TypeCare: Boosting Python Type Inference Models via Context-Aware Re-Ranking and Augmentation

Wonseok Oh
Korea University
Seoul, Republic of Korea
wonseok_oh@korea.ac.kr

Hakjoo Oh*
Korea University
Seoul, Republic of Korea
hakjoo_oh@korea.ac.kr

Abstract

Type annotations improve Python code quality by enabling better readability, static analysis, and developer productivity. However, manually annotating existing code is labor-intensive and error-prone. While recent learning-based models have advanced automatic type inference, they struggle with rare or complex types that are underrepresented in training data.

We present TypeCare, a model-agnostic post-processing technique that refines the outputs of existing type inference models using code context, without requiring retraining or fine-tuning. TypeCare combines two key components: (1) *Re-Ranking*, which prioritizes semantically and syntactically relevant types, and (2) *Augmentation*, which generates additional contextually plausible candidates. Applied to three state-of-the-art type inference models—TypeT5, Tiger, and TypeGen—TypeCare consistently improves top-1 accuracy, achieving up to 40.1% gains on complex types that existing models often fail to predict correctly.

```
1  class DocStore:
2    def read(self, doc_id):
3      if condition:
4        return doc_id
5      return -1
6
7  class Project:
8    def __init__(self):
9      self.doc = DocStore()
10
11   def get(self, project_id: <FILL_IN>):
12     doc = self.doc.read(project_id)
13     return doc + 1
```

Model Output

```
1. str                  ✗
2. int                  ✓
3. Union[int, str]      ✗
```

Figure 1: An example where the goal is to predict the type of `project_id` at line 11. The correct type is `int`, but Tiger [28] predicts `str` as the top-1 candidate.

CCS Concepts

• **Software and its engineering → Software testing and debugging.**

ACM Reference Format:
Wonseok Oh and Hakjoo Oh. 2018. TypeCare: Boosting Python Type Inference Models via Context-Aware Re-Ranking and Augmentation. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Python is one of the most widely used programming languages in the world, and its popularity is growing rapidly [6]. While its dynamic typing system allows developers to write code quickly, it often impedes maintenance, debugging, and analyzing codebases. To alleviate these issues, the Python community has adopted gradual typing [27] through type annotations, enabling developers to progressively add type information to their codebases. The type annotations unlock a variety of benefits, such as improved code readability, better static analysis, and enhanced developer productivity. However, adding type annotations to existing codebases is a time-consuming and error-prone task [19]. This challenge has motivated a growing body of research aimed at automating the type annotation process [4, 5, 20, 21, 28–30].

Existing Approaches. To automatically annotate types, prior research has evolved along two main directions. Early works relied on rule-based models that aimed to infer types by carefully designing heuristic rules [7–9, 14, 15, 23]. However, these methods struggled to achieve broad coverage due to the dynamic and diverse nature of Python code. Recent research has shifted towards learning-based models that leverage machine learning techniques to predict types based on the context of the code [1, 11, 17, 20–22, 28, 30, 32].

Early works to learning-based type annotation focused on deep similarity models calculating probabilities of types in a program [1, 17, 20]. Recent works have further advanced this field by leveraging language models [21, 28, 30]. TypeT5 [30] enriches the model input, i.e., code snippets used for type prediction, with caller-callee relationships using sequence-to-sequence learning on top of a fine-tuned CodeT5 model. TypeGen [21] introduces a Chain-of-Thought prompting technique that guides the LLM's reasoning process for type inference based on a type dependency graph. Tiger [28] integrates a fine-tuned language model with a deep similarity model to reason about types beyond the input context provided to the model.

Limitation. Despite their advancements, these state-of-the-art models share a common limitation: they struggle to make accurate predictions for rare or unseen types that are underrepresented in

*Corresponding author.

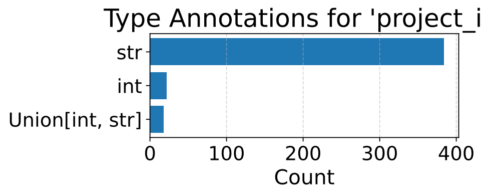

Figure 2: The number of annotations for `project_id` in the dataset used to train Tiger [28].

the training or fine-tuning data. While they excel at predicting statistically common types like `int` or `str` (with an average accuracy of ~86%), their performance drops significantly for data-dependent parameterized types such as `List[Dict[str, int]]` (~30%) or program-specific user-defined types (~54%). Furthermore, this difficulty extends even to simple types when they appear in statistically infrequent usage patterns from the training data.

The example in Figure 1 illustrates this problem. The correct type for `project_id` at line 11 is `int` since it is passed to the `DocStore.read` method at line 12, which returns an `int` value. However, Tiger [28], a state-of-the-art model, incorrectly infers the type of `project_id` at line 11 as `str`. To understand this failure, we examined how `project_id` is annotated in the training dataset used for Tiger. As shown in Figure 2, we found that `project_id` is annotated as `str` in approximately 90% of the cases, suggesting that the model is biased towards predicting `str` for this variable.

This Work. To address this limitation, we present TypeCare, a new technique for improving the performance of type inference models for Python. TypeCare is a model-agnostic, post-processing method that refines the outputs of existing models using code context information, without requiring any retraining or fine-tuning. We propose two key components for refinement: (1) **Re-Ranking**, which leverages semantic and syntactic information to prioritize more plausible types, and (2) **Augmentation**, which generates new candidate types that are likely to be correct based on code context. First, TypeCare re-ranks the model output, i.e., a ranked list of candidate types, by leveraging type validity and code usage patterns observed across the project, promoting correct types to the top. Second, to maximize the effectiveness of the re-ranking step, TypeCare augments uncertain predictions with additional contextually plausible type candidates.

We evaluated TypeCare by applying it to three state-of-the-art type inference models: TypeT5 [30], Tiger [28], and TypeGen [21]. The results show that TypeCare consistently improves top-1 accuracy across all models and settings. In single-variable prediction tasks, TypeCare increased the top-1 accuracy of TypeT5, Tiger, and TypeGen from 71.4%, 67.8%, and 65.4% to 81.1%, 75.8%, and 73.6%, respectively. The improvements were especially significant for complex types, such as parameterized and user-defined types, where TypeCare improved the baseline models by up to 40.1%. Moreover, in the challenging task of function signature prediction, TypeCare increased the top-1 accuracy of TypeT5 from 59.1% to 74.5%.

Contributions. In summary, our contributions are as follows:

```
1  class A:
2    text: Union[str, bytes]
3
4    def ident(self, i_value: str):
5      validate(i_value[1:])
6      msg = self.text + i_value
7
8    def data(self, d_value: bytes):
9      d = d_value.decode('utf-8')
10     return d
11
12   def scope(self, s_value: <FILL_IN>):
13     validate(s_value[1:])
14     msg = self.text + s_value
```

Model Output

```
1. s_value: bytes ✗
2. s_value: int   ✗
3. s_value: str   ✓
```

Figure 3: A simplified example where the goal is to predict the type of `s_value` at line 12. The correct type is `str`, but the recent model, Tiger [28], predicts `bytes` as the top-1 candidate.

- We propose a novel refinement approach, TypeCare, that enhances type prediction models by re-ranking and augmenting model outputs using code context information. By requiring no model retraining, TypeCare offers a lightweight and practical solution for improving Python type inference models.
- We prove the effectiveness of our approach in real-world settings. We show that TypeCare substantially improves top-1 accuracy of existing models, with up to a 40.1% gain on complex, parameterized and user-defined types. We also demonstrate the robustness of our approach through diverse evaluation settings and ablation studies.

