Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python

Amir M. Mir  
se.a.m.mir@tudelft.nl  
Delft University of Technology  
Delft, The Netherlands

Evaldas Latoškinas  
e.latoskinas@student.tudelft.nl  
Delft University of Technology  
Delft, The Netherlands

Sebastian Proksch  
s.proksch@tudelft.nl  
Delft University of Technology  
Delft, The Netherlands

Georgios Gousios  
gousiosg@fb.com  
Meta  
Menlo Park, USA

Abstract
Dynamic languages, such as Python and Javascript, trade static typing for developer flexibility and productivity. Lack of static typing can cause run-time exceptions and is a major factor for weak IDE support. To alleviate these issues, PEP 484 introduced optional type annotations for Python. As retrofitting types to existing codebases is error-prone and laborious, machine learning (ML)-based approaches have been proposed to enable automatic type inference based on existing, partially annotated codebases. However, previous ML-based approaches are trained and evaluated on human-provided type annotations, which might not always be sound, and hence this may limit the practicality for real-world usage. In this paper, we present Type4Py, a deep similarity learning-based hierarchical neural network model. It learns to discriminate between similar and dissimilar types in a high-dimensional space, which results in clusters of types. Likely types for arguments, variables, and return values can then be inferred through the nearest neighbor search. Unlike previous work, we trained and evaluated our model on a type-checked dataset and used mean reciprocal rank (MRR) to reflect the performance perceived by users. The obtained results show that Type4Py achieves an MRR of 77.1%, which is a substantial improvement of 8.1% and 16.7% over the state-of-the-art approaches Typilus and TypeWriter, respectively. Finally, to aid developers with retrofitting types, we released a Visual Studio Code extension, which uses Type4Py to provide ML-based type auto-completion for Python.

Keywords  
Type Inference, Similarity Learning, Machine Learning, Mean Reciprocal Rank, Python

1 Introduction
Over the past years, dynamically-typed programming languages (DPLs) have become extremely popular among software developers. The IEEE Spectrum ranks Python as the most popular programming language in 2021 [3]. It is known that statically-typed languages are less error-prone [54] and that static types improve important quality aspects of software [20], like the maintainability of software systems in terms of understandability, fixing type errors [23], and early bug detection [20]. In contrast to that, dynamic languages such as Python and JavaScript allow rapid prototyping which potentially reduces development time [23, 59], but the lack of static types in dynamically-typed languages often leads to type errors, unexpected run-time behavior, and suboptimal IDE support.

To mitigate these shortcomings, the Python community introduced PEP 484 [61], which adds optional static typing to Python 3.5 and newer. Static type inference methods [19, 24] can be employed to support adding these annotations, which is otherwise a manual, cumbersome, and error-prone process [46]. However, static inference is imprecise [50], caused by dynamic language features or by the required over-approximation of program behavior [39]. Moreover, static analysis is usually performed on full programs, including their dependencies, which is slow and resource-intensive.

To address these limitations of static type inference methods, researchers have recently employed Machine Learning (ML) techniques for type prediction in dynamic languages [12, 26, 41, 51]. The experimental results of these studies show that ML-based type prediction approaches are more precise than static type inference methods or they can also work with static methods in a complementary fashion [12, 51]. Despite the superiority of ML-based type prediction approaches, their type vocabulary is small and fixed-sized (i.e. 1,000 types). This limits their type prediction ability for user-defined and rare types. To solve this issue, Allamanis et al. [12] recently introduced Typilus which does not constraint the type vocabulary size and it outperforms the other models with small-sized type vocabulary.

While the ML-based type inference approaches are effective, we believe that there are two main drawbacks in the recent previous work [12, 51]:
The neural models are trained and evaluated on developer-provided type annotations, which are not always correct [46, 52]. This might be a (major) threat to the validity of the obtained results. To address this, a type checker should be employed to detect and remove incorrect type annotations from the dataset.

Although the proposed approaches [12, 51] obtain satisfying performance for Top-10, it is important for an approach to give a correct prediction in Top-1 as developers tend to use the first suggestion by a tool [48]. Like the API recommendation research [25, 36], the Mean Reciprocal Rank (MRR) metric should also be used for evaluation, which partially rewards an approach where the correct API is not in the Top-1 suggestion.

Motivated by the above discussion, we present Type4Py, a type inference approach based on deep similarity learning (DSL). The proposed approach consists of an effective hierarchical neural network that maps programs into type clusters in a high-dimensional feature space. Similarity learning has, for example, been used in Computer Vision to discriminate human faces for verification [15]. Similarly, Type4Py learns how to distinguish between different types through a DSL-based hierarchical neural network. As a result, our proposed approach can not only handle a very large type vocabulary, but also it can be used in practice by developers for retrofitting type annotations. In comparison with the state-of-the-art approaches, the experimental results show that Type4Py obtains an MRR of 77.1%, which is 8.1% and 16.7% higher than Typilus [12] and TypeWriter [51], respectively.

Overall, this paper presents the following main contributions:

- **Type4Py**, a new DSL-based type inference approach.
- A type-checked dataset with 5.1K Python projects and 1.2M type annotations. Invalid type annotations are removed from both training and evaluation.
- A Visual Studio Code extension [9], which provides ML-based type auto-completion for Python.

To foster future research, we publicly released the implementation of the Type4Py model and its dataset on Zenodo.\(^1\)

The rest of the paper is organized as follows. Section 2 reviews related work on static and ML-based type inference. The proposed approach, Type4Py, is described in Section 3. Section 4 gives details about the creation of the type-checked dataset for evaluation. The evaluation setup and empirical results are given in Section 5 and Section 6, respectively. Section 7 describes the deployment of Type4Py and its usage in Visual Studio Code. Section 8 discusses the obtained results and gives future directions. Finally, we summarize our work in Section 9.

