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

Amir M. Mir
s.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

ACM Reference Format:
Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510124

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510124

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* [60], 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].

[1] https://doi.org/10.5281/zenodo.5913787

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.,

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

Approach	Size of type vocabulary	ML model	Type hints			Supported Predictions		
			Contextual	Natural	Logical	Argument	Return	Variable
Type4Py	Unlimited	HNN (2x RNNs)	✓	✓	✗	✓	✓	✓
JSNice [55]	10+	CRFs	✓	✓	✗	✓	✗	✗
Xu et al. [64]	-	PGM	✗	✓	✓	✗	✗	✓
DeepTyper [26]	10K+	biRNN	✓	✓	✗	✓	✓	✓
NL2Type [41]	1K	LSTM	✗	✓	✗	✓	✓	✗
TypeWriter [51]	1K	HNN (3x RNNs)	✓	✓	✗	✓	✓	✗
LAMBDANET [62]	100[a]	GNN	✓	✓	✓	✗	✗	✓
OptTyper [47]	100	LSTM	✗	✓	✓	✓	✓	✗
Typilus [12]	Unlimited	GNN	✓	✓	✗	✓	✓	✓
TypeBert [29]	40K	BERT	✓	✓	✗	✓	✓	✓

[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 representations, 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 $E_c : w_1, \ldots, w_l \rightarrow \mathbb{R}^{l \times d}$ for both code context and identifier tokens, where w_i 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.

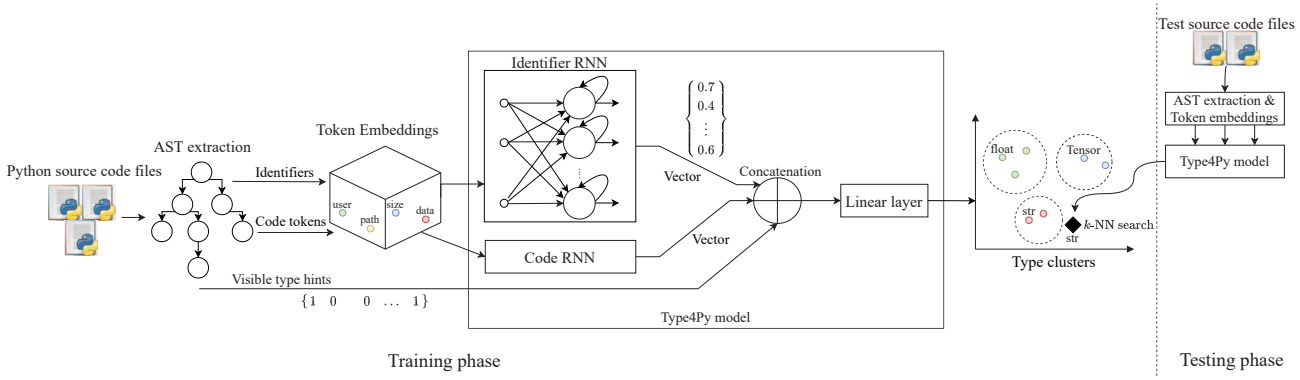

Figure 1: Overview of Type4Py approach

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[2]. 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 $h_i^{(t)}$ of the i-th LSTM unit at the time step t is defined as follows:

$$h_i^{(t)} = \tanh(s_i^t)\,\sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right) \quad (1)$$

which has sigmoid function σ, current input vector x_j, unit state s_i^t 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 *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. `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:

$$L(t_a, t_p, t_n) = max(0, m + \|t_a - t_p\| - \|t_a - t_n\|) \quad (2)$$

[2]The separator is a vector of ones with appropriate dimension.

The goal of the objective function L is to make t_a examples closer to the similar examples t_p than to t_n examples. We use the Euclidean metric to measure the distance of t_a with t_p and t_n.

