Type4Py: Deep Similarity Learning-Based Type Inference for Python

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

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

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

Georgios Gousios
Department of Software Technology
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

***Abstract*—Dynamic languages, such as Python and Javascript, trade static typing for developer flexibility. While this allegedly enables greater productivity, lack of static typing can cause run-time exceptions, type inconsistencies, 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, learning-based approaches have been proposed to enable automatic type annotations based on existing, partially annotated codebases. However, the prediction of rare and user-defined types is still challenging. In this paper, we present Type4Py, a deep similarity learning-based type inference model for Python. We design a hierarchical neural network model that learns to discriminate between types of the same kind and dissimilar types in a high-dimensional space, which results in clusters of types. Nearest neighbor search suggests likely type signatures of given Python functions. The types visible to analyzed modules are surfaced using lightweight dependency analysis. The results of quantitative and qualitative evaluation indicate that Type4Py significantly outperforms state-of-the-art approaches at the type prediction task. Considering the Top-1 prediction, Type4Py obtains 19.33% and 13.49% higher precision than Typilus and TypeWriter, respectively, while utilizing a much bigger vocabulary.**

***Index Terms*—Type Inference, Deep Similarity Learning, Machine Learning, Dynamic Language, Python**

I. Introduction

Over the past few years, dynamically-typed programming languages have become extremely popular among software developers. According to the IEEE Spectrum ranking [1], Python was the most popular programming language in 2019. Dynamic languages such as Python and JavaScript allow rapid prototyping which potentially reduces development time [2], [3]. However, the lack of static types often leads to type errors, unexpected run-time errors, and suboptimal IDE support. On the other hand, statically-typed languages are generally less error-prone [4] and static typing improves software quality [5] and the maintainbility of software systems in terms of understanding undocumented code, fixing type errors [2] and early bug detection [5].

To mitigate static type-related issues of dynamic languages, the Python community introduced PEP 484 [6] which enables developers to add optional type annotations to their Python codebase. The optional static typing is a manual, cumbersome, and error-prone process [7]. Static type inference methods [8], [9] can be employed to infer type annotations. Aside from the dynamic features of the mentioned languages, static analysis over-approximates programs' behavior [10]. Thus static type inference methods are often not precise [11]. Moreover, these methods typically analyze a full program and its dependencies which makes them quite slow for the type inference task.

Motivated by the above limitations of static type inference methods, researchers have recently employed Machine Learning (ML) techniques to predict type annotations for dynamic languages [12]–[15]. 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 [14], [15]. Despite the superiority of ML-based type prediction approaches, their type vocabulary is small and fixed-sized. This limits their type prediction ability for user-defined and rare types. To solve this issue, Allamanis et al. [15] recently introduced the Typilus model that does not constraint the type vocabulary size. Although the Typilus model outperforms the other models with closed type vocabulary, the type prediction task is still quite challenging, especially for rare types.

In this paper, we present Type4Py model, a deep similarity learning-based type inference for Python. By formulating ML-based type inference as a Deep Similarity Learning (DSL) problem, we propose an effective hierarchical neural network that maps programs into a high-dimensional feature space, known as *type clusters*. Similarity learning has been used in Computer Vision, for example, to discriminate a person's face from faces of others for verification [16]. Similarly, Type4Py learns how to distinguish between different types through a hierarchical neural network. As a result, our proposed approach can handle a very large type vocabulary, which in

turn enables support for the inference of user-defined and rare types. The evaluation results also indicate that our proposed approach outperforms state-of-the-art ML-based type inference models both quantitatively and qualitatively. Considering F1-score of top-1 prediction, Type4Py obtains 18.37% higher score than Typilus.

Overall, this paper presents the following main contributions:

- *Type4Py*, a DSL-based type inference approach based on hierarchical neural network and type clusters, which handles very large type vocabulary.
- A large dataset of 4,910 open-source Python projects that contains more than 1.1M type annotations.

We publicly released the reproduction package of the Type4Py model on GitHub[1].

II. Related Work

Type checking and inference for Python: In 2014, the Python community introduced a type hints proposal [6] 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 [17]. 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* [18], *PyRight* [19], and *Pyre* [20].

Moreover, a number of research work proposed a type inference algorithm for Python [8], [21], [22]. 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 [11], due to the dynamic nature of Python and the over-approximation of programs' behavior by static analysis [10].