2 Overview

In this section, we illustrate the high-level ideas of our approach using examples.

Type Prediction Problem. In this paper, we consider the cloze-style type prediction problem for annotating Python function signatures, a problem setting that has been widely adopted in recent studies [4, 5, 28–30]. Listing 1 illustrates an example, where the `<FILL_IN>` placeholders indicate the positions in the function signature to be predicted. Given such input, the goal of a type inference model is to infer the appropriate types for `x`, `y`, and the return value.

Listing 1: An example of the type prediction problem

```
def add(x: <FILL_IN>, y: <FILL_IN>) -> <FILL_IN>:
    return x + y
```

However, state-of-the-art type inference models [21, 28, 30] often fail to predict complex or project-specific types that occur infrequently in the training data. To overcome this limitation, we introduce TypeCare, a post-processing technique that leverages code context to refine model predictions. Our approach consists of two main components: (1) Re-ranking the model's outputs to prioritize the correct type, and (2) Augmenting the model outputs with additional, contextually plausible types.

2.1 Code Context-based Re-Ranking

The goal of the re-ranking step is to improve the model outputs by promoting the correct types based on the surrounding code context. To this end, we propose a scoring method that combines semantic signals derived from static type checkers and syntactic signals obtained from code usage patterns.

2.1.1 Re-ranking with Type Validity. Re-ranking based on type validity can significantly enhance type prediction performance. When type-checking the example in Figure 3, annotating the `project_id` parameter with `str` triggers a type error alarm at line 13 because the variable `doc` can be `str` type, while `int` does not raise any type error alarm, which can be a signal that `str` is a more plausible type than `int`. Prior approaches have also utilized type validity by filtering out candidate types that trigger type errors [20, 22]. However, we observed that this conventional filtering strategy can inadvertently degrade the performance of type inference models.

For example, consider Figure 3, where the goal is to infer the type of `s_value` at line 12, which should correctly be annotated as `str`. Unfortunately, if we insert `str` into the placeholder `<FILL_IN>` and perform static type analysis, a type error alarm is raised at line 14 because `self.text` can be either `str` or `bytes` (line 2), and concatenating a `bytes` object with a `str` object is invalid. Consequently, according to the existing filtering strategy, the correct type `str` would incorrectly be excluded from consideration.

Relatively Better Type Validity. This observation led us to conclude that rather than simply checking whether a candidate type raises an alarm, we should assess *how favorable the type is relative to other candidates.* Consider the `int` type in Figure 3. If `int` is inserted into the `<FILL_IN>` placeholder, an additional type error is raised at line 13 as `int` is not subscriptable (`s_value[1:]`). In contrast, inserting `str` does not raise this particular error. Thus, among the model's outputs, `str` is relatively better than `int`, allowing us to re-rank `str` higher.

2.1.2 Re-ranking with Code Usage Pattern. However, we found that type correctness alone is insufficient to effectively re-rank the model outputs. For instance, when the `bytes` type is assigned to the `s_value` parameter in Figure 3, it triggers the same type error as the `str` type. As a result, the `bytes` type is not penalized relative to the `str` type, even though only the latter is correct.

To address this limitation, we incorporate syntactic information to further enhance the effectiveness of refinement. The key insight is that variables with similar code usage patterns are more likely to share the same type. Let us consider the `ident` method at line 4 and the `data` method at line 8 in Figure 3. The parameter `i_value` in `ident` is used in a manner similar to the parameter `i_value` in `ident` (e.g., `i_value[1:]` and `self.text + i_value`). In contrast, the parameter `d_value` in `data` exhibits a distinct usage pattern (`d_value.decode('utf-8')`). Based on this observation, we infer that the parameter `s_value` is more likely to have the same type as `i_value`. Accordingly, we re-rank the `str` type as the top-1 candidate type for the `s_value` parameter.

Type Similarity Model. To this end, we propose a type similarity model to estimate the likelihood that two variables have the same type based on their code usage patterns. Specifically, the model estimates how closely the target variable resembles other variables in terms of syntactic features. For example, in Figure 3, we compute the similarity scores for the pairs `(s_value, i_value)` and `(s_value, d_value)`. If the score for `(s_value, i_value)` is higher than that for `(s_value, d_value)`, we conclude that `s_value` is more likely to have the same type as `i_value`. In this case, it suggests the type of `s_value` is more likely to be `str` rather than `bytes`.

To achieve this, we train a model to predict a similarity score that indicates the likelihood that two variables have the same type, based on their code usage patterns. We categorize the syntactic usage pattern into four representative types: (1) class similarity, (2) function similarity, (3) variable similarity, and (4) usage similarity. From these categories, we extract 24 features and use them to train the scoring model (see Table 1 for the full list of features). Using these syntactic features, the model correctly predicts that `s_value` is more likely to have type `str` rather than `bytes`, as its AST-based usage pattern is more similar to that of `i_value` than `d_value`.

```
 1 def make_identifier(
 2     session: <FILL_IN>,
 3     name: <FILL_IN>
 4 ):
 5   with session.cursor() as c:
 6     c.execute(...)
 7
 8 def fetch_identifier(
 9     d_session: DatabaseSession
10 ):
11   with session.cursor() as c:
12     c.execute(...)
```

Model Output

```
1. session: Session ✗
   name: str      ✓

2. session: dict   ✗
   name: str      ✓

3. session: list   ✗
   name: str      ✓
```

Figure 4: An example where the goal is to predict the types of `session` (line 2) and `name` (line 3). The correct types are `DatabaseSession` and `str`, but the recent model, TypeT5 [30], fails to predict the correct type for variable `session`.

2.2 Code Context-based Augmentation

In the previous section, we introduced a re-ranking approach to score the relative quality of the model outputs based on code context. This naturally raises the question: can the effectiveness of re-ranking be further amplified by improving the quality of the candidate types?

For instance, Figure 4 presents a case where the model should predict the correct types of both `session` and `name` at lines 2 and 3, which are `DatabaseSession` and `str`, respectively. While the model (TypeT5) correctly infers the type of `name`, it fails to predict the correct type for `session`. This is largely due to `DatabaseSession` being a user-defined, project-specific type, which poses a challenge for the model to predict. However, if the candidate answers include the correct pair (`session: DatabaseSession`, `name: str`), our re-ranking algorithm can prioritize the correct type as the top-1 prediction. Other candidates are penalized by the type validity check because they do not support the `cursor` attribute accessed at line 5, whereas the `DatabaseSession` type does not raise any alarms. This observation motivated us to explore how to augment the candidate set with such likely types to further improve model performance.

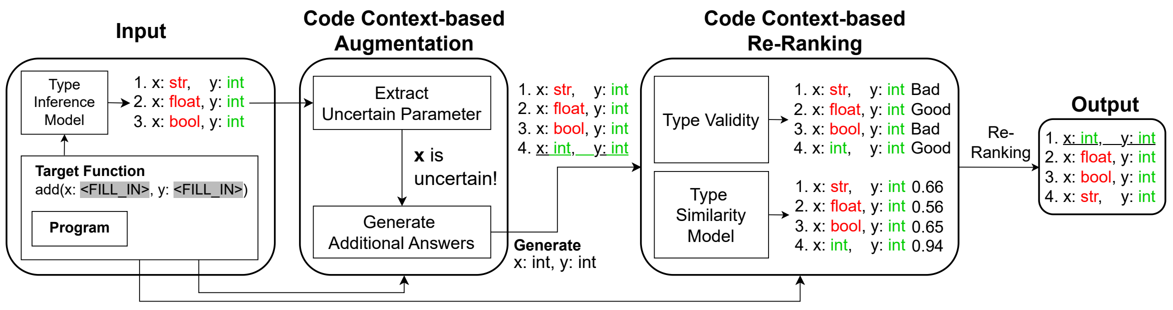

Figure 5: The overall workflow of our approach.