At prediction time, we first map a query example t_q to the type clusters. The query example t_q can be a function's argument, the return type of a function or a variable. Then we find the k-nearest neighbor (KNN) [16] of the query example t_q. Given the k-nearest examples t_i with a distance d_i from the query example t_q, the probability of t_q having a type t' can be obtained as follows:

$$P(t_q : t') = \frac{1}{N} \sum_{i}^{k} \frac{\mathbb{I}(t_i = t')}{(d_i + \varepsilon)^2} \tag{3}$$

where $\mathbb{I}$ is the indicator function, N is a normalizing constant, and ε 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-duplicate 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:

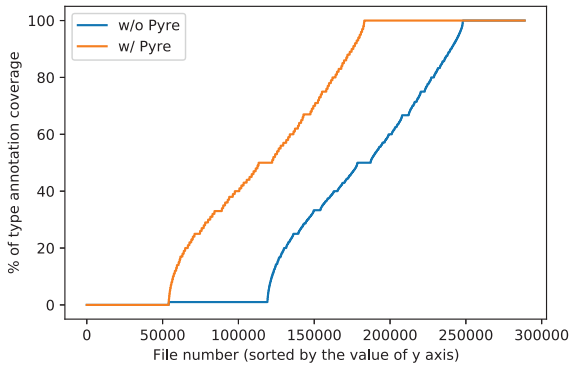

Figure 2: The effect of using Pyre on the type annotation coverage of source code files

- 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

Metrics[a,b]	Our dataset
Repositories	5,092
Files	201,613
Lines of code[c]	11.9M
Functions	882,657
...with return type annotations	94,433 (10.7%)
Arguments	1,558,566
...with type annotations	128,363 (14.5%)
Variables	2,135,361
...with type annotations	1,023,328 (47.9%)
Types	1,246,124
...unique	60,333

[a] Metrics are counted after the ASTs extraction phase of our pipeline.
[c] Comments and blank lines are ignored when counting lines of code.

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

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

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.

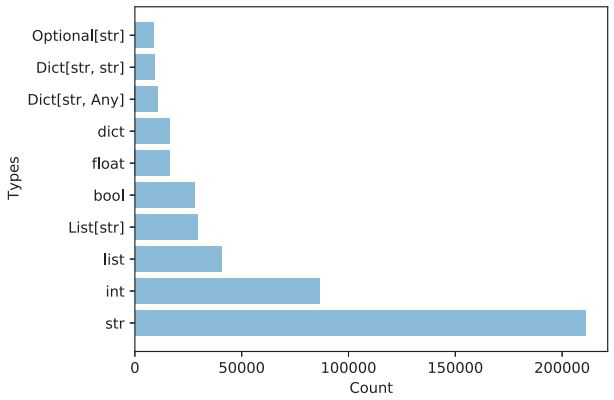

Figure 3: Top 10 most frequent types (`Any` and `None` types are excluded)

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 work 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.

Table 4: Value of hyperparameters for neural models

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

[a] The model stopped at epoch 38 due to the early stopping technique.

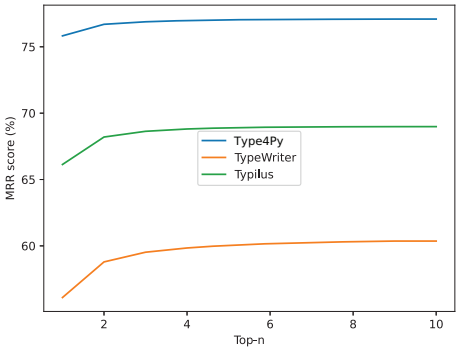

Figure 4: The MRR score of the models considering different top-n predictions

model and hence it has to predict the output of three data points for every single training batch (see Eq. 2). Typilus completes a single training epoch in around 6 minutes[4]. For all the neural models, the validation set is used 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 of an approach 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:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{r_i} \tag{4}$$

The MRR metric partially rewards the neural models by giving a score of $\frac{1}{r_i}$ 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.

[4]The public implementation of Typilus does not take advantage of our two GPUs.

6 Evaluation

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

RQ_1 What is the general type prediction performance of Type4Py?

RQ_2 How does Type4Py perform while considering different predictions tasks?

RQ_3 How do each proposed type hint and the size of type vocabulary contribute to the performance of Type4Py?