Learning-based type inference: In 2015, Rachev et al. [23] 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. [24] 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 dataflow between variables. Although the probabilistic model of Xu et al. [24] outperforms static type inference systems, their proposed system is slow and lacks scalability.

Considering the mentioned issue of JSNice, Hellendoorn et al. [12] 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. [13] 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 [13].

Motivated by the NL2Type model, Pradel et al. [14] proposed 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. To overcome the mentioned drawback of DeepTyper, Wei et al. [25] 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. The experimental results of Wei et al. [25] show the superiority of LAMBDANET over DeepTyper, though LAMBDANET's prediction ability is limited to top-100 most common types.

Given that the natural constraints such as identifiers and comments are an uncertain source of information, Pandi et al. [26] proposed OptTyper which predicts types for 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 [26].

So far, all the discussed learning-based type inference methods employ a vocabulary of maximum 1,000 types. This hinders their ability to infer user-defined and rare types. Probably the most related work to this paper is Typilus, which has been proposed by Allamanis et al. [15]. Typilus is a graph-based deep neural network 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. The differences between Type4Py and other learning-based approaches are summarized in Table I.

Among all the described approaches here, we chose TypeWriter and Typilus as a baseline. Since TypeWriter is an HNN-based model and outperforms NL2Type and DeepTyper. Moreover, Typilus has no limit on the size of type vocabulary and is the current state-of-the-art approach in ML-based type inference. LAMBDANET and OptTyper were not selected as a baseline as their quite small type vocabulary make the comparison with Typilus and Type4Py futile.

[1]https://github.com/saltudelft/type4py

TABLE I: Comparison between Type4Py and other learning-based type inference approaches

Approach	Size of type vocabulary	ML model	Type hints		
			Contexual	Natural	Logical
Type4Py	Unlimited	HNN (2x RNNs)	✓	✓	✗
JSNice [23]	10+	CRFs	✓	✓	✗
Xu et al. [24]	-	PGM	✗	✓	✓
DeepTyper [12]	10K+	biRNN	✓	✓	✗
NL2Type [13]	1K	LSTM	✗	✓	✗
TypeWriter [14]	1K	HNN (3x RNNs)	✓	✓	✗
LAMBDANET [25]	100	GNN	✓	✓	✓
OptTyper [26]	100	LSTM	✗	✓	✓
Typilus [15]	Unlimited	GNN	✓	✓	✗

III. 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.

A. Type hints

We extract the Abstract Syntax Tree (AST) from Python source code. From ASTs, we obtain type hints that are valuable for predicting types of function arguments and return types. Similar to the previous work [13], [14], the obtained type hints are based on code contexts and natural information, which are described in this section.

Natural Information: As indicated by the previous work [13], source code contains useful and informal natural language information that is considered as a source of type hints. Identifier names and code comments are the two main sources of natural information. 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. We also denote a function's argument as N_{arg} hereafter. Different from the previous work [13], [14], we do not use code comments as a type hint as they are shown to be ineffective for the type prediction task in both the experiment of Pradel et al. [14] and ours. After extracting the described natural language information, we preprocess them by applying common Natural Language Processing (NLP) techniques. This preprocessing step involves tokenization, stop word removal, and lemmatization [27].

Code Context: Similar to TypeWriter [14], 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. 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: In contrast to previous work that only analyzed the direct imports [14], 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 study shows that visible type hints substantially improve the performance of both Type4Py and TypeWriter models (subsection V-C).

B. 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 semantic similarities, a word embedding technique is used to map words into a d-dimensional vector space, $\mathbb{R}^d$. Specifically, we employ Word2Vec [28] embeddings to train a code embedding $E_c : w_1, \ldots, w_n \rightarrow \mathbb{R}^{l \times d}$ for both code context and identifier tokens, where w_i and n denote a single token and the length of a sequence, respectively. In the following, we describe the vector representation of code contexts and identifiers 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:

$$E_c(N_f) \circ s \circ E_c(N_{args})$$

Code contexts: For a function argument, we concatenate the sequences of the argument's usages in the function body 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 both argument and return types, we create a 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.

