using the model's tokenizer. Subsequently, TIGER employs the encoding-decoding process of the base model, as detailed in Section IV-A2, and calculates the loss.

Specifically, TIGER employs the encoder to generate the hidden states $\boldsymbol{H}_X$ for X (*cf.* Equation 1). Subsequently, in the decoding process, it feeds the first $(t-1)$ tokens in Y (*i.e.,* $Y^{:t-1}$) into the decoder, computing the probability distribution $\boldsymbol{p}_t$ using Equations 2 and 3. Following this, the cross-entropy loss is computed based on the expected next token y_t and $\boldsymbol{p}_t$, adhering to the span masking objective [14], [55] that is commonly utilized in the pre-training encoder-decoder models [14]. The fine-tuning process involves minimizing this loss, guiding the optimization of model parameters using the Adam optimizer [56], which aims to enhance the model's capability to maximize the probability of predicting y_t within $\boldsymbol{p}_t$.

3) Resulting Generation Model: After fine-tuning, the resulting generation model in TIGER serves two primary purposes:

- **Candidate Type Generation.** When presented with a type-missed function $func$, TIGER initially tokenizes it into the corresponding token sequence X. Subsequently, TIGER feeds X into the generation model's encoder and utilizes the decoder to generate a token sequence Y', adhering to the encoding-decoding process outlined in Section IV-A2. It is essential to note that, during the decoding process, the previously predicted tokens serve as the decoder's input to predict the next token. In other words, when predicting the next token y'_t at the t-th decoding step, the input partial token sequence $Y^{:t-1}$ (*cf.* Section IV-A2) comprises the previous predicted $(t-1)$ tokens (*i.e.,* $Y^{:t-1} = [\texttt{<BOS>}, y'_1, ..., y'_{t-1}]$) by the decoder. Upon completing the decoding process, the generated token sequence Y' undergoes conversion into a candidate type. Additionally, TIGER integrates beam search into the decoding process, generating multiple token sequences and, consequently, yielding multiple candidate types for the placeholder within the given type-missed function.

- **Generative Likelihood Computation.** When presented with a type-missed function $func$ and a candidate type $type$, TIGER initiates the process by tokenizing them into the corresponding token sequences X and Y. Employing the encoding-decoding process detailed in Section IV-A2, TIGER feeds X into the generation model's encoder. Subsequently, it sequentially inputs the tokens in Y into the decoder to compute the generative likelihood of Y. Specifically, at the t-th decoding step, the decoder takes as input the initial $t-1$ tokens in Y, *i.e.,* $Y^{:t-1} = [\texttt{<BOS>}, y_1, ..., y_{t-1}]$, and predicts the probability distribution $\boldsymbol{p}_t$. TIGER retrieves the probability of the next token y_t in Y from $\boldsymbol{p}_t$, denoted as $p(y_t)$, and computes the generative likelihood of $type$ by multiplying all probabilities when finishing inputting all decoding steps on Y:

$$lik(func, type) = \prod_{y_t \in Y} p(y_t) \qquad (4)$$

C. Similarly Model Training

The purpose of the similarity model is to effectively gauge the contextual similarity between the given type-missed function and the candidate types. To achieve this, TIGER employs a contrastive training objective [22]–[25], which seeks to cultivate generalizable and robust representations for both type-missed functions and candidate types. This objective achieves this goal by maximizing the similarity between the type-missed function and the expected type while minimizing the similarity between the type-missed function and other candidate types.

1) Import Analysis: TIGER incorporates user-defined types for the type placeholder within the given type-missed function while constructing the training dataset for contrastive learning. This data is derived by analyzing import statements existing in the same project environment as the type-missed function,

in accordance with established practices [3], [57]. Specifically, TIGER parses the current source file containing the target function and gathers candidate user-defined types from two distinct sources. Firstly, all type definitions in the current file are directly considered as candidate user-defined types. Additionally, TIGER examines import statements in the current file to identify the source files being imported. It then includes the types defined in these imported files in the list of candidate user-defined types. The gathered user-defined types might be project-specific, with names that are not encountered (unseen) in other projects.

2) Contrastive Fine-tuning: The fine-tuning process for the similarity model consists of three pivotal components: negative instance construction, similarity computation, and model training. Adhering to the terminology commonly used in contrastive training conventions, we designate the type-missed function as the *anchor*, the corresponding expected type as the *positive instance*, and other candidates as *negative instances*.

- **Negative Instance Construction.** Constructing suitable negative instances for the expected type is crucial for effective contrastive training. TIGER achieves this by generating K negative instances from two distinct sources. For the given type-missed function (*i.e.,* anchor), the first source utilizes the trained generation model, employing a beam search algorithm during model decoding to generate K candidates (*cf.* Section IV-B3). These candidates typically encompass builtin types, commonly used third-party types, and parameterized types. The second source involves user-defined types obtained through import analysis. TIGER combines the types from both sources, excluding the positive instance, and randomly selects K ones as the negative instances.