2 Related Work

**Type checking and inference for Python:** In 2014, the Python community introduced a type hints proposal [60] that describes adding optional type annotations to Python programs. A year later, Python 3.5 was released with optional type annotations and the mypy type checker [33]. This has enabled gradual typing of existing Python programs and validating added type annotations. Since the introduction of type hints proposal, other type checkers have been developed such as PyType [8], PyRight [7], and Pyre [6].

A number of research works proposed type inference algorithms for Python [24, 40, 56]. These are static-based approaches that have a pre-defined set of rules and constraints. As previously mentioned, static type inference methods are often imprecise [50], due to the dynamic nature of Python and the over-approximation of programs’ behavior by static analysis [39].

**Learning-based type inference:** In 2015, Rachev et al. [55] proposed JSNice, a probabilistic model that predicts identifier names and type annotations for JavaScript using conditional random fields (CRFs). The central idea of JSNice is to capture relationships between program elements in a dependency network. However, the main issue with JSNice is that its dependency network cannot consider a wide context within a program or a function.

Xu et al. [64] adopt a probabilistic graphical model (PGM) to predict variable types for Python. Their approach extracts several uncertain type hints such as attribute access, variable names, and data flow between variables. Although the probabilistic model of Xu et al. [64] outperforms static type inference systems, their proposed system is slow and lacks scalability.

Considering the mentioned issue of JSNice, Hellendoorn et al. [26] proposed DeepTyper, a sequence-to-sequence neural network model that was trained on an aligned corpus of TypeScript code. The DeepTyper model can predict type annotations across a source code file by considering a much wider context. Yet DeepTyper suffers from inconsistent predictions for the token-level occurrences of the same variable. Malik et al. [41] proposed NL2Type, a neural network model that predicts type annotations for JavaScript functions. The basic idea of NL2Type is to leverage the natural language information in the source code such as identifier names and comments. The NL2Type model is shown to outperform both the JSNice and DeepTyper at the task of type annotations prediction [41].

Motivated by the NL2Type model, Pradel et al. [51] proposed the TypeWriter model which infers type annotations for Python. TypeWriter is a deep neural network model that considers both code context and natural language information in the source code. Moreover, TypeWriter validates its neural model’s type predictions by employing a combinatorial search strategy and an external type checker. Wei et al. [62] introduced LAMBDANET, a graph neural network-based type inference for TypeScript. Its main idea is to create a type dependency graph that links to-be-typed variables with logical constraints and contextual hints such as variables assignments and names. For type prediction, LAMBDANET employs a pointer-network-like model which enables the prediction of unseen user-defined types. The experimental results of Wei et al. [62] show the superiority of LAMBDANET over DeepTyper.

Given that the natural constraints such as identifiers and comments are an uncertain source of information, Pandi et al. [47] proposed OptTyper which predicts types for the TypeScript language. The central idea of their approach is to extract deterministic information or logical constraints from a type system and combine them with the natural constraints in a single optimization problem. This allows OptTyper to make a type-correct prediction without violating the typing rules of the language. OptTyper has been shown to outperform both LAMBDANET and DeepTyper [47].

Except for LAMBDANET, all the discussed learning-based type inference methods employ a (small) fixed-size type vocabulary, e.g.,

\(^1\)https://doi.org/10.5281/zenodo.5913787
We extract the Abstract Syntax Tree (AST) from Python source code with simple token-sequence representation. Their empirical results show that Type4Py generally outperforms LAMBDANET. The differences between Type4Py and other learning-based approaches are summarized in Table 1.

### Table 1: Comparison between Type4Py and other learning-based type inference approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Size of type vocabulary</th>
<th>ML model</th>
<th>Type hints</th>
<th>Supported Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type4Py</strong></td>
<td>Unlimited</td>
<td>HNN (2x RNNs)</td>
<td>✓ ✓ x</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>JSXnice [55]</td>
<td>10+</td>
<td>CRFs</td>
<td>✓ ✓ x</td>
<td>✓ ✓ x</td>
</tr>
<tr>
<td>Xu et al. [64]</td>
<td>-</td>
<td>PGM</td>
<td>x ✓ x</td>
<td>x x x</td>
</tr>
<tr>
<td>DeepTyper [26]</td>
<td>10K+</td>
<td>biRNN</td>
<td>✓ ✓ x</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>NLZType [41]</td>
<td>1K</td>
<td>LSTM</td>
<td>x ✓ x</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>TypeWriter [51]</td>
<td>1K</td>
<td>HNN (3x RNNs)</td>
<td>✓ ✓ x</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>LAMBDANET [62]</td>
<td>100a</td>
<td>GNN</td>
<td>✓ ✓ ✓</td>
<td>x x x</td>
</tr>
<tr>
<td>OptTyper [47]</td>
<td>100</td>
<td>LSTM</td>
<td>x ✓ ✓</td>
<td>✓ ✓ x</td>
</tr>
<tr>
<td>Typilus [12]</td>
<td>Unlimited</td>
<td>GNN</td>
<td>✓ ✓ x</td>
<td>✓ ✓ ✓</td>
</tr>
<tr>
<td>TypeBert [29]</td>
<td>40K</td>
<td>BERT</td>
<td>✓ ✓ x</td>
<td>✓ ✓ ✓</td>
</tr>
</tbody>
</table>

*a Note that LAMBDANET’s pointer network model enables to predict user-defined types outside its fixed-size type vocabulary.*

1,000 types. This hinders their ability to infer user-defined and rare types. To address this, Allamanis et al. [12] proposed Typilus, which is a graph neural network (GNN)-based model that integrates information from several sources such as identifiers, syntactic patterns, and data flow to infer type annotations for Python. Typilus is based on metric-based learning and learns to discriminate similar to-be-typed symbols from different ones. However, Typilus requires a sophisticated source code analysis to create its graph representation, i.e., data flow analysis. Very recently, inspired by "Big Data", Jesse et al. [29] presented TypeBert, a pre-trained BERT model with simple token-sequence representation. Their empirical results show that TypeBert generally outperforms LAMBDANET. The differences between Type4Py and other learning-based approaches are summarized in Table 1.