6.1 Type Prediction Performance (RQ_1)

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 5: Performance evaluation of the neural models considering different top-*n* predictions

Top-*n* predictions	Approach	% Exact Match				% Base Type Match[a]		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1	Type4Py	**75.8**	**100.0**	**82.3**	19.2	**80.6**	**85.2**	36.0
	Typilus	66.1	92.5	73.4	**21.6**	74.2	81.6	**41.7**
	TypeWriter	56.1	93.5	60.9	16.2	58.3	64.4	19.9
Top-3	Type4Py	**78.1**	**100.0**	**87.3**	23.4	**83.8**	**90.6**	43.2
	Typilus	71.6	96.2	83.0	**26.8**	79.8	88.7	**49.2**
	TypeWriter	63.7	98.8	79.2	20.8	67.3	83.5	27.9
Top-5	Type4Py	**78.7**	**100.0**	**88.6**	24.5	**84.7**	**92.1**	45.5
	Typilus	72.7	96.7	85.1	**28.2**	80.9	90.1	**51.0**
	TypeWriter	65.9	99.6	84.9	23.0	70.4	89.1	32.1
Top-10	Type4Py	**79.2**	**100.0**	89.7	25.2	**85.4**	**93.3**	46.9
	Typilus	73.3	97.04	86.4	**28.9**	81.5	90.9	**51.9**
	TypeWriter	68.2	99.9	**90.8**	25.5	73.2	93.8	36.5
MRR@10	Type4Py	**77.1**	**100.0**	**85.1**	21.4	**74.1**	**79.9**	29.4
	Typilus	69.0	94.4	78.5	**24.4**	67.4	75.8	**32.8**
	TypeWriter	60.4	96.1	71.3	19.1	56.5	68.0	19.7

[a] Ubiquitous types are not a base type match. However, they are considered in the All column.

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 (RQ_2)

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

Method: Similar to the RQ_1, 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 (RQ_3)

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

Metric	Task	Approach	% Exact Match				% Base Type Match		
			All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1 prediction	Argument	Type4Py	**61.9**	**100.0**	**64.5**	17.4	**63.9**	**69.3**	20.1
		Typilus	53.8	83.3	46.6	**23.7**	57.0	52.5	**29.6**
		TypeWriter	58.4	93.6	61.3	19.6	60.1	64.4	22.1
	Return	Type4Py	**56.4**	**100.0**	59.3	**14.4**	**60.3**	**65.4**	20.9
		Typilus	42.5	84.0	41.6	12.3	49.9	49.5	**24.8**
		TypeWriter	50.7	93.3	**59.9**	9.2	54.1	64.4	15.0
	Variable[a]	Type4Py	**80.4**	**100.0**	**86.8**	20.7	**85.9**	**89.1**	44.6
		Typilus	71.4	95.1	80.5	**22.5**	80.7	**89.1**	**48.6**
MRR@10	Argument	Type4Py	**64.2**	**100.0**	69.5	20.7	**59.9**	62.2	20.6
		Typilus	58.7	87.9	55.4	**27.5**	56.0	52.2	**28.1**
		TypeWriter	63.3	96.2	**72.4**	23.0	59.6	**69.3**	22.7
	Return	Type4Py	**57.9**	**100.0**	63.3	**16.1**	**52.9**	55.8	18.5
		Typilus	46.0	86.9	49.8	14.3	44.9	46.6	**21.4**
		TypeWriter	54.2	95.9	**68.9**	10.9	49.9	**65.1**	14.2
	Variable[a]	Type4Py	**81.4**	**100.0**	**89.1**	22.7	**79.1**	**85.0**	34.1
		Typilus	73.7	96.3	84.7	**25.1**	72.4	82.7	**36.1**

[a] Note that TypeWriter cannot predict the type of variables.