- **Similarity Computation.** To quantify contextual similarity between the anchor, namely $func$, and a positive/negative instance, namely $type$, TIGER tokenizes them into their respective token sequences, X and Y. Subsequently, these sequences undergo processing by a pre-trained base model (*i.e.,* a pre-trained encoder-decoder model), executing the encoding-decoding process described in Section IV-A2. Upon completion of the process, TIGER computes the average of the hidden states $\boldsymbol{H_X}$ produced by the encoder as the vector representation of $func$, and the average of the hidden states $\boldsymbol{H_Y}$ produced by the decoder as the vector representation of $type$. The two representations, namely $\text{AVG}(\boldsymbol{H_X})$ and $\text{AVG}(\boldsymbol{H_Y})$, are employed to assess the contextual similarity between $func$ and $type$ by calculating their cosine similarity:

$$\text{sim}(func, type) = \text{cosine}(\text{AVG}(\boldsymbol{H_X}), \text{AVG}(\boldsymbol{H_Y})) \quad (5)$$

- **Model Training.** Given the anchor $func$, TIGER computes the InfoNCE loss [23], [25] based on it, its corresponding positive instance pos, and the generated negative instances $\mathbb{N}$. The loss function is defined as follows:

$$\mathcal{L} = -\log \frac{e^{\text{sim}(func, pos)}}{e^{\text{sim}(func, pos)} + \sum_{neg \in \mathbb{N}} e^{\text{sim}(func, neg)}}$$

Maximizing this loss function guides the model to enhance the disparity between the similarities of the positive instance and the negative instances, allowing the model to learn to distinguish the most suitable type for the given type-missed function. The optimization process utilizes the Adam optimizer [56] to update the model's parameters.

3) Resulting Similarity Model: Following fine-tuning, the resulting similarity model demonstrates an effective capability to measure the contextual similarity between the provided type-missed function and its associated candidate types, as depicted in Equation 5. Due to the contrastive training objective and the incorporation of negative instances from two distinct sources, the similarity model exhibits strong generalizability and robustness [23], encompassing both generated candidates and user-defined types.

D. Two-Stage Type Inference

Leveraging both the generation model and the similarity model, TIGER employs the two-stage *GTR* strategy for type inference, as depicted in Figure 1.

1) Type Generating: When presented with a function and a variable slated for annotation, TIGER initially inserts a type placeholder after the variable based on its category (*i.e.,* local variable, function argument, or return value), creating a type-missed function $func$ akin to the example in Figure 4. Subsequently, this type-missed function undergoes processing by the generation model, employing beam search during decoding to generate K token sequences, thereby forming K candidate types (*cf.* Section IV-B3).

2) Type Ranking: TIGER conducts import analysis, as detailed in Section IV-C1, to acquire user-defined types available for the type placeholder in $func$. These user-defined types, combined with the generated candidates, constitute a more comprehensive candidate pool. Note that if a generated candidate is neither a built-in type nor a parameterized type whose base is a built-in type, and it is not found in the user or library-defined types, we exclude it from the candidate pool. Subsequently, TIGER ranks all candidates in the pool by considering both generative likelihood and contextual similarity for each candidate. For a given candidate $cand$ in the pool, TIGER computes its generative likelihood using the generation model (*cf.* Section IV-B3) and its contextual similarity with $func$ using the similarity model (*cf.* Section IV-C3).

The final score of candidate $cand$ is determined by the sum of generative likelihood $\text{lik}(func, cand)$ and contextual similarity $\text{sim}(func, cand)$, expressed as:

$$\text{score}(cand) = \text{lik}(func, cand) + \text{sim}(func, cand) \quad (6)$$

Ultimately, all candidates for the given type-missed function are ranked based on their scores in descending order, forming a ranked list of types for the variable intended for annotation.

The inclusion of both generative likelihood and contextual similarity in the score calculation is motivated by specific considerations, inspired by the methodology utilized in contrastive text generation [26], [58]. Solely relying on generative likelihood, as discussed in Section IV-B3, could lead to

biased rankings. During training, the decoding process is guided by the ground truth input, which is absent during generation [59]. This mismatch introduces exposure bias, making the decoding process less robust in distinguishing between "good" and "bad" tokens in the certain decoding steps. In contrast, the similarity model generates robust and generalizable representations for both the generated candidates and the user-defined types. This is achieved by combining these two sources during the construction of negative instances for contrastive learning [23]. Therefore, by integrating both generative likelihood and contextual similarity, TIGER capitalizes on the task alignment benefits of the generation model and the generalizability and robustness of the similarity model. This dual consideration enhances the overall effectiveness of the *GTR* inference process.

V. Evaluation

We conduct extensive experiments to assess the effectiveness of TIGER in Python type inference from various dimensions. The research questions are summarized as follows:

- **RQ1 (Effectiveness)**: How effectively can TIGER infer types for variables? How effectively can TIGER handle with various categories of types, especially parameterized types and (unseen) user defined types?
- **RQ2 (Robustness)**: How robust is TIGER in inferring types for different categories of variables, including local variables, function arguments, and return types?
- **RQ3 (Efficiency)**: How effectively can TIGER infer types for different sources of types, encompassing elementary types, parameterized types, and (unseen) user-defined types?
- **RQ4 (Ablation Study)**: Are the two stages in *GTR* useful for type inference?

A. Setup

We present the experimental setup for the evaluation, encompassing the implementation, dataset, baselines, and metrics.

1) Implementation: In our current implementation, we utilize CodeT5 as the base model for training both the generation and similarity models, given its widespread use as an encoder-decoder pre-trained model for code. We implement TIGER in Python using the Transformers library [60], a widely-used library for language models. The base models for training the generation and similarity models is downloaded from the Hugging Face Hub [61]. Following established practices [2], [3], [17], we consider the top-5 candidate types for predictions, setting $K = 5$ (*i.e.,* beam size) during both the training and inference stages of TIGER. The training hyperparameters for both the generation model and similarity model include a training epoch of 3, a learning rate of 1e-5, and a training batch size of 8.