### 3 Proposed Approach

This section presents the details of Type4Py by going through the different steps of the pipeline that is illustrated in the overview of the proposed approach in Figure 1. We first describe how we extract type hints from Python source code and then how we use this information to train the neural model.

#### 3.1 Type Hints

We extract the Abstract Syntax Tree (AST) from Python source code files. By traversing the nodes of ASTs, we obtain type hints that are valuable for predicting types of function arguments, variables, and return types. The obtained type hints are based on natural information, code context, and import statements which are described in this section.

**Natural Information:** As indicated by the previous work [27, 41], source code contains useful and informal natural language information that is considered as a source of type hints. In DPLs, developers tend to name variables and functions’ arguments after their expected type [44]. Based on this intuition, we consider identifier names as the main source of natural information and type hint. Specifically, we extract the name of functions ($N_f$) and their arguments ($N_{args}$) as they may provide a hint about the return type of functions and the type of functions’ arguments, respectively. We also denote a function’s argument as $N_{arg}$ hereafter. For variables, we extract their names as denoted by $N_v$.

**Code Context:** We extract all uses of an argument in the function body as a type hint. This means that the complete statement, in which the argument is used, is included as a sequence of tokens. Similarly, we extract all uses of a variable in its current and inner scopes. Also, all the return statements inside a function are extracted as they may contain a hint about the return type of the function.

**Visible type hints (VTH):** In contrast to previous work that only analyzed the direct imports [51], we recursively extract all the import statements in a given module and its transitive dependencies. We build a dependency graph for all imports of user-defined classes, type aliases, and NewType declarations. For example, if module A imports B.Type and C.D.E, the edges (A, B.Type) and (A, C.D.E) will be added to the graph. We expand wildcard imports like from foo import * and resolve the concrete type references. We consider the identified types as visible and store them with their fully-qualified name to reduce ambiguity. For instance, tf.Tensor and torch.Tensor are different types. Although the described inspection-based approach is slower than a pure AST-based analysis, our ablation analysis shows that VTHs substantially improve the performance of Type4Py (subsection 6.3).

#### 3.2 Vector Representation

In order for a machine learning model to learn from type hints, they are represented as real-valued vectors. The vectors preserve semantic similarities between similar words. To capture those, a word embedding technique is used to map words into a $d$-dimensional vector space, $\mathbb{R}^d$. Specifically, we first preprocess extracted identifiers and code contexts by applying common Natural Language Processing (NLP) techniques. This preprocessing step involves tokenization, stop word removal, and lemmatization [30]. Afterwards, we employ Word2Vec [43] embeddings to train a code embedding $E_c : w_l \rightarrow \mathbb{R}^{l \times d}$ for both code context and identifier tokens, where $w_l$ and $l$ denote a single token and the length of a sequence, respectively. In the following, we describe the vector representation of all the three described type hints for both argument types and return types.
Identifiers: Given an argument’s type hints, the vector sequence of the argument is represented as follows:

\[ E_c(N_{arg}) \circ s \circ E_c(N_f) \circ E_c(N_{args}) \]

where \( \circ \) concatenates and flattens sequences, and \( s \) is a separator. For a return type, its vector sequence is represented as follows:

\[ E_c(N_f) \circ s \circ E_c(N_{args}) \]

Last, a variable’s identifier is embedded as \( E_c(N_v) \).

Code contexts: For function arguments and variables, we concatenate the sequences of their usages into a single sequence. Similarly, for return types, we concatenate all the return statements of a function into a single sequence. To truncate long sequences, we consider a window of \( n \) tokens at the center of the sequence (default \( n = 7 \)). Similar to identifiers, the function embedding \( E_c \) is used to convert code contexts sequences into a real-valued vector.

Visible type hints: Given all the source code files, we build a fixed-size vocabulary of visible type hints. The vocabulary covers the majority of all visible type occurrences. Because most imported visible types in Python modules are built-in primitive types such as List, Dict, and their combinations. If a type is out of the visible type vocabulary, it is represented as a special other type. For function arguments, variables, and return types, we create a sparse binary vector of size \( T \) whose elements represent a type. An element of the binary vector is set to one if and only if its type is present in the vocabulary. Otherwise, the other type is set to one in the binary vector.

3.3 Neural Model

The neural model of our proposed approach employs a hierarchical neural network (HNN), which consists of two recurrent neural networks (RNNs) [63]. HNNs are well-studied and quite effective for text and vision-related tasks [18, 35, 65]. In the case of type prediction, intuitively, HNNs can capture different aspects of identifiers and code context. In the neural architecture (see Fig. 1), the two RNNs are based on long short-term memory (LSTM) units [28]. Here, we chose LSTMs units as they are effective for capturing long-range dependencies [22]. Also, LSTM-based neural models have been applied successfully to NLP tasks such as sentiment classification [53]. Formally, the output \( h_t^{(t)} \) of the \( i \)-th LSTM unit at the time step \( t \) is defined as follows:

\[
    h_t^{(t)} = \text{tanh}(x_t^i) \sigma \left( b_i + \sum_j U_{ij}x_j^{(t)} + \sum_j W_{ij}h_j^{(t-1)} \right) \tag{1}
\]

which has sigmoid function \( \sigma \), current input vector \( x_j \), unit state \( s_t^j \) and has model parameters \( W, U, b \) for its recurrent weights, input weights and biases [22]. The two hierarchical RNNs allow capturing different aspects of input sequences from identifiers and code tokens. The captured information is then summarized into two single vectors, which are obtained from the final hidden state of their corresponding RNN. The two single vectors from RNNs are concatenated with the visible type hints vector and the resulting vector is passed through a fully-connected linear layer.