C. Neural model

The neural model of our proposed approach employs a hierarchical neural network (HNN), which consists of two recurrent neural networks (RNNs) [29]. HNNs are well-studied and quite effective for text and vision-related tasks [30]–[32]. In

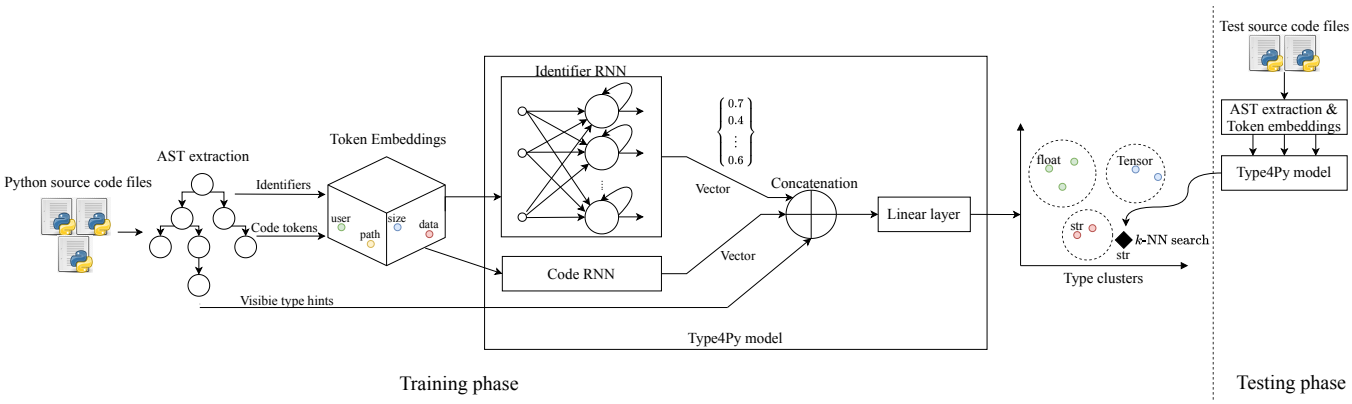


Fig. 1: Overview of Type4Py approach

the case of type prediction, intuitively, HNNs can capture different aspects of identifiers and code context. In the neural architecture, the two RNNs are based on long short-term memory (LSTM) units [33]. Here, we chose LSTMs units as they are effective for capturing long-range dependencies [34]. Also, LSTM-based neural models have been applied successfully to NLP tasks such as sentiment classification [35]. 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 [34]. 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 [13], [14], 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 [16], [36]. By using DSL formulation, our neural model learns to map argument and return types into a real continuous space, called *type clusters* (also known as type space in [15]). 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 type clusters. Unlike the previous work [13], [14], our proposed model is more effective in predicting rare and user-defined types, since it can handle a very large type vocabulary.

To create the described type clusters, we use *Triplet loss* [37] function which is recently used for computer vision tasks such as face recognition [37]. 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 of the same as t_a, and a type t_n which is different than t_a. Therefore, given a positive scalar margin m, the Triplet loss function is defined as follows:

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

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 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 or the return type of a function. Then we find the k-neareast neighbor (KNN) [38] of the query example t_q. Given the k-nearest examples t_i with a distance di from the query example t_q, the probability of t_q having a type t' can be obtained as follows:

$$P(t_q : t') = \sum_i^k \frac{\mathbb{I}(t_i = t')}{(d_i + \varepsilon)^2} \quad (3)$$

where $\mathbb{I}$ is the indicator function and ε is a small scalar (i.e. $\varepsilon = 10^{-10}$).

IV. Evaluation Setup

In this section, we first describe the creation and characteristics of datasets, and code de-duplication procedure. Afterwards, we introduce the implementation details of our proposed approach. Lastly, we describe evaluation metrics to quantitatively measure the performance of type inference approaches.