Type Augmentation for Uncertain Variables. Interestingly, we found that the model tends to produce a more diverse set of type candidates when it is uncertain about a variable's type. For instance, in Figure 4, the model outputs three candidates for `session`: `Session`, `dict`, and `list`. On the other hand, the model outputs a single candidate for `name`: `str`. This suggests that the model is confident in its prediction for `name`, but uncertain about the type of `session`.

Based on this observation, we propose measuring the uncertainty in the model's predictions and introducing new type candidates when the uncertainty is high. In Figure 4, the model exhibits high uncertainty for `session`, as it outputs multiple candidates across all predictions. In contrast, the uncertainty of `name` is low because the model consistently predicts a single candidate.

To generate new type candidates for variables with high uncertainty, we leverage code usage patterns. For instance, to identify candidate types for `session`, we search the codebase for other variables and their associated types. In Figure 4, we identify the parameter `d_session` at line 9, which is annotated with the type `DatabaseSession`. We then compute a similarity score between `session` and `d_session` using the trained model in Section 2.1.2. If the score is sufficiently high, we treat the type of `d_session` as a candidate type for `session`. This leads to the synthesis of a new candidate (`session: DatabaseSession, name: str`), which is added to the candidate set. Finally, we apply our re-ranking procedure to evaluate whether this augmented candidate should be prioritized. As a result, the new candidate (`session: DatabaseSession, name: str`) is re-ranked as the top-1 prediction.

2.3 Overall Workflow

Figure 5 shows the overall workflow of our approach. The input to TypeCare is a Python function signature containing `<FILL_IN>` placeholders, along with candidate answers produced by a type prediction model. The pipeline consists of the following steps: (1) identify uncertain parameters and augment the candidate set with additional answers (Section 2.2), (2) compute type validity and syntactic similarity scores for all candidate answers (Section 2.1), (3) re-rank the candidate answers based on these scores. Finally, TypeCare returns the refined, re-ranked type predictions.

3 Algorithm

In this section, we describe the algorithm of TypeCare in detail.

Preliminaries. We denote types as $t \in \mathit{Type}$ and parameters as $p \in \mathit{Var}$. Next, we represent a sequence of types as $\vec{t} = (t_1, t_2, \ldots, t_n)$ and a sequence of parameters as $\vec{p} = (p_1, p_2, \ldots, p_n)$, where $\vec{t}(n)$ and $\vec{p}(n)$ denote the n-th type and parameter in the sequence, respectively. For simplicity, we assume that the parameter sequence includes the return variable *ret* as the final element. For example, given the function signature `foo(x: int, y: str) -> float`, we define $\vec{p} = (x, y, \mathit{ret})$ and $\vec{t} = (\texttt{int}, \texttt{str}, \texttt{float})$.

Problem Definition. We define a program $P \in \mathit{Program}$ as a sequence of function declarations, denoted by F. For simplicity, we represent each function declaration as a tuple $(f, \vec{p})$, where f is the function name and $\vec{p}$ is the sequence of its parameters. We denote by $f^\#$ the target function to be annotated, and by $\vec{p}^\#$ its parameter sequence. Given a target signature $\theta^\# = (f^\#, \vec{p}^\#) \in \mathit{Signature}$, our goal is to infer its type signature $\vec{t} \in \mathit{Type}^*$, where $\vec{t}$ is a sequence of types such that each $\vec{t}(i)$ corresponds to the type of $\vec{p}^\#(i)$.

We denote a type inference model as a function

$$\mathcal{M} : \mathit{Program} \times \mathit{AnnotMap} \times \mathit{Signature} \times \mathbb{N} \longrightarrow \mathit{ModelOutput}$$

A type-annotation map, $\Gamma \in \mathit{AnnotMap} = \mathit{Func} \times \mathit{Var} \longrightarrow \mathit{Type}$, denotes the type annotations in program P, where $\Gamma(f, p)$ denotes the annotated type of parameter p in function f. Given a program P, an annotation map Γ, a target signature $\theta^\#$, and an integer k, the model $\mathcal{M}$ predicts k candidate type signatures for $\theta^\#$, that is,

$$\mathcal{M}(P, \Gamma, \theta^\#, k) = \Delta,$$

where $\Delta \in \mathbb{N} \longrightarrow \mathit{Type}^*$ and $\Delta(n)$ denotes the n-th candidate type sequence.

Example 3.1. Let us consider the function that adds two numbers:

```
def add(x: <FILL_IN>, y: <FILL_IN>) -> <FILL_IN>:
    return x + y
```

Our goal is to predict the type signature of the function `add`. The target signature is defined as $\theta^\# = (\mathit{add}, (x, y, \mathit{ret}))$. Given the enclosing program P and annotation map Γ, the model $\mathcal{M}(P, \Gamma, \theta^\#, k)$ predicts k candidate type signatures for $\theta^\#$, denoted as Δ. As an example, when $k = 5$, the model output Δ is as follows:

$$\Delta = \begin{bmatrix} 1 & \mapsto & [\texttt{int}, \texttt{int}, \texttt{int}] \\ 2 & \mapsto & [\texttt{float}, \texttt{int}, \texttt{float}] \\ 3 & \mapsto & [\texttt{int}, \texttt{float}, \texttt{float}] \\ 4 & \mapsto & [\texttt{float}, \texttt{float}, \texttt{float}] \\ 5 & \mapsto & [\texttt{str}, \texttt{str}, \texttt{str}] \end{bmatrix}$$

Goal. Our goal is to refine the model outputs $\Delta = \mathcal{M}(P, \Gamma, \theta^{\#}, k)$ for target signature $\theta^{\#}$, which we denote using function *Refine*:

$$Refine(P, \Gamma, \Delta, \theta^{\#})$$

where we set the hyperparameter k to 10 in our experiments. The refinement function *Refine* consists of two components: re-ranking and augmentation:

$$Refine(P, \Gamma, \Delta, \theta^{\#}) = ReRank(P, \Gamma, \Delta^{+}, \theta^{\#}) \quad \text{where } \Delta^{+} = Augment(P, \Gamma, \Delta, \theta^{\#}, m)$$

Here, the hyperparameter m specifies the number of additional candidates to be augmented, and we set $m = 3$ in our experiments. We describe the details of re-ranking (*ReRank*) and augmentation (*Augment*) components in the following sections.

3.1 Re-Ranking

We re-rank the candidate types predicted by the type inference model using a scoring function that considers type validity and code usage patterns. We define the re-ranking function *ReRank* as follows:

$$ReRank(P, \Gamma, \Delta, \theta^{\#}) = Top_k(\text{Sort}(S))$$

where

$$S = \{(\Delta(i), Score(P, \Gamma, \theta^{\#}, \Delta(i), i)) \mid 1 \le i \le |\Delta|\}.$$

Here, Sort(S) returns the sequence sorted by score in descending order, and Top_k selects the top-k candidates from that sequence. The sorting function Sort is defined as:

$$\text{Sort}(S) = [x_1, x_2, \ldots, x_n] \quad \text{where } S = \{(x_1, s_1), \ldots, (x_n, s_n)\} \text{ and } s_i \ge s_j \text{ if } i < j$$

If the score s_i is a tuple, elements are sorted in lexicographic order.