In previous work [41, 51], the type prediction task is formulated as a classification problem. As a result, the linear layer of their neural model outputs a vector of size 1,000 with probabilities over predicted types. Therefore, the neural model predicts \( \text{unkonwn} \) if it has not seen a type in the training phase. To address this issue, we formulate the type prediction task as a Deep Similarity Learning problem [15, 34]. By using the DSL formulation, our neural model learns to map argument, variable, return types into real continuous space, called type clusters (also known as type space in [12]). In other words, our neural model maps similar types (e.g. \( \text{str} \)) into its own type cluster, which should be as far as possible from other clusters of types. Unlike the previous work [41, 51], our proposed model can handle a very large type vocabulary.

To create the described type clusters, we use Triplet loss [14] function which is recently used for computer vision tasks such as face recognition [14]. By using the Triplet loss, a neural model learns to discriminate between similar samples and dissimilar samples by mapping samples into their own clusters in the continuous space. In the case of type prediction, the loss function accepts a type \( t_a \), a type \( t_p \) same as \( t_a \), and a type \( t_n \) which is different than \( t_a \). Given a positive scalar margin \( m \), the Triplet loss function is defined as follows:

\[
    L(t_a, t_p, t_n) = \max(0, m + \|t_a - t_p\| - \|t_a - t_n\|) \tag{2}
\]

\(^2\)The separator is a vector of ones with appropriate dimension.
The search resulted in 5.2K Python projects that are available on Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python ICSE ’22, May 21–29, 2022, Pittsburgh, PA, USA. Therefore, we believe that it is essential to type-check the models when evaluating them on unseen code samples [11]. There-also codeduplication has a negative effect on the performance of ML
groups.

288,760 source code files, 65% of them have more than 40% type annotation coverage. To demonstrate the effect of using Pyre on the type annotation coverage of source code files

\begin{equation}
P(t_q : t') = \frac{1}{N} \sum \mathbb{1}(t_i = t') \frac{1}{(d_i + \varepsilon)^2}
\end{equation}

where \(\mathbb{1}\) is the indicator function, \(N\) is a normalizing constant, and \(\varepsilon\) is a small scalar (i.e., \(\varepsilon = 10^{-10}\)).

4 Dataset

For this work, we have created a new version of our ManyTypes4Py dataset [45], i.e., v0.7. The rest of this section describes the creation of the dataset. To find Python projects with type annotations, on Libraries.io, we searched for projects that depend on the mypy package [5], i.e., the official and most popular type checker for Python. Intuitively, these projects are more likely to have type annotations. The search resulted in 5.2K Python projects that are available on GitHub. Initially, the dataset has 685K source files and 869K type annotations.

4.1 Code De-duplication

On GitHub, Python projects often have file-level duplicates [38] and also code duplication has a negative effect on the performance of ML models when evaluating them on unseen code samples [11]. Therefore, to de-dupe the dataset, we use our code de-duplication tool, CD4Py [2]. It uses term frequency-inverse document (TF-IDF) [42] to represent a source code file as a vector in \(\mathbb{R}^n\) and employs KNN search to find clusters of similar duplicate files. While assuming that the similarity is transitive [11], we keep a file from each cluster and remove all other identified duplicate files from the dataset. Using the described method, we removed around 400K duplicate files from the dataset.

4.2 Augmentation

Similar to the work of Allamanis et al. [12], we have employed a static type inference tool, namely, Pyre [6] v0.9.0 to augment our initial dataset with more type annotations. However, we do note that we could only infer the type of variables using Pyre’s query command. In our experience, the query command could not infer the type of arguments and return types. The command accepts a list of files and returns JSON files containing type information.

Thanks to Pyre’s inferred types, the dataset has now 3.3M type annotations in total. To demonstrate the effect of using Pyre on the dataset, Figure 2 shows the percentage of type annotation coverage for source code files with/without using Pyre. After using Pyre, of 288,760 source code files, 65% of them have more than 40% type annotation coverage.

4.3 Type Checking

Recent studies show that developer-provided types rarely type-check and Python projects may contain type-related defects [31, 46, 52]. Therefore, we believe that it is essential to type-check the dataset to eliminate noisy ground truth (i.e., incorrect type annotations). Not only noisy ground truth can be considered a threat to the validity of results but also it may make the discrimination of types in type clusters more difficult [21]. To clean the dataset from noisy ground truth, we perform basic analysis as follows:

- First, we use mypy to type-check 288,760 source files in the dataset. Of which, 184,752 source files are successfully type-checked.
- Considering the remaining 104,008 source files, for further analysis, we ignore source files that cannot be type-checked further by mypy due to syntax error or other fatal exceptions. This amounts to 63,735 source files in the dataset.
- Given 40,273 source files with type errors, we remove one type annotation at a time from a file and run mypy. If it type-checks, we include the file. Otherwise, we continue this step up to 10 times. This basic analysis fixes 16,861 source files with type errors, i.e., 42% of the given set of files.

4.4 Dataset Characteristics

Table 2 shows the characteristics of our dataset after code de-duplication, augmentation, and type-checking. In total, there are more than 882K functions with around 1.5M arguments. Also, the dataset has more than 2.1M variable declarations. Of which, 48% have type annotations. Figure 3 shows the frequency of top 10 most frequent types in our dataset. It can be observed that types follow a long-tail distribution. Unsurprisingly, the top 10 most frequent types amount to 59% of types in the dataset. Lastly, we randomly split the dataset by files into three sets: 70% training data, 10% validation data, and 20% test data. Table 3 shows the number of data points for each of the three sets.