2) Dataset: In line with prior studies [2], [17], we conduct the evaluation on the ManyTypes4Py dataset [27], which is partitioned into training and test sets with an 8:2 project ratio (no overlapping projects between training set and test set), resulting in 242,954 instances for training and 85,205 instances

TABLE I
OVERVIEW OF TRAINING AND TEST SETS.

Dataset	Type Category			Variable Category		
	#Ele	#Par	#Usr (#Unseen)	#Var	#Arg	#Ret
Training	128,006	67,185	47,763 (-)	172,459	48,461	22,034
(242,954)	52.7%	27.6%	19.7% (-)	70.9%	20.0%	9.1%
Test	5,199	2,748	2,053 (579)	7,091	1,995	914
(10,000)	52.0%	27.5%	20.5% (5.8%)	70.9%	20.0%	9.1%

for testing. For the TypeGen evaluation, the authors further sampled 10,000 instances from the entire test set to reduce the computational cost associated with calling large language models (LLMs). To maintain consistency in our experimental setup, we employ this sampled subset [62] as the definitive test set for all experiments pertaining to RQ1-RQ4. Among the test instances involving user-defined types, 579 instances require the inference of types that are **previously unseen in training set**. These unseen types are challenging to predict and provide insights into the practicality of the inference approach. The training set is utilized for training both TIGER and baselines, as well as providing few-shot examples for the prompt construction in TypeGen. The test set is then employed to assess the type inference effectiveness of TIGER and baselines. Table I provides an overview of the training and test sets, illustrating the distributions of different type and variable categories. "Ele", "Par", and "Usr (Unseen)" represent different type categories [3], denoting Elementary, Parameterized, and (unseen) user-defined Types, respectively. "Var", "Arg", and "Ret" represent different variable categories [3], denoting Local Variables, Function Arguments, and Return Values, respectively.

3) Baselines: We include several state-of-the-art learning-based type inference approaches as baselines. We omit rule-based approaches from consideration, given that learning-based methods have demonstrated superior effectiveness in prior studies [17].

- **TypeWriter** [12]: A classification-based type inference approach that utilizes RNNs to encode various code features (*e.g.,* identifiers and code tokens) to predict types for target variables. Note that TypeWriter cannot process local variables; therefore, the metrics are calculated only on the subset consisting of function arguments and return types.
- **Type4Py** [2]: A similarity-based type inference approach classifies new Python programs by assigning them to a specific type clusters, using those clusters as the predicted types for the target variables.
- **HiTyper** [17]: A hybrid type inference approach combining static inference and similarity-based models by leveraging type dependency graphs (TDGs) to record and integrate type dependencies among variables.
- **CodeT5-*zs***: A pre-trained CodeT5-base model [21] utilized for zero-shot variable type inference, treating the problem as a cloze-style fill-in-the-blank task, similar to

TIGER.

- **CodeT5-*ft***: A fine-tuned CodeT5-base model [21] trained with the same training data of TIGER, also treating the problem as a cloze-style fill-in-the-blank task.
- **TypeGen** [3]: An LLM-generation-based type inference approach that leverages prompt engineering techniques, including in-context learning and chain-of-thoughts, to harness the capability of LLM for generating types for target variables.

For the baselines, we utilize the released replication implementations and data. We opt not to compare our approach with the other two learning-based type inference methods, namely the classification-based **DLInfer** [13] and the generation-based **TypeT5** [35]. These approaches heavily rely on static analysis and were originally trained and evaluated on smaller datasets (700 and 663 projects, respectively). Despite our attempt to apply DLInfer and TypeT5 to the larger ManyTypes4Py dataset (5,996 projects), scalability issues rendered them inapplicable. Additionally, methodological differences, such as DLInfer segmenting source code into syntax-broken short snippets (*e.g.,* incomplete statements) and TypeT5 not processing local variables within functions, make a fair comparison challenging.

4) Metrics: Following established practices [2], [3], [17], we employ two commonly used metrics to evaluate the inference accuracy of TIGER and the baselines:

- **Exact Match (EM)**: This metric calculates the ratio of type predictions made by an approach that *exactly* matches the type annotated by developers.
- **Base Match (BM)**: This metric calculates the ratio of type predictions made by an approach that shares the common base type (*i.e.,* the outermost type) with the type annotations provided by developers. For example, "Union[str,list]" is base-matched, but not exact-matched, with "Union[str,int]".

B. Effectiveness (RQ1)

We evaluate the effectiveness of TIGER and the six baseline approaches in Python type inference. The detailed inference accuracy of TIGER and the baselines on the test set consisting of 10,000 instances is provided in Table II.

1) Overall Results: Across the entire test set (Column "All"), our approach, TIGER, demonstrates Top-1, 3, and 5 Exact Match metrics of 86.2%, 93.2%, and 94.6%, respectively, and Top-1, 3, and 5 Base Match metrics of 92.3%, 96.2%, and 97.4%, respectively. Considering fine-grained type categories (Columns "Ele", "Par", and "Usr (Unseen)"), TIGER achieves either the best or second-best results in terms of Exact Match and Base Match. Particularly noteworthy are the substantial improvements (see Rows "Δ compared with the best baseline") in almost all type categories, especially in unseen user-defined types (up by 20.1% and 23.4% in Top-5 Exact and Base Match, respectively), with minimal declines (less than 1%) observed in few type categories such as Top-1 Exact Match for elementary types (Ele).