The function *Score* returns a tuple of scores for a candidate type signature $\vec{t}$, incorporating type validity, code usage patterns, and the original model rank i:

$$Score(P, \Gamma, \theta^{\#}, \vec{t}, i) = (ScoreTyp(P, \Gamma, \theta^{\#}, \vec{t}), ScorePat(P, \Gamma, \theta^{\#}, \vec{t}), i)$$

where *ScoreTyp* estimates the relative quality of $\vec{t}$ using static type analysis, while *ScorePat* measures the similarity of $\vec{t}$ to usage patterns observed in the code.

3.1.1 *Type Validity.*

We utilize static type analysis to assess the validity of candidate types. As discussed in Section 2.1.1, instead of simply filtering candidates based on whether they raise alarms, we evaluate the relative quality of each type. To this end, we count the number of new alarms introduced by each candidate type compared to the original program.

We assume a static type checker $Static : Program \times AnnotMap \to Alarms$ that reports a set of type errors (alarms) for a given program and its annotations. To measure the impact of a candidate type sequence $\vec{t}$ for the target signature $(f^{\#}, \vec{p}^{\#})$, we compute the number of new alarms introduced by updating the original annotations as follows:

$$ScoreTyp(P, \Gamma, (f^{\#}, \vec{p}^{\#}), \vec{t}) = |Static(P, \Gamma') \setminus Static(P, \Gamma)|$$

where

$$\Gamma' = \Gamma \oplus \{(f^{\#}, \vec{p}^{\#}(i)) \mapsto \vec{t}(i) \mid 1 \le i \le |\vec{p}^{\#}|\}$$

and $\oplus$ denotes an update to the annotation map. In other words, we construct Γ' by assigning the candidate type $\vec{t}(i)$ to the i-th parameter in the target signature, and then apply static type analysis to the updated annotation.

3.1.2 *Code Usage Pattern.*

Our objective is to compute a score that measures how well a candidate signature type $\vec{t}$ fits the target signature $\theta^{\#}$, based on code usage patterns in the program P. To do so, we evaluate the suitability of each type in $\vec{t}$ for its corresponding parameter and sum these scores to produce the final score:

$$ScorePat(P, \Gamma, (f^{\#}, \vec{p}^{\#}), \vec{t}) = \sum_{i=1}^{|\vec{p}^{\#}|} SimScore(P, \Gamma, f^{\#}, \vec{p}^{\#}(i), \vec{t}(i))$$

The function *SimScore* computes the likelihood that the parameter $\vec{p}^{\#}(i)$ has the type $\vec{t}(i)$ by leveraging a type similarity model $\mathcal{R} : (Func \times Var) \times (Func \times Var) \to \mathbb{R}$ defined as follows:

$$SimScore(P, \Gamma, f^{\#}, p^{\#}, t) = \max_{(f,p) \in Cand(P, \Gamma, f^{\#}, t)} \mathcal{R}(\Phi) \quad \text{where } \Phi = FeatureVec((f^{\#}, p^{\#}), (f, p))$$

We collect other parameters in the program P that are annotated with the same type t, and return the maximum similarity score among them. We select such parameters from the candidate set $Cand(P, \Gamma, f^{\#}, t)$, which contain all parameters in the program P annotated with type t as follows:

$$Cand(P, \Gamma, f^{\#}, t) = \{(f, p) \mid (f, \vec{p}) \in P, p \in \vec{p}, f \ne f^{\#}, t = \Gamma(f, p)\}$$

In short, we compare the target parameter $p^{\#}$ with other parameters of the same type, take the maximum similarity score for each, and use the sum of such maximum scores as the final score. In the following, we explain the details of how the model $\mathcal{R}$ and feature vector Φ are designed.

Features. The function $FeatureVec : (Func \times Var) \times (Func \times Var) \to FeatureVector$ returns a feature vector Φ that captures the syntactic similarity between two parameters. We designed 24 features that capture code usage patterns focused on simplicity and generality. Table 1 lists the features used in our model. Rather than describing each feature in detail, we highlight the most important ones below.

- **Name Similarity** (#4-6, #9-11, #16-18): We compute the name similarity using multiple string metrics: Levenshtein and Jaro-Winkler distance [10, 13, 31]. We use the Levenshtein distance [13] to measure overall character-level similarity, and the (Reversed) Jaro-Winkler [10, 31] similarity to capture prefix or suffix-level similarity. These metrics are complementary and together provide a robust measure of name similarity.
- **AST Similarity** (#20-24): We define the usage context of a parameter as the set of AST nodes in which the parameter is used. We collect all expressions and statements that use the given parameter and compute the similarity between their usage contexts. Cosine similarity over the sets of AST nodes is used as the primary metric. Additionally, we compute a usage context ratio, which is the proportion of each parameter's usage context relative to the combined context of both parameters.

Table 1: Features for the type similarity model.

Kind	#	Description
File	#1	If f_1 and f_2 are defined in the same file
Class	#2	If f_1 is defined within a class
	#3	If f_2 is defined within a class
	#4	Levenshtein distance of class names
	#5	Jaro-Winkler distance of class names
	#6	Reversed Jaro-Winkler distance of class names
Func	#7	If the name of f_1 contains the name of f_2
	#8	If the name of f_2 contains the name of f_1
	#9	Levenshtein distance of f_1 and f_2
	#10	Jaro-Winkler distance of f_1 and f_2
	#11	Reversed Jaro-Winkler distance of f_1 and f_2
	#12	The similarity of decorators between f_1 and f_2
	#13	The number of parameters in f_1
	#14	The number of parameters in f_2
	#15	If parameters of f_1 and f_2 are the same
Param	#16	Levenshtein distance of p_1 and p_2
	#17	Jaro-Winkler distance of p_1 and p_2
	#18	Reversed Jaro-Winkler distance of p_1 and p_2
	#19	If one name is the plural form of the other
	#20	The similarity between their usage contexts
	#21	The number of usage contexts for p_1
	#22	The number of usage contexts for p_2
	#23	The usage context ratio of p_1
	#24	The usage context ratio of p_2

We define the feature vector Φ as $(\phi_1, \phi_2, \ldots, \phi_{24})$ where feature ϕ_i denotes the value of the i-th feature in Table 1.

Training the Model $\mathcal{R}$ with Selected Data. Given a training program P_t, we construct a dataset to train the model $\mathcal{R}$. Each data point is a tuple (Φ, l) where $l \in \{0, 1\}$ indicates whether the two parameters represented by the feature vector Φ share the same type in the training program:

$$\bigcup_{(f_1,p_1),(f_2,p_2)\in\Omega\times\Omega} \{(\Phi, \Gamma(f_1, p_1) = \Gamma(f_2, p_2)) \mid f_1 \neq f_2, \mathit{Const}(\Phi)\}$$

where Φ is the feature vector $\mathit{FeatureVec}((f_1, p_1), (f_2, p_2))$ and Ω is set of a function and its corresponding parameters $\{(f, p) | (f, \vec{p}) \in P_t, p \in \vec{p}\}$. It is important to note that instead of using all data, we apply a filtering condition $\mathit{Const}(\Phi)$ to improve training quality by excluding noisy pairs in the training data. Without this filtering, even unrelated parameters with the same type would be labeled 1, making it harder for the model to learn meaningful similarities. Our key insight is to include only parameter pairs that are likely to be semantically related, based on name or usage similarity. The filtering condition $\mathit{Const}(\Phi)$ is defined as follows:

$$\mathit{Const}((\phi_1, \phi_2, \ldots, \phi_{24})) = (\phi_{16} > \alpha \wedge (\phi_{17} > \beta \vee \phi_{18} > \beta)) \vee \phi_{20} > \gamma$$

which means that the data is selected if either of the following conditions is satisfied: (1) their names are sufficiently similar, or (2) their usage contexts are sufficiently similar. We set the thresholds as $\alpha = 0.9$, $\beta = 0.9$ and $\gamma = 0.5$ in our experiments.