4.5 Pre-processing

Similar to the previous work [12, 51], before training ML models, we have performed several pre-processing steps:

- Trivial functions such as \_\_str\_\_ and \_\_len\_\_ are not included in the dataset. The return type of this kind of functions is
Table 2: Characteristics of the dataset used for evaluation

<table>
<thead>
<tr>
<th>Metrics,(^a,b)</th>
<th>Our dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td>5,092</td>
</tr>
<tr>
<td>Files</td>
<td>201,613</td>
</tr>
<tr>
<td>Lines of code</td>
<td>11.9M</td>
</tr>
<tr>
<td>Functions</td>
<td>882,657</td>
</tr>
<tr>
<td>...with return type annotations</td>
<td>94,433 (10.7%)</td>
</tr>
<tr>
<td>Arguments</td>
<td>1,558,566</td>
</tr>
<tr>
<td>...with type annotations</td>
<td>128,363 (14.5%)</td>
</tr>
<tr>
<td>Variables</td>
<td>2,135,361</td>
</tr>
<tr>
<td>...with type annotations</td>
<td>1,023,328 (47.9%)</td>
</tr>
<tr>
<td>Types</td>
<td>1,246,124</td>
</tr>
<tr>
<td>...unique</td>
<td>60,333</td>
</tr>
</tbody>
</table>

\(^a\) Metrics are counted after the ASTs extraction phase of our pipeline.

\(^b\) Comments and blank lines are ignored when counting lines of code.

Table 3: Number of data points for train, validation and test sets

<table>
<thead>
<tr>
<th></th>
<th>Argument type</th>
<th>Return type</th>
<th>Variable type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td>90,114</td>
<td>37,803</td>
<td>426,235</td>
</tr>
<tr>
<td>Validation</td>
<td>9,387</td>
<td>3,932</td>
<td>48,518</td>
</tr>
<tr>
<td>Test</td>
<td>24,121</td>
<td>10,444</td>
<td>118,319</td>
</tr>
<tr>
<td>Total</td>
<td>108,888 (16.06%)</td>
<td>45,667 (6.74%)</td>
<td>523,271 (77.20%)</td>
</tr>
</tbody>
</table>

straightforward to predict, i.e., `__len__` always returns `int`, and would blur the results.

- We excluded `Any` and `None` type annotations as it is not helpful to predict these types.
- We performed a simple type aliasing resolving to make type annotations of the same kind consistent. For instance, we map `[]` to `List`, `{}` to `Dict`, and `Text` to `str`.
- We resolved qualified names for type annotations. For example, `array` is resolved to `numpy.array`. This makes all the occurrences of a type annotation across the dataset consistent.
- Same as the work of Allamanis et al. [12], we rewrote the components of a base type whose nested level is greater than 2 to `Any`. For instance, we rewrite `List[List[Tuple[int]]]` to `List[List[Any]]`. This removes very rare types or outliers.

5 Evaluation Setup

In this section, we describe the baseline models, the implementation details and the training of the neural models. Lastly, we explain evaluation metrics to quantitatively measure the performance of ML-based type inference approaches.

5.1 Baselines

We compare `Type4Py` to Typilus [12] and TypeWriter [51], which are recent state-of-the-art ML-based type inference approaches for Python. Considering Table 1, `Type4Py` has an HNN-based neural model whereas Typilus’s neural model is GNN-based. However, Typilus has the same prediction abilities as `Type4Py` and has no limitation on the size of type vocabulary which makes it an obvious choice for comparison. Compared with `Type4Py`, TypeWriter has two main differences. First, TypeWriter’s type vocabulary is small and pre-defined (i.e. 1,000 types) at training time. Second, TypeWriter cannot predict the type of variables, unlike `Type4Py` and Typilus.

5.2 Implementation Details and Environment Setup

We implemented `Type4Py` and TypeWriter in Python 3 and its ecosystem. We extract the discussed type hints from ASTs using LibSA4Py [4]. The data processing pipeline is parallelized by employing the `joblib` package. We use NLTK [37] for performing standard NLP tasks such as tokenization and stop word removal. To train the Word2Vec model, the `gensim` package is used. For the neural model, we used bidirectional LSTMs [57] in the PyTorch framework [49] to implement the two RNNs. Lastly, we used the Annoy [1] package to perform a fast and approximate nearest neighbor search. For Typilus, we used its public implementation on GitHub [10].

We performed all the experiments on a Linux operating system (Ubuntu 18.04.5 LTS). The computer had an AMD Ryzen Threadripper 1920X with 24 threads (@3.5GHz), 64 GB of RAM, and two NVIDIA GeForce RTX 2080 TIs.

5.3 Training

To avoid overfitting the train set, we applied the Dropout regularization [58] to the input sequences except for the visible types. Also, we employed the Adam optimizer [32] to minimize the value of the Triplet loss function. For both `Type4Py` and TypeWriter, we employed the data parallelism feature of PyTorch to distribute training batches between the two GPUs with a total VRAM of 22 GB. For the `Type4Py` model, given 554K training samples, a single training epoch takes around 4 minutes. It takes 7 seconds for the TypeWriter model providing that its training set contains 127K training samples.\(^3\) Aside from the training sample size, `Type4Py` is a DSL-based

\(^3\) Note that TypeWriter uses only argument and return samples as it lacks the variable prediction ability.
We measure the type prediction performance of an approach by considering types that we have seen more than 100 times in the dataset, as these types are among the top 10 frequent types in the dataset (see Fig. 3). We evaluate the neural models on the validation set to find the optimal number of epochs for training. The value of the neural models’ hyperparameters is reported in Table 4.

5.4 Evaluation Metrics

We measure the type prediction performance by comparing the type prediction $t_p$ to the ground truth $t_g$ using two criteria originally proposed by Allamanis et al. [12]:

**Exact Match:** $t_p$ and $t_g$ are exactly the same type.

**Base Type Match:** ignores all type parameters and only matches the base types. For example, `List[str]` and `List[int]` would be considered a match.