Table 7: Performance evaluation of Type4Py with different configurations

Metric	Approach	% Exact Match				% Base Type Match		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1 prediction	Type4Py	**75.8**	**100.0**	82.3	**19.2**	**80.6**	85.2	**36.0**
	Type4Py (w/o identifiers)	72.7	**100.0**	71.8	17.4	76.5	73.9	30.9
	Type4Py (w/o code context)	67.9	**100.0**	59.2	11.4	70.6	63.3	17.9
	Type4Py (w/o visible type hints)	65.4	86.2	71.9	15.8	70.0	74.9	31.5
	Type4Py (w/ top 1,000 types)	74.5	**100.0**	**83.3**	12.9	79.1	**86.3**	28.5
MRR@10	Type4Py	**77.1**	**100.0**	85.1	**21.4**	**74.1**	79.9	**29.4**
	Type4Py (w/o identifiers)	73.8	**100.0**	74.6	19.2	69.3	66.6	25.1
	Type4Py (w/o code context)	69.7	**100.0**	63.9	13.6	63.8	55.4	17.7
	Type4Py (w/o visible type hints)	68.6	89.3	76.2	18.2	65.8	70.1	26.2
	Type4Py (w/ top 1,000 types)	75.6	**100.0**	**86.2**	14.2	72.4	**81.7**	22.8

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

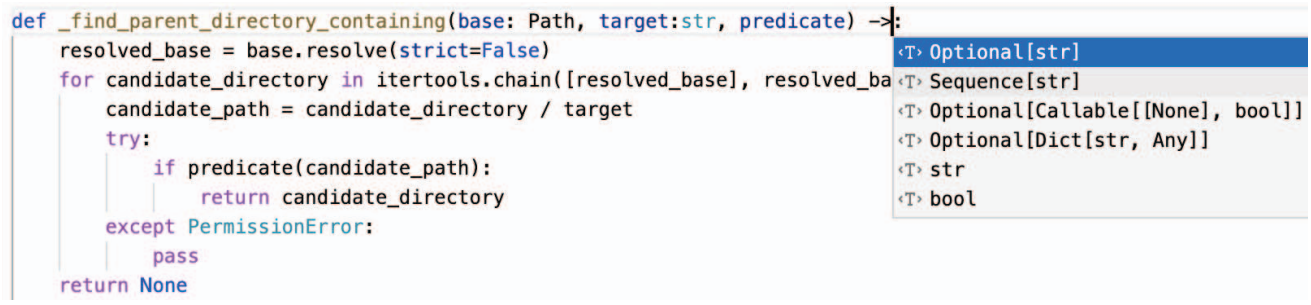

Figure 5: A type auto-completion example from VSC. The code has not seen during training. The expected return type is `Optional[str]`.

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 (RQ_1), 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 RQ_1 and RQ_2, 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 (RQ_3), 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 the 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

[1] [n.d.]. Annoy. https://github.com/spotify/annoy. Accessed on: 2022-02-08.
[2] [n.d.]. CD4Py: Code De-Duplication for Python. https://github.com/saltudelft/CD4Py. Accessed on: 2022-02-07.
[3] [n.d.]. IEEE Spectrum's the Top Programming Languages 2021. https://spectrum.ieee.org/top-programming-languages. Accessed on: 2022-02-07.
[4] [n.d.]. LibSA4Py: Light-weight static analysis for extracting type hints and features. https://github.com/saltudelft/libsa4py. Accessed on: 2022-02-08.
[5] [n.d.]. Mypy: A static type checker for Python 3. https://mypy.readthedocs.io/. Accessed on: 2022-02-07.
[6] [n.d.]. Pyre: A performant type-checker for Python 3. https://pyre-check.org/. Accessed on: 2022-02-07.
[7] [n.d.]. PyRight. https://github.com/microsoft/pyright. Accessed on: 2022-02-07.
[8] [n.d.]. PyType. https://github.com/google/pytype. Accessed on: 2022-02-07.
[9] [n.d.]. Type4Py's Visual Studio Code extension. https://marketplace.visualstudio.com/items?itemName=saltud.type4py. Accessed on: 2022-02-08.
[10] [n.d.]. Typilus' public implementation. https://github.com/typilus/typilus. Accessed on: 2022-02-08.
[11] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.
[12] 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*. 91–105.
[13] Uri Alon and Eran Yahav. 2020. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *International Conference on Learning Representations*.
[14] De Cheng, Yihong Gong, Sanping Zhou, Jinjun Wang, and Nanning Zheng. 2016. Person re-identification by multi-channel parts-based cnn with improved triplet loss function. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1335–1344.
[15] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 1. IEEE, 539–546.
[16] Thomas Cover and Peter Hart. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.
[17] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/.
[18] Yong Du, Wei Wang, and Liang Wang. 2015. Hierarchical recurrent neural network for skeleton based action recognition. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1110–1118.
[19] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. 2009. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*. 1859–1866.
[20] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769.
[21] Luís PF Garcia, André CPLF de Carvalho, and Ana C Lorena. 2015. Effect of label noise in the complexity of classification problems. *Neurocomputing* 160 (2015), 108–119.
[22] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.
[23] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (2014), 1335–1382.
[24] 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.
[25] Xincheng He, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. 2021. PyART: Python API Recommendation in Real-Time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1634–1645.
[26] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 152–162.
[27] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.
[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[29] 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*. 1483–1486.
[30] Daniel Jurafsky and James H. Martin. 2009. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., USA.
[31] Faizan Khan, Boqi Chen, Daniel Varro, and Shane Mcintosh. 2021. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* (2021).
[32] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[33] J Lehtosalo et al. 2017. Mypy-optional static typing for python.
[34] Wentong Liao, Michael Ying Yang, Ni Zhan, and Bodo Rosenhahn. 2017. Triplet-based deep similarity learning for person re-identification. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*. 385–393.
[35] Fagui Liu, Lailei Zheng, and Jingzhong Zheng. 2020. HieNN-DWE: A hierarchical neural network with dynamic word embeddings for document level sentiment classification. *Neurocomputing* 403 (2020), 21–32.
[36] Xiaoyu Liu, LiGuo Huang, and Vincent Ng. 2018. Effective API recommendation without historical software repositories. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 282–292.
[37] Edward Loper and Steven Bird. 2002. NLTK: The Natural Language Toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for*