An interesting observation is that the Base Match metrics of all the approaches are notably higher than the Exact Match metrics for parameterized types (*i.e.,* Column "Par") compared with the other two type categories Columns "Ele" and "Usr (Unseen)"). This discrepancy arises because inference approaches correctly predict the base type "Base" but face challenges predicting the specific combinations of "[T1,T2,...]".

2) Comparison with Classification-based Approach: TIGER significantly outperforms the classification-based baseline approach, TypeWriter, for all type categories in terms of Exact Match and Base Match metrics. Specifically, in Top-5 Exact Match, TIGER demonstrates a significant improvement over TypeWriter, with an overall enhancements of 40.6% across the entire test set and ranging from 15.6% to 132.9% for fine-grained type categories.

The superiority is attributed to the limitations of classification-based approaches, such as TypeWriter, which rely on fixed-sized type vocabularies. Such vocabularies often fail to comprehensively cover diverse parameterized types and user-defined types in their visible candidate set. TIGER addresses this limitation by leveraging the capacity of its generating stage to produce complex parameterized types and the ranking capability of its ranking stage to handle (unseen) user-defined types. Notably, while TypeWriter achieves about 30%-40% accuracy for unseen types, as its 1,000-type vocabulary covers a portion of unseen types in the ManyType4Py test set, it is unable to predict types beyond its vocabulary limit.

3) Comparison with Similarity-based Approaches: While demonstrating comparable accuracy for elementary types, TIGER significantly outperforms the similarity-based baseline approaches, Type4Py and HiTyper, for parameterized types and (unseen) user-defined types in terms of Exact Match and Base Match metrics. Specifically, in Top-5 Exact Match, TIGER achieves notable improvements over Type4Py and HiTyper, with enhancements of 17.8% and 18.8% respectively across the entire test set, ranging from 21.6% to 679.8% over Type4Py and from 31.4% to 47.9% over HiTyper for parameterized types and user-defined types.

The enhancements achieved by TIGER can be attributed to the following reasons. While similarity-based approaches can handle commonly used parameterized types (*e.g.,* "list[str]"), they struggle with inferring diverse parameterized types due to the vast number of possible combinations. In contrast, TIGER leverages the creative type generation capacity of its generation model during the generating stage to overcome this limitation. Moreover, in the case of (unseen) user-defined types, TIGER's superior performance is primarily attributed to its robustness and generalizability in the ranking stage, facilitated by the contrastive training objective [23].

4) Comparison with Generation-based Approaches: Among the three generation-based baseline approaches, CodeT5-*zs* exhibits poor accuracy due to its application in a zero-shot setting, while the fine-tuned CodeT5-*ft* and TypeGen achieve the second-best or the co-best results for specific type categories and metrics. Specifically, CodeT5-*ft* demonstrates effectiveness for elementary types (Column "Ele") and parameterized types (Column "Par"), but is relatively ineffective for user-defined types (Column "Usr (Unseen)"); whereas TypeGen achieves

Metric	Approach	Catetory	Top-1 (%)				Top-3 (%)				Top-5 (%)			
			Ele	Par	Usr (Unseen)	All	Ele	Par	Usr (Unseen)	All	Ele	Par	Usr (Unseen)	All
Exact Match (EM)	TypeWriter*	CLS	69.9	43.0	36.5 (30.0)	55.0	78.8	53.4	39.3 (32.6)	62.4	84.5	59.7	42.4 (36.5)	67.3
	Type4Py	SIM	95.1	67.9	43.0 (6.9)	76.9	95.3	71.7	49.5 (10.0)	79.5	95.4	73.5	50.9 (10.9)	80.3
	HiTyper*		94.0	60.0	61.3 (51.3)	76.3	94.2	60.4	68.0 (56.3)	78.7	94.3	60.7	71.6 (57.5)	79.6
	CodeT5-zs	GEN	34.1	15.9	35.3 (33.2)	29.4	33.0	18.9	36.5 (37.7)	29.9	33.5	21.2	37.7 (38.7)	31.0
	CodeT5-ft		94.0	73.9	73.2 (61.3)	84.2	97.0	86.7	78.8 (66.3)	90.4	97.7	89.4	80.8 (67.7)	91.9
	TypeGen		89.8	60.0	79.0 (63.6)	79.2	93.8	74.6	83.7 (69.9)	86.2	94.3	77.8	84.6 (70.8)	87.5
	TIGER	GEN+SIM	94.5	75.1	80.2 (67.5)	86.2	97.5	87.2	90.2 (79.8)	93.2	97.7	89.4	94.1 (85.0)	94.6
	Δ to the best baseline		↓0.6	↑1.6	↑1.5 (↑6.1)	↑2.4	↑0.5	↑0.6	↑7.8 (↑14.2)	↑3.1	0.0	0.0	↑11.2 (↑20.1)	↑2.9
Base Match (BM)	TypeWriter*	CLS	71.9	49.6	37.3 (31.8)	57.5	82.1	68.6	41.2 (36.5)	67.4	88.3	79.5	45.8 (42.5)	73.9
	Type4Py	SIM	95.3	79.2	45.3 (10.2)	80.6	95.7	85.4	53.8 (17.8)	84.3	95.7	87.9	56.6 (21.4)	85.5
	HiTyper*		94.4	84.5	61.3 (51.3)	85.1	94.6	88.2	68.0 (56.3)	87.6	94.8	90.4	71.6 (57.5)	89.1
	CodeT5-zs	GEN	38.2	53.4	36.8 (36.8)	42.1	37.7	57.3	39.4 (42.1)	43.4	39.1	60.5	42.2 (44.7)	45.6
	CodeT5-ft		95.5	93.3	73.8 (63.0)	90.5	97.5	96.9	79.4 (67.9)	93.7	98.0	97.8	81.5 (69.6)	94.6
	TypeGen		91.3	87.1	79.6 (63.9)	87.4	94.7	91.8	84.4 (70.6)	91.5	95.1	92.6	85.1 (71.5)	92.0
	TIGER	GEN+SIM	96.2	93.2	81.3 (70.5)	92.3	97.9	96.8	91.0 (82.4)	96.2	98.0	97.9	95.3 (88.4)	97.4
	Δ to the best baseline		↑0.7	↓0.1	↑2.1 (↑10.3)	↑2.0	↑0.4	↓0.1	↑7.8 (↑16.7)	↑2.7	0.0	↑0.1	↑12.0 (↑23.6)	↑3.0