In addition to these two criteria, as stated earlier, we opt for the MRR metric [42], since the neural models predict a list of types for a given query. The MRR of multiple queries $Q$ is defined as follows:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{r_i}$$

The MRR metric partially rewards the neural models by giving a score of $\frac{1}{r}$ to a prediction if the correct type annotation appears in rank $r$. Like Top-1 accuracy, a score of 1 is given to a prediction for which the Top-1 suggested type is correct. Hereafter, we refer to the MRR of the Top-$n$ predictions as MRR@$n$. We evaluate the neural models up to the Top-10 predictions as it is a quite common methodology in the evaluation of ML-based models for code [12, 25, 51].

Similar to the evaluation methodology of Allamanis et al. [12], we consider types that we have seen more than 100 times in the train set as common or rare otherwise. Additionally, we define the set of ubiquitous types, i.e., `{str, int, list, bool, float}`. These types are among the top 10 frequent types in the dataset (see Fig. 3) and they are excluded from the set of common types. Furthermore, Unlike Type4Py and Typilus, TypeWriter predicts `unknown` if the expected type is not present in its type vocabulary. Thus, to have a valid comparison with the other two approaches, we consider other predictions by TypeWriter in the calculation of evaluation metrics.

6 Evaluation

To evaluate and show the effectiveness of Type4Py, we focus on the following research questions.

**RQ1**: What is the general type prediction performance of Type4Py?

**RQ2**: How does Type4Py perform while considering different prediction tasks?

**RQ3**: How do each proposed type hint and the size of type vocabulary contribute to the performance of Type4Py?

6.1 Type Prediction Performance (RQ1)

In this subsection, we compare our proposed approach, Type4Py, with the selected baseline models in terms of overall type prediction performance.

**Method:** The models get trained on the training set and the test set is used to measure the type prediction performance. We evaluate the neural models by considering different top-$n$ predictions, i.e., $n = \{1, 3, 5, 10\}$. Also, for this RQ, we consider all the supported inference tasks by the models, i.e., arguments, return types, and variables.

**Results:** Table 5 shows the overall performance of the neural models while considering different top-$n$ predictions. Given the Top-10 prediction, Type4Py outperforms both Typilus and TypeWriter based on both the exact and base type match criteria (all). Specifically, considering the exact match criteria (all types), Type4Py performs better than Typilus and TypeWriter at the Top-10 prediction by a margin of 5.9% and 11%, respectively. Moreover, it can be seen that the Type4Py’s performance drop is less significant compared to the other two models when decreasing the value of $n$ from Top-10 to Top-1. For instance, by considering Top-1 rather than Top-10 and the exact match criteria (all), the performance of Type4Py, Typilus, and TypeWriter drop by 3.4%, 7.2%, 12.1%, respectively. Concerning the prediction of rare types, Typilus slightly performs better than Type4Py, which can be attributed to the use of an enhanced triplet loss function. It is also worth mentioning that Type4Py achieves a 100% exact match for the ubiquitous types at Top-1, which is remarkable.

---

Table 4: Value of hyperparameters for neural models

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Type4Py</th>
<th>TypeWriter</th>
<th>Typilus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word embedding dimension (i.e. $d$)</td>
<td>100</td>
<td>100</td>
<td>N/A</td>
</tr>
<tr>
<td>Size of visible type hints vocabulary (i.e. $T$)</td>
<td>1024</td>
<td>1024</td>
<td>N/A</td>
</tr>
<tr>
<td>LSTM hidden nodes</td>
<td>256</td>
<td>256</td>
<td>N/A</td>
</tr>
<tr>
<td>GNN hidden nodes</td>
<td>N/A</td>
<td>N/A</td>
<td>64</td>
</tr>
<tr>
<td>Dimension of linear layer’s output</td>
<td>1536</td>
<td>1000</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of LSTM’s layers</td>
<td>1</td>
<td>1</td>
<td>N/A</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.002</td>
<td>0.002</td>
<td>0.00025</td>
</tr>
<tr>
<td>Dropout rate</td>
<td>0.25</td>
<td>0.25</td>
<td>0.1</td>
</tr>
<tr>
<td>Number of epochs</td>
<td>25</td>
<td>25</td>
<td>500$^a$</td>
</tr>
<tr>
<td>Batch size</td>
<td>5864</td>
<td>4096</td>
<td>N/A</td>
</tr>
<tr>
<td>Value of $k$ for nearest neighbor search</td>
<td>10</td>
<td>N/A</td>
<td>10</td>
</tr>
<tr>
<td>Triplet loss’ margin value (i.e. $m$)</td>
<td>2.0</td>
<td>N/A</td>
<td>2.0</td>
</tr>
<tr>
<td>Model’s trainable parameters</td>
<td>4.6M</td>
<td>4.7M</td>
<td>650R</td>
</tr>
</tbody>
</table>

---

$^a$ The model stopped at epoch 38 due to the early stopping technique.
As stated earlier, developers are more likely to use the first suggestion by a tool \[48\]. Therefore, we evaluated the neural models by the MRR@10 metric at the bottom of Table 5. Ideally, the difference between the MRR@10 metric and the Top-1 prediction should be zero. However, this is very challenging as the neural models are not 100% confident in their first suggestion for all test samples. Given the results of MRR@10, we observe that Type4Py outperforms both Typilus and TypeWriter by a margin of 8.1% and 16.7%, respectively. In addition, we investigated the MRR score of the neural models while considering different values of Top-n, which is shown in Figure 4. As can be seen, Type4Py has a substantially higher score than the other models across all values of n. Moreover, the MRR score of all the three neural models almost converges to a fixed value after MRR@3. Given the findings of the RQ1, we use MRR@10 and the Top-1 prediction for the rest of the evaluation as we believe this better shows the practicality of the neural models for assisting developers.

### 6.2 Different Prediction Tasks (RQ2)

Here, we compare Type4Py with other baselines while considering different prediction tasks, i.e., arguments, return types, and variables.