A. Dataset

A useful dataset maximizes the amount of projects that include type annotations. As such, we have used the popular web-service libraries.io to find Python packages that depend on the `mypy` package, the official and most popular type checker for Python. We believe that these packages are more likely to have type annotations. The search results consists of 4,910

TABLE II: Characteristics of the datasets used for evaluation

Metrics[a]	Type4Py's dataset		Typilus' dataset [15]	
	Original	De-duplicated	Original	De-duplicated
Repositories	4,910	4,861	592	591
Files	428,917	168,407	167,498	101,755
Lines of code[b]	26.4M	17.8M	21.8M	15.4M
Functions	5,459,266	1,908,338	2,062,976	1,209,385
...with comment	2,419,461 (44.3%)	612,277 (32.1%)	539,920 (26.2%)	341,860 (28.3%)
...with return type annotations	478,972 (8.8%)	256,893 (13.5%)	75,896 (3.7%)	62,869 (5.2%)
...with both	201,998 (3.7%)	92,946 (4.9%)	22,083 (1.1%)	20,234 (1.7%)
Arguments	9,926,874	3,444,473	3,831,902	2,235,714
...with type annotations	651,254 (6.6%)	356,439 (10.3%)	93,704 (2.4%)	85,115 (3.8%)
Types	1,130,226	613,332	169,600	147,984
...unique	47,349	45,883	12,870	12,595

[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 III: Number of each datapoints for train, validation and test sets

	Type4Py's dataset	
	Argument type	Return type
Training	246,715	103,035
Validation	29,549	11,417
Test	69,749	28,136
Total	342,013 (70.6%)	142,588 (29.4%)

	Typilus' dataset [15]	
	Argument type	Return type
Training	60,574	23,227
Validation	5,647	2,243
Test	16,964	6,349
Total	83,185 (72.3%)	31,819 (27.7%)

Python projects, which we have cloned for further analysis. In addition, we have used the dataset of Allamanis et al. [15], which allows us to explore Type4Py's efficiency on a smaller dataset.

Code De-Duplication: Recently, Allamanis [39] found that code duplication has a negative effect on the performance of machine learning models when evaluating on unseen code samples. The study shows that code de-duplication is important for indicating the practicality of machine learning-based tools. To de-duplicate both datasets, we use term frequency-inverse document (TF-IDF) [40] to represent a source code file as a vector in $\mathbb{R}^d$. Next, we employ k-nearest neighbor search to find k candidate duplicates of a file. To filter out the candidate duplicates, we apply the threshold t to the distance between a file and its candidate duplicates. After obtaining the duplicates of each file in the dataset, we find clusters of duplicate files while assuming that the similarity is transitive [39]. Basically, a cluster contains similar duplicate files. Finally, we keep a file from each cluster and remove all other identified duplicate files from the dataset. In our experiments, we set $d = 4096$, $k = 5$, $t = 0.95$. Our code de-duplication tool is publicly available on GitHub[2].

Dataset Characteristics: Table II shows the characteristics of the two datasets before and after removing duplicate source code files. When compared to the Typilus' dataset, Type4Py's dataset includes a much larger sample of repositories (4,910 vs. 591) and more than four times the amount of type annotations.

Figure 2 shows the frequency of top 10 most frequent types in Type4Py's dataset. It can be observed that types follow a long-tail distribution. Unsurprisingly, the top 10 most frequent types amount to 45.4% of types in Type4Py's dataset. The same distribution was observed for the Typilus' dataset [15]. Lastly, we randomly split the two datasets by files into three sets: 70% training data, 10% validation data and 20% test data. Table III shows the number of data points for each of the three sets.

Pre-processing: 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 straightforward to predict, i.e., `__len__` always returns `int`, and would blur the results.
- We exclude parameters and return types with `Any` or `None` type annotations as it is not helpful to predict these types and they can be inferred statically.
- We perform 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`.

B. Implementation details and environment setup

We implemented the Type4Py and TypeWriter approaches in Python 3 and its ecosystem. We extract ASTs from source files using the *ast* module of Python's standard library. The data processing pipeline is parallelized by employing *joblib* package. We use the NLTK package [41] for performing