* TypeWriter and HiTyper are capable of inferring types for 2,909 and 1,032 test instances, respectively. Consequently, their metrics are calculated using only these instances.

good effectiveness for user-defined types but struggles with handling parameterized types effectively.

CodeT5-ft's lower efficacy in handling user-defined types stems from its inability to grasp project-specific contexts, while the limited effectiveness of TypeGen for parameterized and unseen user-defined types can be attributed to its reliance on prompt engineering techniques to leverage the capability of LLMs, which often poses challenges in designing an optimal prompt template [18]. In comparison, our approach, TIGER, consistently achieves substantial inference accuracies for all type categories, especially (unseen) user-defined types (Column "Usr (Unseen)"), with enhancements of 16.5% (25.6%) and 11.2% (20.1%) compared to CodeT5-ft and TypeGen, respectively, in Top-5 Exact Match. These improvements are attributed to the comprehensive visible candidate pool and robust inference capacity integrated within the two-stage inference framework. Considering the prevalence of encountering (unseen) user-defined types in real-world projects, our approach proves to be more practical and applicable.

> **SUMMARY:** TIGER outperforms all six baseline approaches across the entire test set, achieving a Top-5 Exact Match metric of 93.1%. For fine-grained type categories, TIGER consistently achieves (almost) the highest accuracies, particularly for unseen user-defined types, with enhancements of 25.6% and 20.1% compared to the best two baselines, CodeT5-ft and TypeGen, respectively, in Top-5 Exact Match.

Approach	Top-1 EM (%)			Top-3 EM (%)			Top-5 EM (%)		
	Var	Arg	Ret	Var	Arg	Ret	Var	Arg	Ret
CodeT5-ft	89.8	67.8	76.3	94.3	79.4	84.4	95.3	82.7	85.8
TypeGen	82.0	73.9	69.1	88.6	81.6	77.6	90.0	83.5	79.4
TIGER	90.6	74.6	77.7	95.6	87.5	87.0	96.2	91.6	89.1

C. Robustness (RQ2)

As noted in prior research [3], type inference for certain variable categories such as function arguments and return values poses greater difficulty. Hence, we delve deeper into the robustness of TIGER and the best two baselines in RQ1, CodeT5-ft and TypeGen, across various variable categories. The evaluation results are outlined in Table III.

Overall, TIGER demonstrates superior robustness across different variable categories. Specifically, CodeT5-ft exhibits significant disparities (↓9.1% in Top-1 Exact Match) with TIGER for function arguments (Column "Arg"), while achieving comparable results for local variables (Column "Var") and return values (Column "Ret"). Conversely, TypeGen displays notable differences (↓9.5% and ↓11.1% in Top-1 Exact Match) with TIGER for local variables (Column "Var") and return values (Column "Ret"), while attaining comparable results for function arguments (Column "Arg"). In summary, our approach,

TIGER, consistently achieves consistently accurate results for all the three variable categories.

TIGER achieves consistently strong results across all three variable categories for two main reasons. First, compared to TypeGen, TIGER formulates the inference problem as a unified cloze-style fill-in-the-blank task, aligning perfectly with the PEP guidelines and the pre-training tasks of the base models. This allows for effective leveraging of the capacity of the pre-trained models. In contrast, TypeGen depends on distinct prompt templates tailored for each variable category, constraining the consistent utilization of LLMs across all categories. Second, compared to CodeT5-*ft*, which also utilizes the unified fill-in-the-blank task formulation, the contrastive training employed in TIGER's similarity model training enhances robustness in discriminating multiple candidates [23]. Although inferring types for function arguments is particularly challenging as their corresponding definition statements are often unavailable within the given function [3], TIGER can effectively discriminate candidate types for function arguments based on limited code context, such as the usage information of the arguments.

> **SUMMARY:** In comparison to CodeT5-*ft* and TypeGen, our approach, TIGER, showcases strong and more consistent inference accuracy across all three variable categories, particularly for function arguments and return types.