Algorithm 1 *MakeCands*

```
Require: A program $P$, an annotation map $\Gamma$, a model output $\Delta$,
    a target signature $\theta^\# = (f^\#, \vec{p}^\#)$, and the number of additional
    augmented candidates $m$.
Ensure: A sequence of augmented candidate types $\mathbb{A}$
 1: $\Lambda \leftarrow \mathit{ParamTypes}(\Delta, f^\#, \vec{p}^\#)$
 2: $\mathbb{P} \leftarrow \mathit{Uncertain}(P, \Gamma, f^\#, \vec{p}^\#, \Lambda)$
 3: $\mathbb{A} \leftarrow []$
 4: for $1 \le j \le m$ do
 5:     $\mathcal{T}_j \leftarrow []$
 6:     for $1 \le i \le |\vec{p}^\#|$ do
 7:         if $\vec{p}^\#(i) \in \mathbb{P}$ then
 8:             $[t_1, t_2, \ldots, t_m] \leftarrow \mathit{CandTypes}(P, \Gamma, f^\#, \vec{p}^\#(i))$
 9:             $\mathcal{T}_j \leftarrow \mathcal{T}_j \parallel t_j$
10:         else
11:             $[t_1, t_2, \ldots, t_k] \leftarrow \Delta(1)$
12:             $\mathcal{T}_j \leftarrow \mathcal{T}_j \parallel t_j$
13:     $\mathbb{A} \leftarrow \mathbb{A} \parallel \mathcal{T}_j$
```

3.2 Augmentation

To enhance the effectiveness of the re-ranking step, we augment the model outputs with additional candidate types. We define the function $\mathit{Augment}(P, \Gamma, \Delta, \theta^\#, m)$, which adds m new additional candidates to the original model outputs Δ:

$$\mathit{Augment}(P, \Gamma, \Delta, \theta^\#, m) = \Delta \oplus \{(|\Delta| + j) \mapsto \mathcal{T}_j \mid 1 \le j \le m\}$$
$$\text{where } [\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m] = \mathit{MakeCands}(P, \Gamma, \Delta, \theta^\#, m)$$

where *MakeCands* generates m additional candidates $\mathcal{T}_j \in \mathit{Type}^*$ $(1 \le j \le m)$ for the target signature $\theta^\#$. These new candidates are appended to the end of the original output list Δ. Algorithm 1 describes the complete procedure.

Uncertain Parameters. We generate additional candidates by focusing on the parts of the signature where the model is uncertain. Our goal is to identify uncertain parameters and augment them with contextually plausible types.

As a first step, we construct Λ that associates each parameter with a multiset of its predicted types across all candidates (line 1):

$$\mathit{ParamTypes}(\Delta, f^\#, \vec{p}^\#) = \{\vec{p}^\#(i) \mapsto [\![\vec{t}(i) \mid \vec{t} \in \Delta]\!] \mid 1 \le i \le |\vec{p}^\#|\}$$

where $[\![\vec{t}(i) \mid \vec{t} \in \Delta]\!]$ denotes the multiset of type predictions for the i-th parameter $\vec{p}^\#(i)$. Next, we identify the uncertain parameters by computing the Shannon entropy [26] of the type multiset $\Lambda[p]$ for each parameter $p \in \vec{p}^\#$ (line 2):

$$\mathit{Uncertain}(P, \Gamma, f^\#, \vec{p}^\#, \Lambda) = \{p \in \vec{p}^\# \mid H_n(\mathit{TypeProb}(\Lambda[p])) > \nu\}$$

where $\mathit{TypeProb}(\Lambda[p])$ returns a probability distribution of types for the given multiset $\Lambda[p]$ and H_n computes the normalized Shannon entropy. The hyperparameter ν controls the threshold for detecting uncertainty, and we set $\nu = 0.6$ in our experiments.

Example 3.2. For instance, in Example 3.1, the type multiset for the parameter x is $\Lambda[x] = [\![$`int`, `float`, `int`, `float`, `str`$]\!]$. The corresponding type probability distribution is then computed as $\mathit{TypeProb}(\Lambda[x]) = \{\frac{2}{5}, \frac{2}{5}, \frac{1}{5}\}$ for {`int`, `float`, `str`}.

Algorithm 2 Final Algorithm

```
Require: A program $P$, an annotation map $\Gamma$, a target signature
    $\theta^{\#}$, a type inference model $\mathcal{M}$, and hyperparameters $k$ and $m$.
Ensure: Re-ranked candidate types $\Delta_k^*$
1: $\Delta \leftarrow \mathcal{M}(P, \Gamma, \theta^{\#}, k)$
2: $\Delta^+ \leftarrow Augment(P, \Gamma, \Delta, \theta^{\#}, m)$          ▷ Section 3.2
3: $\Delta^{++} \leftarrow ReRanking(P, \Gamma, \Delta^+, \theta^{\#})$          ▷ Section 3.1
4: return $\Delta^{++}$
```

We compute the normalized Shannon entropy of the type probability distribution $Q = \{q_1, q_2, \dots, q_{|Q|}\}$ as follows:

$$H_n(Q) = -\frac{1}{\log_2(|Q|)} \sum_{i=1}^{|Q|} q_i \log_2(q_i)$$

In summary, we classify the parameters as uncertain if the normalized Shannon entropy of the type probability distribution is greater than a threshold v.

Generating Additional Candidates. The next step is to generate m additional candidates, denoted as $[\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_m]$, where each $\mathcal{T}_j \in Type^*$ denotes an augmented signature type.

To construct each $\mathcal{T}_j$, we iterate over the target parameters $\vec{p}^{\#}$ (line 6) and check whether each parameter is uncertain (line 7). For uncertain parameters, we collect new candidate types for the parameter as follows (line 8):

$$CandTypes(P, \Gamma, f^{\#}, p_i^{\#}) =$$
$$\text{Sort}(\{(\Gamma(f,p), \mathcal{R}(\Phi)) \mid (f, \vec{p}) \in P, p \in \vec{p}, f \neq f^{\#}, \mathcal{R}(\Phi) > \mu\})$$

where Φ is the feature vector $FeatureVec((f^{\#}, p_i^{\#}), (f, p))$ and μ is a hyperparameter that controls the threshold of the similarity score. We set $\mu = 0.65$ in our experiments. We collect the types of other parameters in the program P whose similarity score with parameter $p_i^{\#}$ is greater than the threshold μ. In other words, we gather the types of other parameters that are syntactically similar to the uncertain parameter $p_i^{\#}$. After sorting them by the score, we append the j-th type from $CandTypes(P, \Gamma, (f^{\#}, p_i^{\#}))$ to the j-th candidate $\mathcal{T}_j$ (line 9). Otherwise, we instead use the candidate type from the model output Δ (line 11).

Finally, we append each augmented candidate $\mathcal{T}_j$ to the sequence $\mathbb{A}$ (line 13). As a result, we obtain the augmented candidate list $\mathbb{A} = [\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_m]$.

3.3 Final Algorithm

We summarize the overall algorithm in Algorithm 2. First, we obtain the candidate types for the target signature $\theta^{\#}$ by using the type inference model $\mathcal{M}$ (line 1). Next, we augment the candidate types by adding m additional candidates using the function *Augment* (line 2). Finally, we re-rank the candidate types using the function *ReRanking* (line 3).