Teaching Natural Language Processing and Computational Linguistics. 63–70.
[38] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.
[39] Magnus Madsen. 2015. *Static analysis of dynamic languages*. Ph.D. Dissertation. Aarhus University.
[40] Eva Maia, Nelma Moreira, and Rogério Reis. 2012. A static type inference for python. *Proceedings of the 6th Workshop on Dynamic Languages and Applications* 5, 1 (2012), 1.
[41] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.
[42] Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Cambridge university press.
[43] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
[44] Nevena Milojkovic, Mohammad Ghafari, and Oscar Nierstrasz. 2017. Exploiting type hints in method argument names to improve lightweight type inference. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 77–87.
[45] Amir M. Mir, Evaldas Latoskinas, and Georgios 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. https://doi.org/10.1109/MSR52588.2021.00079
[46] 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.
[47] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. 2020. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. *arXiv preprint arXiv:2004.00348* (2020).
[48] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.
[49] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.
[50] Zvonimir Pavlinovic. 2019. *Leveraging Program Analysis for Type Inference*. Ph.D. Dissertation. New York University.
[51] 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*. 209–220.
[52] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 types in the wild: a tale of two type systems. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. 57–70.
[53] Guozheng Rao, Weihang Huang, Zhiyong Feng, and Qiong Cong. 2018. LSTM with sentence representations for document-level sentiment classification. *Neurocomputing* 308 (2018), 49–57.
[54] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 155–165.
[55] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.
[56] Michael Salib. 2004. Faster than C: Static type inference with Starkiller. *in PyCon Proceedings, Washington DC* (2004), 2–26.
[57] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.
[58] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
[59] Andreas Stuchlik and Stefan Hanenberg. 2011. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*. 97–106.
[60] Guido Van Rossum, Jukka Lehtosalo, and Lukasz Langa. 2014. PEP 484–type hints. *Index of Python Enhancement Proposals* (2014).
[61] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Advances in neural information processing systems*. 2692–2700.
[62] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2019. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*.
[63] Ronald J Williams and David Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural computation* 1, 2 (1989), 270–280.
[64] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 607–618.
[65] Jianming Zheng, Fei Cai, Wanyu Chen, Chong Feng, and Honghui Chen. 2019. Hierarchical neural representation for document classification. *Cognitive Computation* 11, 2 (2019), 317–327.
[66] Zhi-Hua Zhou. 2018. A brief introduction to weakly supervised learning. *National science review* 5, 1 (2018), 44–53.