D. Efficiency (RQ3)

We evaluate the inference efficiency of TIGER alongside the best two baselines, CodeT5-*ft* and TypeGen, using a set of 50 test instances randomly sampled from the test set. All three approaches are executed on the same machine with a stable network connection. For CodeT5-*ft* and TIGER, a single NVIDIA V100-32G is utilized, and the test instances are processed one by one sequentially (*i.e.,* batch size of 1). For TypeGen, we adopt the default settings from their original implementation.

The average inference time of the three approaches for each instance is depicted in Figure 5. Notably, our approach, TIGER, and CodeT5-*ft* can process one instance within 1 second on average, while TypeGen requires significantly more time for inference (more than 10 seconds). This discrepancy can be attributed to the fact that TIGER and CodeT5-*ft* employ relatively smaller models compared to TypeGen (220M parameters vs. 175B parameters) and do not rely on complex program analysis such as code slicing. Furthermore, both TIGER and CodeT5-*ft* have the potential for further optimization in large-scale inference scenarios by leveraging parallel processing on GPUs (*e.g.,* increasing the batch size). However, TypeGen faces scalability challenges due to its reliance on OpenAI's online APIs for accurate inference, which significantly impacts the efficiency of TypeGen, given potential issues such as unstable network connections, API request frequency limits, and high financial costs.
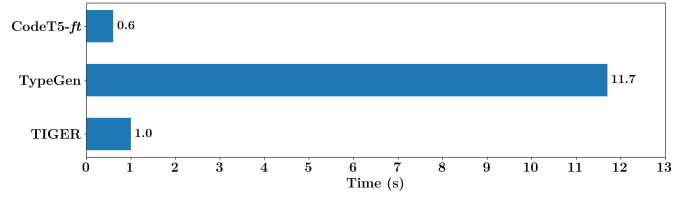

Fig. 5. Average inference time of CodeT5-*ft*, TypeGen, and TIGER.

TABLE IV
CONTRIBUTIONS OF THE TWO STAGES. THE BEST AND SECOND-BEST RESULTS FOR EACH CATEGORY ARE ARE HIGHLIGHTED.

Ablation	Top-5 Exact Match (%)						
	Ele	Par	Usr (Unseen)	Var	Arg	Ret	All
TIGER	97.7	89.4	94.1 (85.0)	96.2	91.6	89.1	94.6
only *generating*	97.7	89.4	80.8 (67.7)	95.3	82.7	85.8	91.9
only *ranking*	0.7*	5.3*	94.4 (86.5)	20.0	31.8	23.4	21.2

* The non-zero values arise from certain user-defined types sharing names with elementary types.

> **SUMMARY:** On average, our approach, TIGER, processes each instance in approximately 1 second, a performance comparable to CodeT5-*ft* and significantly more efficient than the LLM-based TypeGen. Moreover, TIGER's efficiency can be further enhanced for large-scale inference scenarios through GPU parallel processing.

E. Ablation Study (RQ4)

We conduct an ablation study to explore the contributions of the two stages of TIGER.

Table IV showcases the inference results of TIGER and its variants: **only *generating*** and **only *ranking***. When solely utilizing the generating stage, the inference achieves robust accuracy for most type and variable categories, except for (unseen) user-defined types (Column "Usr (Unseen)"). Conversely, when only employing the ranking stage, the inference demonstrates the highest accuracy for (unseen) user-defined types but performs poorly for the other type and variable categories. By synergizing the strengths of both stages, TIGER harnesses the creativity of the generation model and the discriminative ability of the similarity model, resulting in consistently practical inference accuracy across all type and variable categories.

> **SUMMARY:** The results of the ablation study confirm the significant contributions of both stages, *i.e., generating* and *ranking*, to robust and practical type inference.

F. Case Study

We have identified certain suboptimal cases, as illustrated in Figure 6, concerning user-defined types.

In Figure 6(a), TIGER's generation model produces complex parameterized types for a type-missed function, with their corresponding generative likelihoods (*i.e.,* lik). However, the expected type "promise" does not appear in the candidate list.

```python
def then(self, callback, on_error=None):
    if not isinstance(callback, Thenable):
        callback: <TYPE> = promise(callback, on_error=on_error)
    if self.cancelled:
        callback.cancel()
        return callback
    # ... omitted
```

Ground-truth: _promise_
(a user-defined type)

Candidates by Generation Model of TIGER

1: Thenable (lik=0.9369)
2: Callable[..., Any] (lik=0.7968)
3: Union[Callable[..., Any], Thenable] (lik=0.7390)
4: Union[Callable[..., Any], Callable[..., Any]] (lik=0.7385)
5: Union[Callable[..., Any], Thenable] (lik=0.7179)

Complex parameterized typed are generated

Candidates by Similarity Model of TIGER

1: promise (sim=0.8406)
2: Thenable (sim=0.6906)
...

"promise" is ranked as top-1 candidate by similarity

Final Results by TIGER

1: Thenable (score=0.7837)
2: promise (score=0.4603, lik=0.0003)
...

"promise" is ranked after "Thenable" in final results due to its lower likelihood

(a) Example 1: Predictions for Missing Type "`promise`"

```python
def OnBatchEnd(self, run_id, epoch_type, epoch_num, batch_num,
    timer, data: <TYPE>, results):
    if epoch_type in self.detailed_batch_epoch_types:
        details = log_database.BatchDetails.Create(
            data=data, results=results)
    else:
        details = None
    # ... omitted
```

Ground-truth: _Data_
(a user-defined type)

Candidates by Generation Model of TIGER

1: dict[str, typing.Any] (lik=0.8601)
2: list[dict[str, Any]] (lik=0.7325)
3: list[Any] (lik=0.7012)
4: str (lik=0.6626)
5: dict (lik=0.6343)

Diverse parameterized typed are generated

Candidates by Similarity Model of TIGER

1: Data (sim=0.8136)
2: Results (sim=0.7811)
...