**Method:** Similar to the RQ1, the models are trained and tested on the entire training and test sets, respectively. However, we consider each prediction task separately while evaluating the models at Top-1 and MRR@10.

**Results:** Table 6 shows the type prediction performance of the approaches for the three considered prediction tasks. In general, considering the exact match criteria (all), Type4Py outperforms both Typilus and TypeWriter in all prediction tasks at both Top-1 and MRR@10. For instance, considering the return task and Top-1, Type4Py obtains 56.4% exact matches (all), which is 13.9% and 5.7% higher than that of Typilus and TypeWriter, respectively. Also, for the same task, the Type4Py’s MRR@10 is 11.9% and 3.7% higher compared to Typilus and TypeWriter, respectively. However, concerning the prediction of common types and MRR@10, TypeWriter performs better than both Type4Py and Typilus at the argument and return tasks. This might be due to the fact that TypeWriter predicts from the set of 1,000 types, which apparently makes it better at the prediction of common types. Moreover, both Type4Py and Typilus have a much larger type vocabulary and hence they need more training samples to generalize better providing that both argument and return types together amount to 22.8% of all the data points in the dataset (see Table 3). Lastly, in comparison with Typilus, Type4Py obtains 7.7% and 6.7% higher MRR@10 score for the exact and base type match criteria (all), respectively.

### 6.3 Ablation Analysis (RQ3)

Here, we investigate how each proposed type hint and the size of type vocabulary contribute to the overall performance of Type4Py.

**Method:** For ablation analysis, we trained and evaluated Type4Py with 5 different configurations, i.e., (1) complete model (2) w/o identifiers (3) w/o code context (4) w/o visible type hints (5) w/ a vocabulary of top 1,000 types. Similar to the previous RQs, we measure the performance of Type4Py with the described configurations at Top-1 and MRR@10.

**Results:** Table 7 presents the performance of Type4Py with the five described configurations. It can be observed that all three type hints contribute significantly to the performance of Type4Py. Code context has the most impact on the model’s performance compared to the
Table 6: Performance evaluation of the neural models considering different tasks

<table>
<thead>
<tr>
<th>Metric</th>
<th>Task</th>
<th>Approach</th>
<th>% Exact Match</th>
<th>% Base Type Match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>All</td>
<td>Ubiquitous</td>
</tr>
<tr>
<td><strong>Top-1 prediction</strong></td>
<td>Argument</td>
<td>Type4Py</td>
<td>61.9</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Typlis</td>
<td>53.8</td>
<td>83.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TypeWriter</td>
<td>58.4</td>
<td>93.6</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>Type4Py</td>
<td>56.4</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Typlis</td>
<td>42.5</td>
<td>84.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TypeWriter</td>
<td>50.7</td>
<td>93.3</td>
</tr>
<tr>
<td></td>
<td>Variable&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Type4Py</td>
<td>80.4</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Typlis</td>
<td>71.4</td>
<td>95.1</td>
</tr>
<tr>
<td></td>
<td>MRR@10</td>
<td>Type4Py</td>
<td>64.2</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>Return</td>
<td>Typlis</td>
<td>58.7</td>
<td>87.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TypeWriter</td>
<td>63.3</td>
<td>96.2</td>
</tr>
<tr>
<td></td>
<td>Variable&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Type4Py</td>
<td>57.9</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Typlis</td>
<td>46.0</td>
<td>86.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TypeWriter</td>
<td>54.2</td>
<td>95.9</td>
</tr>
<tr>
<td></td>
<td>Variable&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Type4Py</td>
<td>81.4</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Typlis</td>
<td>73.7</td>
<td>96.3</td>
</tr>
</tbody>
</table>

<sup>a</sup> Note that TypeWriter cannot predict the type of variables.

Table 7: Performance evaluation of Type4Py with different configurations

<table>
<thead>
<tr>
<th>Metric</th>
<th>Approach</th>
<th>% Exact Match</th>
<th>% Base Type Match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>All</td>
<td>Ubiquitous</td>
</tr>
<tr>
<td><strong>Top-1 prediction</strong></td>
<td>Type4Py</td>
<td>77.1</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>(w/o identifiers)</td>
<td>73.8</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>(w/o code context)</td>
<td>69.7</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>(w/o visible type hints)</td>
<td>68.6</td>
<td>89.3</td>
</tr>
<tr>
<td></td>
<td>(w/ top 1,000 types)</td>
<td>75.6</td>
<td>100.0</td>
</tr>
<tr>
<td><strong>MRR@10</strong></td>
<td>Type4Py</td>
<td>75.6</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>(w/o identifiers)</td>
<td>73.8</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>(w/o code context)</td>
<td>69.7</td>
<td>100.0</td>
</tr>
<tr>
<td></td>
<td>(w/o visible type hints)</td>
<td>68.6</td>
<td>89.3</td>
</tr>
<tr>
<td></td>
<td>(w/ top 1,000 types)</td>
<td>75.6</td>
<td>100.0</td>
</tr>
</tbody>
</table>

other two type hints. For instance, when ignoring code context, the model’s exact match score for common types drops significantly by 23.1%. After code context, visible type hints have a large impact on the performance of the model. By ignoring VTH, the model’s exact match for ubiquitous types reduces from 100% to 86.2%. Although the Identifiers type hint contributes substantially to the prediction of common types, it has a less significant impact on the overall performance of Type4Py compared to code context and VTH. In summary, we conclude that code context and VTH are the strongest type hints for our type prediction model.

By limiting the type vocabulary of Type4Py to the top 1,000 types, similar to TypeWriter, we observe that the model’s performance for common types is slightly improved while its performance for rare types is reduced significantly, i.e., 7.2% considering MRR@10. This is expected as the model’s type vocabulary is much smaller compared to the complete model’s.

7 **Type4Py in Practice**

To make the Type4Py model practical, we developed an end-to-end solution including a web server and a Visual Studio Code (VSC) extension. We deployed this as an openly accessible web service.
that serves requests from the VSC extension. In this section, we describe the deployment components of Type4Py.