3.4 Implementation

In this section, we describe the implementation aspects of the algorithm. We implemented TypeCare in about 2,900 lines of Python code (v3.10.15). We set the existing type inference models to produce the top-10 candidate types ($k = 10$) for each target signature.

Table 2: Statistics of the test datasets used in our evaluation. *Problem* refers to the number of type inference problem and *Annotation* refers to the number of type annotations in the dataset.

Dataset	Problem			Annotation	
	Param	Ret	Total	Param	Ret
BetterTypes4Py	9081	6143	15224	49.1%	55.8%
ManyTypes4Py	1847	840	2687	12.2%	15.5%

3.4.1 Type Validity. To extract static type analysis alarms, we use Pyright [15], a type checker developed by Microsoft. Although Pyright is generally fast, running it across all candidate types incurred non-negligible overhead. To mitigate this, we parallelized the alarm extraction process using Python's multiprocessing module.

3.4.2 Type Similarity Model. We built two similarity models: one for the function parameters and another for the return value. The parameter similarity model uses all features described in Table 1. In contrast, the return value similarity model excludes feature #16-#19 and computes feature #20-#24 based on return statements.

To measure usage context similarity (Feature #20 in Table 1), we extract the parent AST node of each parameter usage and compute the cosine similarity between these nodes. We used the `ast` module in Python to process the syntax tree.

Using these features, we trained the similarity model through a random forest regression [2]. The model is trained on the training data used in the previous work [21, 28, 30].

3.4.3 Augmentation. When generating additional candidates using *CandTypes* (line 8 in Algorithm 1), the number of available candidate types may be less than m in some cases. In such cases, we utilized the top-1 prediction returned by *CandTypes*.

4 Evaluation

In this section, we experimentally evaluate TypeCare to answer the following research questions:

- **Effectiveness of TypeCare**: How effective is our technique in improving the accuracy of state-of-the-art type inference models?
- **Robustness of TypeCare**: How robustly does our technique perform across various type inference tasks?
- **Importance of Components**: How do the re-ranking and augmentation components individually contribute to the overall effectiveness of our technique?

4.1 Setup

In this section, we describe the setup of our evaluation. All experiments were conducted on a Linux machine (Ubuntu 24.04) with the Intel Xeon Silver 4214 processor and 128GB memory.

Baselines and Datasets. We selected three state-of-the-art type inference models as baselines: TypeT5 [30], Tiger [28], and TypeGen [21]. These models have proven effectiveness by outperforming prior approaches [21, 28, 30]. Although we initially considered DLInfer [32], we excluded it from our evaluation due to its limited scalability, which restricted its applicability to small datasets [28]. We utilized the artifacts provided by the authors of each model.

Table 3: Single-variable type inference accuracy of the baseline models and our approach on the BetterTypes4Py and ManyTypes4Py datasets. CodeT5$_B$ and CodeT5$_M$ are the baseline models used in prior works.

Dataset	Model	Exact Match Top 1 (%)	Exact Match Top 3 (%)	Exact Match Top 5 (%)	Base Match Top 1 (%)	Base Match Top 3 (%)	Base Match Top 5 (%)
Better Types 4Py	CodeT5$_B$	65.5%	70.4%	72.5%	73.7%	77.8%	79.4%
	TypeT5	71.4%	77.2%	78.9%	78.1%	83.7%	85.4%
	+TypeCare	(+13.6%) 81.1%	(+7.4%) 82.9%	(+5.8%) 83.5%	(+9.2%) 85.3%	(+4.3%) 87.3%	(+3.0%) 88.0%
Many Types 4Py	CodeT5$_M$	62.9%	71.7%	73.8%	72.0%	79.6%	81.1%
	Tiger	67.8%	78.0%	80.2%	75.8%	85.5%	88.0%
	+TypeCare	(+11.8%) 75.8%	(+4.4%) 81.4%	(+4.5%) 83.8%	(+6.5%) 80.7%	(+2.3%) 87.5%	(+2.7%) 90.4%
	-	-	-	-	-	-	-
	TypeGen	65.4%	73.4%	75.0%	71.6%	79.9%	81.6%
	+TypeCare	(+12.5%) 73.6%	(+7.6%) 79.0%	(+6.8%) 80.1%	(+9.9%) 78.7%	(+5.4%) 84.2%	(+4.9%) 85.6%

We conducted evaluation on two datasets: ManyTypes4Py [16] and BetterTypes4Py [30]. The ManyTypes4Py dataset was designed for single-type inference tasks, where the goal is to predict the type of a single placeholder (i.e., a parameter or return value), whereas the BetterTypes4Py dataset targets full function signature inference and includes 7,293 type inference problems. We used only problems related to function signatures from these datasets.

In line with the original setup of each baseline, we evaluated Tiger and TypeGen on the ManyTypes4Py dataset and TypeT5 on the BetterTypes4Py dataset. This ensures a fair and consistent comparison with prior work, as each model was assessed using the benchmark it was originally trained and evaluated on. Also, we followed the original data splitting strategies used by each model [21, 28, 30]. Table 2 summarizes the statistics of the test sets.

Model Settings. We obtained the top-10 candidates from each model and applied our technique to refine the candidates. However, the three baseline models differ slightly in their input and output formats. Thus, we adapted the problem setting of each model to ensure a fair comparison.

- TypeT5 performs full function signature prediction. Accordingly, we inserted `<FILL_IN>` placeholders into function signatures, collected the top-10 predictions for function signatures, and applied our technique to re-rank the predictions.
- Tiger predicts the type of a single variable (i.e., a parameter or return value) at a time. For each variable, we collected the top-10 predictions and evaluated the performance of our technique on the top-10 predictions.
- TypeGen also predicts the type of a single variable at a time. However, since TypeGen is based on ChatGPT, it adopts a frequency-based evaluation strategy by sampling 50 responses in parallel and choosing the most frequent types. We followed this setup by extracting the top-10 predictions from the 50 sampled responses, and applied our technique accordingly.

Accuracy Metrics. Following the evaluation setup of TypeT5 [30], we applied type normalization to both the predicted types and the ground truth types since Python allows semantically equivalent types to be expressed in different syntactic forms (e.g., `Optional[int]` vs `Union[int, None]`), and then assessed type correctness. In particular, since Tiger and TypeGen originally used different evaluation criteria, we re-evaluated them using the same criteria in this paper.

4.2 Results

4.2.1 *Effectiveness of TypeCare.*

Setup. We applied our technique on top of the three baseline models In addition, to highlight the effectiveness of our technique, we also evaluated the baseline models used in TypeT5 and Tiger, namely CodeT5$_B$ and CodeT5$_M$, respectively. CodeT5$_B$ is CodeT5 fine-tuned on the BetterTypes4Py dataset, while CodeT5$_M$ is fine-tuned on the ManyTypes4Py dataset. For CodeT5$_M$, we used the checkpoint provided by the authors [28]. As a corresponding artifact for CodeT5$_B$ was unavailable, we replicated it by fine-tuning CodeT5 using the same setup and training set [30]. In contrast, a baseline model for TypeGen was excluded from our evaluation due to the difficulty of reproducing its baseline.

We evaluated the performance on two inference scenarios: (1) inferring a type for a single placeholder, and (2) inferring a type for a full function signature. For single placeholder inference (`<FILL_IN>`), a prediction was considered correct if the type of the placeholder matched the ground truth type. Conversely, for function signature inference, a prediction was considered correct only if the entire function signature matched the ground truth. The function signature inference was conducted only for TypeT5 because it is the only model that supports this setting.