"Data" is ranked as top-1 candidate by similarity

Final Results by TIGER

1: str (score=0.7837)
2: dict[str, Any] (score=0.4603)
3: dict (score=0.5369)
4: list[Any] (score=0.4602)
5: Data (score=0.4545, lik=0.0022)

"Data" is ranked 5-th in final results due to its low likelihood

(b) Example 2: Predictions for Missing Type "`Data`"

Fig. 6. Two Suboptimal Cases

Conversely, TIGER's similarity model ranks the ground-truth type as the top-1 candidate with a high similarity score (*i.e.*, sim). When the two models are combined, TIGER eventually returns "`promise`" as the 2nd result, just after "`Thenable`". Similarly, in Figure 6(b), the expected "`Data`" type is absent from the candidates generated by the generation model but is ranked 1st by the similarity model. However, in the final results, it only appears in 5th place.

These suboptimal outcomes can be attributed to the fact that some user-defined types, which are not generated as candidates by the generation model, tend to have lower likelihoods compared to model-generated candidates. As a result, even with high similarities from the similarity model, their final scores, as calculated by Eq. 6, remain lower. While we have attempted to adjust the weights of the two measures in Eq. 6, finding an optimal balance has proven challenging. In the future, we plan to explore more effective methods for combining these two measures, such as incorporating likelihood into the contrastive loss.

VI. THREATS TO VALIDITY

The primary internal threat to validity arises from potential data errors within the training and test sets. To mitigate this threat, we employ the widely-used ManyTypes4Py dataset for training and evaluating Python type inference approaches, ensuring consistency with common practices in the field. Another potential internal threat to validity pertains to the settings of model training hyperparameters. To address this concern, we adhere to standard values for training hyperparameters, including learning rate and epoch numbers.

Regarding the external threat, although we have consdiered a varity of baselines, the absence of a comparison with DLInfer [13] in our evaluation also poses a external threat to validity. The lack of a publicly available implementation and the complexity of reimplementation make it challenging to include DLInfer for a fair comparison. Instead, we compared with the more recent state-of-the-art approach, TypeGen, which also leverages code slicing similar to DLInfer.

VII. CONCLUSION AND FUTURE WORK

This paper introduces TIGER, a novel Python type inference approach employing a *GTR* framework. TIGER trains a generation model with a generative span masking objective and a similarity model with a contrastive training objective. Both models contribute to the *GTR* inference process, generating and ranking candidate types alongside user-defined types based on generative likelihood and contextual similarity. Evaluation on the ManyTypes4Py dataset reveals that TIGER performs effectively across various type categories, notably demonstrating significant improvements in (unseen) user-defined types. Furthermore, the evaluation results underscore the resilience and efficiency of TIGER, underscoring the significance of the employed two-stage approach.

In the future, we plan to extend the applicability of TIGER to other dynamic programming languages such as JavaScript and explore the further integration with static analysis for type validation. Additionally, we aim to adapt TIGER to address other practical type inference challenges, including those related to partial Python functions.

DATA AVAILABILITY

Our replication package is publically available at [63].

REFERENCES

[1] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 91–105.

[2] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Practical deep similarity learning-based type inference for python," in *Proceedings of 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2241–2252.

[3] Y. Peng, C. Wang, W. Wang, C. Gao, and M. R. Lyu, "Generative type inference for python," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 988–999.

[4] (2023) Pep 484 – type hints. [Online]. Available: https://peps.python.org/pep-0484/

[5] (2023) Pep 526 – syntax for variable annotations. [Online]. Available: https://peps.python.org/pep-0526/

[6] (2023) Mypy: Static typing for python. [Online]. Available: https://github.com/python/mypy/

[7] (2023) Pyre - a performant type-checker for python 3. [Online]. Available: https://pyre-check.org/

[8] (2023) Pyright - static type checker for python. [Online]. Available: https://github.com/microsoft/pyright

[9] (2023) pytype - a static type analyzer for python code. [Online]. Available: https://github.com/google/pytype

[10] J. Ore, S. G. Elbaum, C. Detweiler, and L. Karkazis, "Assessing the type annotation burden," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 190–201.

[11] K. Jesse, P. T. Devanbu, and T. Ahmed, "Learning type annotation: is big data enough?" in *Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 1483–1486.

[12] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: neural type prediction with search-based validation," in *Proceedings of 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 209–220.

[13] Y. Yan, Y. Feng, H. Fan, and B. Xu, "Dlinfer: Deep learning with static slicing for python type inference," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2009–2021.

[14] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 8696–8708.

[15] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 7212–7225.

[16] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *Proceedings of The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[17] Y. Peng, C. Gao, Z. Li, B. Gao, D. Lo, Q. Zhang, and M. R. Lyu, "Static inference meets deep learning: A hybrid type inference approach for python," in *Proceedings of 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2019–2030.