7.1 Deployment
To deploy the pre-trained Type4Py model for production, we convert the Type4Py’s PyTorch model to an ONNX model [17] which enables querying the model on both GPUs and CPUs with faster inference speed. Thanks to Annoy [1], fast and memory-efficient KNN search is performed to suggest type annotations from type clusters.

7.2 Web Server
We have implemented a small Flask application to handle concurrent type prediction requests from users with Nginx as a proxy. This enables us to have quite a number of asynchronous workers that have an instance of Type4Py’s ONNX model plus Type Clusters each. Specifically, the web application receives a Python source file via a POST request, queries an instance of the model, and finally it gives the file’s predicted type annotations as a JSON response.

7.3 Visual Studio Code Extension
As stated earlier, retrofitting type annotations is a daunting task for developers. To assist developers with this task, we have released a Visual Studio Code extension for Type4Py [9], which uses the web server’s API to provide ML-based type auto-completion for Python code. Figure 5 shows an example of a type recommendation from the VSC IDE. As of this writing, the extension has 909 installs on the Visual Studio Marketplace. Based on the user’s consent, the VSC extension gathers telemetry data for research purposes. Specifically, accepted types, their rank in the list of suggestions, type slot kind, identifiers’ name, and identifiers’ line number are captured from the VSC environment and sent to our web server. In addition, rejected type predictions are captured when a type auto-completion window is closed without accepting a type.

By analyzing the gathered telemetry data from Jul. ’21 to Aug. ’21 and excluding the author(s), of 26 type auto-completion queries, 19 type annotations were accepted by the extension’s users. Moreover, the average of accepted type annotations per developer is 69.6%. Given that the gathered telemetry data is pretty small, we cannot draw a conclusion regarding the performance of Type4Py in practice. However, our telemetry infrastructure and concerted efforts to broaden the user base will enable us to improve Type4Py in the future.

8 Discussion and Future Work
Based on the formulated RQs and their evaluation in Section 6, we provide the following remarks:

- We used Pyre [6], a static type inference tool, to augment our dataset with more type annotations. However, this can be considered as a weakly supervision learning problem [66], meaning that inferred types by the static tool might be noisy or imprecise despite the pre-processing steps. To eliminate this threat, we employed a static type checker, mypy, to remove source files with type errors from our dataset. Future work can devise a guided-search analysis to fix type errors in source files, which may improve the fix rate.

- It would be ideal for ML-based models to give a correct prediction in their first few suggestions, preferably Top-1, as developers tend to use the first suggestion by a tool [48]. Therefore, different from previous work on ML-based type prediction [12, 51], we use the MRR metric in our evaluation. We believe that the MRR metric better demonstrates the potential and usefulness of ML models to be used by developers in practice. Overall, considering the MRR metric, Type4Py significantly outperforms the state-the-art ML-based type prediction models, namely, Typilus and TypeWriter.

- Considering the overall type prediction performance (RQ1), both Type4Py and Typilus generally perform better than TypeWriter. This could be attributed to the fact that the two models map types into a high-dimensional space (i.e. type clusters). Hence this not only enables a much larger type vocabulary but also significantly improves their overall performance, especially the prediction of rare types.

- Given the results of RQ3 and RQ4, our HNN-based neural model, Type4Py, has empirically shown to be more effective than the GNN-based model of Typilus. We attribute this to the inherent bottleneck of GNNs which is over-squashing information into a fixed-size vector [13] and thus they fail to capture long-range interaction. However, our HNN-based model concatenates learned features into a high-dimensional vector and hence it preserves information and its long-range dependencies.

- According to the results of ablation analysis (RQ3), the three proposed type hints, i.e., identifiers, code context, and VTHs are all effective and positively contribute to the performance of
Type4Py. This result does not come at the expense of generalizability; our visible type analysis is not more sophisticated than what an IDE like PyCharm or VSCode do to determine available types for, e.g., auto-completion purposes.

- Both Type4Py and Typilus cannot make a correct prediction for types beyond their pre-defined (albeit very large) type clusters. For example, they currently cannot synthesize types, meaning that they will never suggest a type such as Optional[Dict[str, int]] if it does not exist in their type clusters. To address this, future research can explore pointer networks [61] or a GNN model that captures type system rules.

- We believe that Type4Py’s VSC extension is one step forward towards improving developers’ productivity by using machine-aided code tools. In this case, the VSC extension aids Python developers to retrofit types for their existing codebases. After gathering sufficiently large telemetry data from the usage of Type4Py, we will study how to improve Type4Py’s ranking and quality of predictions for, ultimately, a better user experience.

9 Summary

In this paper, we present Type4Py, a DSL-based hierarchical neural network type inference model for Python. It considers identifiers, code context, and visible type hints as features for learning to predict types. Specifically, the neural model learns to efficiently map types of the same kind into their own clusters in a high-dimensional space, and given type clusters, the k-nearest neighbor search is performed to infer the type of arguments, variables, and functions’ return types. We used a type-checked dataset with sound type annotations to train and evaluate the ML-based type inference models. Overall, the results of our quantitative evaluation show that Type4Py model outperforms other state-of-the-art approaches. Most notably, considering the MRR@10 score, our proposed approach achieves a significantly higher score than that of Typilus and TypeWriter’s by a margin of 8.1% and 16.7%, respectively. This indicates that our approach gives a more relevant prediction in its first suggestion, i.e., Top-1. Finally, we have deployed Type4Py in an end-to-end fashion to provide ML-based type auto-completion in the VSC IDE and aid developers to retrofit type annotations for their existing codebases.

Acknowledgments

This research work was funded by H2020 grant 825328 (FASTEN). We thank the anonymous reviewers for their valuable feedback and comments.

References

Teaching Natural Language Processing and Computational Linguistics 63–70.