We used the term **Exact Match** to refer to the case where the predicted type exactly matches the ground truth type, and **Base Match** to refer to the case where the predicted type is a base type of the ground truth type. For example, if the ground truth type is `list[int]` and the predicted type is `list`, it is counted as a base match, but not an exact match.

Single Variable Results. Table 3 presents the single placeholder inference results of existing models and our technique. We observed that our technique consistently improves the type inference accuracy of all existing models. For the exact match, our technique further increased the top-1 accuracy of TypeT5 and Tiger by 13.6% and 11.8%, respectively. Notably, this improvement is more than double the gain achieved by existing techniques. Our technique also

Table 4: Function-signature type inference accuracy of TypeT5 and TypeCare on the BetterTypes4Py dataset.

Model	Exact Match			Base Match		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
TypeT5	59.1%	66.7%	68.7%	67.4%	74.8%	76.6%
+TypeCare	74.5%	77.0%	77.5%	77.2%	80.0%	81.0%
Improve	**+26.1%**	**+15.4%**	**+12.8%**	**+14.5%**	**+7.0%**	**+5.7%**

Table 5: Type inference accuracy of all baseline models and TypeCare on parameter and return types.

Model	Parameter			Return		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
TypeT5	67.5%	74.3%	76.5%	77.1%	81.4%	82.5%
+TypeCare	79.6%	81.4%	82.1%	83.2%	85.1%	85.6%
Improve	**+17.9%**	**+9.6%**	**+7.3%**	**+7.9%**	**+4.5%**	**+3.8%**
Tiger	68.5%	80.0%	82.7%	66.2%	73.5%	74.6%
+TypeCare	78.9%	84.2%	87.0%	68.8%	74.9%	76.5%
Improve	**+15.2%**	**+5.3%**	**+5.2%**	**+3.9%**	**+1.9%**	**+2.5%**
TypeGen	67.8%	75.6%	77.1%	60.1%	68.6%	70.5%
+TypeCare	77.2%	82.5%	83.6%	65.6%	71.1%	72.0%
Improve	**+13.9%**	**+9.1%**	**+8.4%**	**+9.2%**	**+3.6%**	**+2.1%**

improved the top-1 accuracy of TypeGen by 12.5% over TypeGen. In terms of base match, our technique led to substantial improvements for all models, boosting their top-1 accuracy by 9.2%, 6.5%, and 9.9% for TypeT5, Tiger, and TypeGen, respectively. Since our approach also improved the top-3 and top-5 accuracy of all models, it proved to be beneficial for enhancing the overall performance.

Function Signature Results. Table 4 indicates the function signature type inference accuracy of TypeT5 and our technique. Both exact match and base match metrics showed performance improvements, with top-1 accuracy increasing by 26.1% and 14.5% for exact match and base match, respectively. Notably, our model's top-1 accuracy exceeded its top-5 accuracy of TypeT5, highlighting its strong performance on the function signature type inference.

4.2.2 Robustness of TypeCare.

Setup. We evaluated the robustness of our technique on the following aspects: (1) **Parameter and Return**: performance on parameter types and return types evaluated separately; (2) **Type Categories**: performance across three type categories: elementary types (e.g., `int`, `str`, `bool`), parameterized types (e.g., `list[int]`, `dict[str, int]`), and user-defined types (e.g., `MyClass`). Since type categories were provided only in the ManyTypes4Py dataset, we conducted comparisons with Tiger and TypeGen models only. In this evaluation, we used exact match as the evaluation metric.

Performance by Parameter and Return. Table 5 shows the type inference accuracy on parameter and return types. In particular, the top-1 accuracy for parameter types improved by an average of 15.7% across all models. For return types, our technique also improved the top-1 accuracy by an average of 7.0% across all models. This indicates that our technique is effective in enhancing type inference performance for both parameter and return types.

Performance by Type Category. Table 6 presents the type inference accuracy across different type categories. Except for the top-3 and top-5 accuracy of Tiger in **Element** category, our method consistently improved type inference accuracy across all categories and models. Notably, the **Parameterized** and **User-defined** categories showed the most significant improvements, with top-1 accuracy increasing by 23.3% and 40.1% for Tiger, and 24.2% and 27.7% for TypeGen, respectively. This indicates that our technique is particularly effective in enhancing the performance of type inference for more complex types, such as parameterized and user-defined types, which are often more challenging to predict accurately.

4.2.3 Importance of Components. We conducted an ablation study to evaluate the importance of the two major components of our technique: re-ranking and augmentation. We sequentially applied the two components to the existing models and evaluated the cumulative performance on a single variable type inference task. Table 7 shows the results of the ablation study. The re-ranking component primarily boosted top-1 accuracy, while the augmentation component improved uniform gains across top-1, top-3 and top-5 accuracy. This suggests that two components are complementary in enhancing the type inference performance of existing models.

4.3 Discussion

Type Validity. Whereas prior works employed a type validity to filter out invalid types, our approach assessed the relative quality of each type. To substantiate this, we simulated the traditional method by discarding any type that triggered an alarm instead of assessing its relative quality in TypeCare. Table 8 shows the results of this experiment. TypeT5 showed a decrease in accuracy across all metrics, with an 12.0% drop in top-5 accuracy. Even though Tiger and TypeGen increased their top-1 accuracy, the gain was less than that of TypeCare. Furthermore, they suffered a decline in top-3 and top-5 accuracy. This indicates that the relative quality assessment approach is more effective than the type validity filtering approach in improving type inference accuracy.

Type Similarity Model. We mentioned that the model should be trained on selected data to ensure the effectiveness of the type similarity model. When the model was trained without data filtering, it exhibited poor performance on predicting label 1. For parameter types, precision and recall were only 0.65 and 0.23, respectively; for return types, precision and recall were 0.67 and 0.65, respectively. In contrast, training on the filtered data yielded significant improvements. Precision and recall for parameter types surged to 0.87 and 0.94, while for return types, they reached 0.65 and 0.82.

Furthermore, we investigated which feature of the type similarity model was most influential in making its predictions. For parameter inference, the Reversed Jaro-Winkler distance was the most influential feature, suggesting a strong tendency for similarly named parameters to share a common type. For the return type, the number of usage contexts emerged as a significant feature, indicating that it primarily captures the critical distinction between functions with no return and those with at least one.

Time Overhead. We measured the time overhead introduced by our technique to assess its practical applicability. Figure 6 shows the time overhead of our technique when applied to the baseline models.

Table 6: Type inference accuracy of the baseline models (TIGER and TYPEGEN) and TypeCare across different type categories on the ManyTypes4Py dataset. *Element*: elementary types (e.g., `int`, `str`, `bool`), *Parameterized*: parameterized types (e.g., `list[int]`, `dict[str, int]`), *User-defined*: user-defined types (e.g., `MyClass`). # indicates the number of problems in each category.

Model	**Element** (#: 1405)			**Parameterized** (#: 483)			**User-defined** (#: 799)		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
TIGER	88.2%	94.2%	94.5%	29.2%	41.0%	41.8%	55.3%	71.8%	78.1%
+TypeCare	88.6%	91.5%	94.0%	36.0%	45.5%	48.2%	77.5%	85.2%	87.4%
Improve	**+0.5%**	-2.9%	-0.5%	**+23.3%**	**+11.0%**	**+15.3%**	**+40.1%**	**+18.7%**	**+11.9%**
TYPEGEN	84.3%	90.0%	90.7%	30.2%	42.9%	45.8%	53.4%	62.8%	65.2%
+TypeCare	89.0%	92.4%	93.0%	37.5%	50.9%	54.0%	68.2%	72.3%	73.2%
Improve	**+5.6%**	**+2.7%**	**+2.5%**	**+24.2%**	**+18.6%**	**+17.9%**	**+27.7%**	**+15.1%**	**+12.3%**

Table 7: Ablation study on a single variable inference task.