[18] J. D. Zamfirescu-Pereira, R. Y. Wong, B. Hartmann, and Q. Yang, "Why johnny can't prompt: How non-ai experts try (and fail) to design LLM prompts," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, CHI 2023, Hamburg, Germany, April 23-28, 2023*, A. Schmidt, K. Väänänen, T. Goyal, P. O. Kristensson, A. Peters, S. Mueller, J. R. Williamson, and M. L. Wilson, Eds. ACM, 2023, pp. 437:1–437:21.

[19] (2023) Openai api reference. [Online]. Available: https://platform.openai.com/docs/api-reference/

[20] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," in *Proceedings of The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, ser. Findings of ACL, T. Cohn, Y. He, and Y. Liu, Eds., vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.

[22] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 1597–1607.

[23] A. van den Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *CoRR*, vol. abs/1807.03748, 2018.

[24] A. van den Oord, O. Vinyals, and K. Kavukcuoglu, "Neural discrete representation learning," in *Proceedings of Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 6306–6315.

[25] K. He, H. Fan, Y. Wu, S. Xie, and R. B. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 2020, pp. 9726–9735.

[26] C. An, J. Feng, K. Lv, L. Kong, X. Qiu, and X. Huang, "Cont: Contrastive neural text generation," in *NeurIPS*, 2022.

[27] A. M. Mir, E. Latoskinas, and G. Gousios, "Manytypes4py: A benchmark python dataset for machine learning-based type inference," in *Proceedings of 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 585–589.

[28] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, ser. Lecture Notes in Computer Science, A. P. Black, Ed., vol. 3586. Springer, 2005, pp. 428–452.

[29] S. Chen and M. Erwig, "Principal type inference for gadts," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 416–428.

[30] M. Emmi and C. Enea, "Symbolic abstract data type inference," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 513–525.

[31] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript," in *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, ser. Lecture Notes in Computer Science, J. Palsberg and Z. Su, Eds., vol. 5673. Springer, 2009, pp. 238–255.

[32] Z. Pavlinovic, Y. Su, and T. Wies, "Data flow refinement type inference," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–31, 2021.

[33] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 607–618.

[34] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," in *Proceedings of 8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[35] J. Wei, G. Durrett, and I. Dillig, "Typet5: Seq2seq type inference using static analysis," in *Proceedings of The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[36] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, L. Li, and Z. Sui, "A survey for in-context learning," *CoRR*, vol. abs/2301.00234, 2023.

[37] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture? - A structural analysis of pre-trained language models for source code," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2377–2388.

[38] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 1332–1336.

[39] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun, Y. Liu, and X. Peng, "Teaching code llms to use autocompletion tools in repository-level code generation," *CoRR*, vol. abs/2401.06391, 2024.

[40] Y. Wang, Y. Wang, D. Guo, J. Chen, R. Zhang, Y. Ma, and Z. Zheng, "Rlcoder: Reinforcement learning for repository-level code completion," *arXiv preprint arXiv:2407.19487*, 2024.

[41] J. Liu, Y. Chen, M. Liu, X. Peng, and Y. Lou, "Stall+: Boosting llm-based repository-level code completion with static analysis," *arXiv preprint arXiv:2406.10018*, 2024.

[42] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[43] E. Shi, Y. Wang, W. Gu, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cocosoda: Effective contrastive learning for code search," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2198–2210.

[44] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019.

[45] C. Wang, J. Liu, X. Peng, Y. Liu, and Y. Lou, "Boosting static resource leak detection via llm-based resource-oriented intention inference," *CoRR*, vol. abs/2311.04448, 2023.

[46] J. Zhang, C. Wang, A. Li, W. Sun, C. Zhang, W. Ma, and Y. Liu, "An empirical study of automated vulnerability localization with large language models," *arXiv preprint arXiv:2404.00287*, 2024.

[47] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2339–2356.

[48] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1162–1174.

[49] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, A. Roychoudhury, C. Cadar, and M. Kim, Eds. ACM, 2022, pp. 382–394.

[50] C. Wang, Y. Lou, J. Liu, and X. Peng, "Generating variable explanations via zero-shot prompt learning," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 748–760.

[51] 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: Pretraining code representations with data flow," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[52] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 1171–1179.

[53] A. Fan, M. Lewis, and Y. N. Dauphin, "Hierarchical neural story generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, I. Gurevych and Y. Miyao, Eds. Association for Computational Linguistics, 2018, pp. 889–898.

[54] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[55] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020.

[56] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[57] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-based environment dependency inference for python programs," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1245–1256.

[58] Y. Su, T. Lan, Y. Wang, D. Yogatama, L. Kong, and N. Collier, "A contrastive framework for neural text generation," in *NeurIPS*, 2022.

[59] K. Arora, L. E. Asri, H. Bahuleyan, and J. C. K. Cheung, "Why exposure bias matters: An imitation learning perspective of error accumulation in language generation," in *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, S. Muresan, P. Nakov, and A. Villavicencio, Eds. Association for Computational Linguistics, 2022, pp. 700–710.

[60] (2023) Transformers. [Online]. Available: https://github.com/huggingface/transformers

[61] (2023) Hugging face – codet5-base. [Online]. Available: https://huggingface.co/Salesforce/codet5-base

[62] (2023) Hugging face – codet5-base. [Online]. Available: https://github.com/JohnnyPeng18/TypeGen/releases/tag/data

[63] (2023) Replication package. [Online]. Available: https://anonymous.4open.science/r/TypeInfer-Replication-FC77/README.md