Model	Top-1	Diff	Top-3	Diff	Top-5	Diff
TYPET5	71.4%	-	77.2%	-	78.9%	-
+ReRank	77.8%	+6.4	79.6%	+2.4	80.1%	+1.2
+Augment	81.1%	+3.3	82.9%	+3.3	83.5%	+3.4
TIGER	67.8%	-	78.0%	-	80.2%	-
+ReRank	74.0%	+6.2	79.1%	+1.1	81.0%	+0.8
+Augment	75.8%	+1.8	81.4%	+2.3	83.8%	+2.8
TYPEGEN	65.4%	-	73.4%	-	75.0%	-
+ReRank	70.8%	+5.4	75.3%	+1.9	76.1%	+1.1
+Augment	73.6%	+2.8	79.0%	+3.7	80.1%	+4.0

Table 8: The effect of type validity filtering on a single variable inference task.

Metric	TypeT5	w/ filter	TIGER	w/ fiter	TypeGen	w/ filter
Top-1	71.4%	68.2%	67.8%	71.7%	65.4%	66.0%
Top-3	77.2%	69.3%	78.0%	75.0%	73.4%	68.0%
Top-5	78.9%	69.5%	80.2%	75.3%	75.0%	68.1%

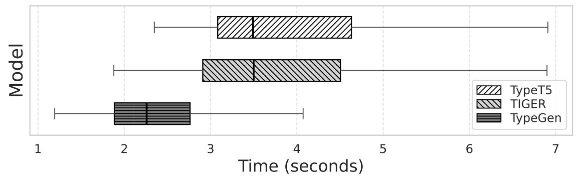

Figure 6: Time overhead

The overhead corresponds to the time taken to apply our technique to the top-10 predictions for a single placeholder `<FILL_IN>`. The median time for TIGER and TYPET5 is around 3.5 seconds, while TYPEGEN takes around 2.3 seconds. Considering existing models require as little as 1 second and up to 11.7 seconds per prediction [28], we conclude that the overhead introduced by our method is practically acceptable.

5 Related Works

Rule-based Type Inference. Rule-based type inference techniques infer types using predefined rules derived from the structure and semantics of the code [3, 8, 9, 12, 18, 24, 25]. In practice, rule-based techniques have been widely adopted in industry, particularly in static analysis tools for Python [7, 14, 15, 23]. While these techniques offer high precision and domain-specific effectiveness due to their carefully engineered rules, they often face challenges in handling the dynamic and diverse nature of Python code, resulting in limited coverage and scalability issues.

Learning-based Type Inference. Learning-based type inference techniques have emerged as a promising approach to address the limitations of rule-based methods. Early approaches centered on supervised learning utilizing type embedding or deep similarity model. [1, 11, 17, 20, 22, 32]. Recent works [21, 28, 30] have further advanced this field by leveraging pre-trained language models for code, such as CODET5 [28, 30] and GPT [21]. These approaches fine-tune or re-train existing language models to predict types based on code context. TYPET5 [30] adapts a pre-trained T5 model using a sequence-to-sequence learning to capture caller-callee relationships, while TYPEGEN [21] investigates effective prompt designs to harness the capabilities of large language models for type inference. TIGER [28] proposes a specialized architecture that combines a generation model and deep similarity learning. However, these learning-based methods face challenges in inferring complex or rare patterns that are underrepresented in the training data. To address this limitation, we present a new post-processing technique that refines model outputs by leveraging code context.

Several works combined models with static analysis to achieve better type inference [20, 22, 32]. Both TYPEWRITER [22] and HITYPER [20] utilized type checking to select only types that do not violate the type constraints of the code. DLINFER [32] collected code slices through static analysis, but it was conducted on a small dataset due to scalability issues [28]. In contrast, we investigated a more effective use of static analysis and demonstrated that our approach can be applied to large-scale datasets.

6 Conclusion

We presented TypeCare, a model-agnostic refinement technique that enhances the performance of Python type inference models via context-aware re-ranking and augmentation. TypeCare leverages both semantic signals (via static type analysis) and syntactic signals (via code usage similarity) to prioritize and supplement type predictions produced by existing models. Our experiments demonstrate that TypeCare consistently improves top-1 accuracy across multiple state-of-the-art models, especially for complex types such as parameterized and user-defined types. Ablation studies further confirm that re-ranking and augmentation are complementary, jointly contributing to the robustness and effectiveness of our approach.

Data Availability

For open science, we have made our code and data publicly available via GitHub repository[1].

References

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. doi:10.1145/3385412.3385997

[2] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[3] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: analysis for machine learning programs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3211346.3211349

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[5] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. [n. d.]. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

[6] GitHub. 2024. Octoverse 2024: The most popular programming languages. https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages Accessed: 2025-06-28.

[7] Google. 2015. Pytype: A Static Type Analyzer for Python Code. https://github.com/google/pytype. Accessed: 2025-07-08.

[8] Michael Gorbovitski, Yanhong A Liu, Scott D Stoller, Tom Rothamel, and Tuncay K Tekle. 2010. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*. 27–42.

[9] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-based type inference for Python 3. In *International Conference on Computer Aided Verification*. Springer, 12–19.

[10] Matthew A Jaro. 1989. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical association* 84, 406 (1989), 414–420.

[11] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. doi:10.1145/3468264.3473135

[12] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static analysis of shape in TensorFlow programs. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 15–1.

[13] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[14] Meta. 2025. Pyrefly: A fast type checker and IDE for Python. hhttps://github.com/facebook/pyrefly. Accessed: 2025-07-08.

[15] Microsoft. 2019. Pyright: Static type checker for Python. https://github.com/microsoft/pyright. Accessed: 2025-07-02.

[16] A. M. Mir, E. Latoskinas, and G. Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-Based Type Inference. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE Computer Society, 585–589. doi:10.1109/MSR52588.2021.00079

[17] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. doi:10.1145/3510003.3510124

[18] Wonseok Oh and Hakjoo Oh. 2024. Towards Effective Static Type-Error Detection for Python. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1808–1820.

[19] John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. 2018. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 190–201.

[20] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. doi:10.1145/3510003.3510038

[21] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 988–999. doi:10.1109/ASE56229.2023.00031

[22] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. doi:10.1145/3368089.3409715

[23] python. 2012. Mypy: Optional Static Typing for Python. https://github.com/python/mypy. Accessed: 2025-07-08.

[24] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 944–953. doi:10.1145/1176617.1176753

[25] Michael Salib. 2004. *Starkiller: A static type inferencer and compiler for Python*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[26] C. E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27, 3 (1948), 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x

[27] Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

[28] Chong Wang, Jian Zhang, Yiling Lou, Mingwei Liu, Weisong Sun, Yang Liu, and Xin Peng. 2025. TIGER: A Generating-Then-Ranking Framework for Practical Python Type Inference. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 321–333.

[29] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *EMNLP 2021-2021 Conference on Empirical Methods in Natural Language Processing, Proceedings*. Association for Computational Linguistics (ACL), 8696–8708.

[30] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. Typet5: Seq2seq type inference using static analysis. *arXiv preprint arXiv:2303.09564* (2023).

[31] William E Winkler. 1990. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. (1990).

[32] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2009–2021. doi:10.1109/ICSE48619.2023.00170

[1]Artifacts link for reviewing: https://anonymous.4open.science/r/TypeCare-ICSE26-D287