[2]https://github.com/saltudelft/CD4Py

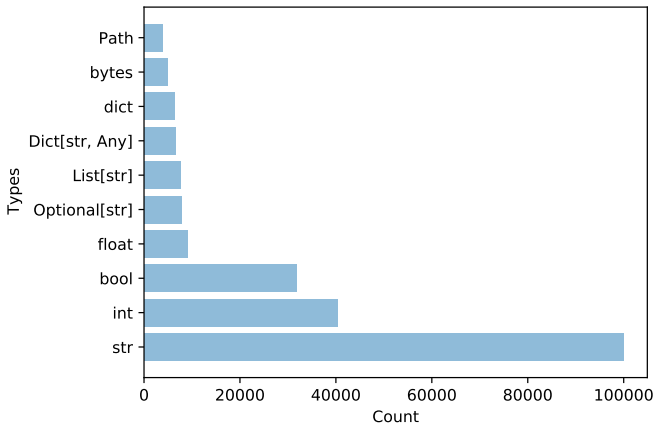


Fig. 2: Top 10 most frequent types

TABLE IV: Value of hyperparameters for neural models

Hyperparameter	Type4Py	TypeWriter	Typilus
Word embedding dimension	100	100	-
LSTM's hidden nodes	512	512	-
GNN's hidden nodes	-	-	64
Dimension of linear layer's output	4096	1000	-
Number of LSTM's layers	1	1	-
Learning rate	0.002	0.002	0.00025
Dropout rate	0.25	0.25	0.1
Number of epochs	15	25	500
Batch size	2536	4096	-
Value of k for nearest neighbor search	10	-	10
Tripet loss' margin value (i.e. m)	2.0	-	2.0
Model's trainable parameters	17.6M	11.6M	650K

standard NLP tasks such as tokenization and stop work removal. To train the Word2Vec model, gensim package is used. The neural model is implemented in the PyTorch framework [42]. For splitting the datasets and evaluating neural models' performance, we use the Scikit-learn package [43]. Lastly, we use Annoy [44] package to perform a fast and approximate nearest neighbor search. More specifically, we used bidirectional LSTMs [45] in PyTorch framework to implement the two RNNs of the Type4Py's neural model. To avoid overfitting the train set, we applied the Dropout regularization [46] to the two input sequences except for the visible types. Also, we employed Adam optimizer [47] to minimize the value of the Tripletloss function. The value of the neural models' hyperparameters is reported in Table IV. For Typilus, we used its public implementation on GitHub [48].

We performed all the experiments on Linux (Ubuntu 18.04.4 LTS). The computer had an Intel Xeon CPU E5-2690 v4 with 24 cores (@2.6GHz), 128 GB of RAM, and two NVIDIA GeForce GTX 1080 TIs. We employed the data parallelism feature of the PyTorch framework to distribute training batches between the two GPUs with the total VRAM of 22 GB.

C. 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. [15]:

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

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

In addition to these two criteria, we calculate weighted F1-score, precision, and recall for the exact matches. Furthermore, we consider types that we have seen more than 100 times in the train set as *common* or *rare* otherwise. Unlike Type4Py and Typilus, TypeWriter predicts `other` 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.

V. 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 top-n predictions?
- RQ_3 How does each considered type hint contribute to the performance of Type4Py?
- RQ_4 How effective is Type4Py in terms of training and inference speed?
- RQ_5 How does Type4Py compare qualitatively to the other approaches?

A. Type Prediction Performance (RQ_1)

In this section, we compare our proposed approach, Type4Py, with the state-of-the-art approaches TypeWriter and Typilus in terms of type prediction performance.

Methodology: The comparison is based on both datasets of Type4Py and Typilus. The models get trained on the training sets and the test sets are used to measure the type prediction performance. We consider three inference tasks: argument type, return type, and the combination of both. We differentiate between the two datasets, the three prediction tasks, the three inference approaches (Type4Py, TypeWriter, and Typilus), and between.

Results: Table V indicates the type predication performance of the approaches. Considering the exact match criteria (all types), Type4Py outperforms both Typilus and TypeWriter in all tasks. For instance, Type4Py performs better than Typilus and TypeWriter at the prediction rare argument types by a margin of 16.3% and 14.9%, respectively. More importantly, our Type4Py model gives more relevant predictions for all type prediction tasks based on the precision metric. Given the results of return type predictions on Type4Py's dataset, our proposed approach obtains higher precision than Typilus and TypeWriter by a margin of 10.4% and 17.3%, respectively. Since TypeWriter is limited to a type vocabulary size of 1000, it is substantially weaker in rare type prediction (especially rare return types) than the other two models. However, TypeWriter's limited type vocabulary results in a slightly better

TABLE V: Performance evaluation of neural type inference models on Type4Py's dataset considering top-10 predictions.

Dataset	Task	Approach	% Exact Match			% Parametric Type Match			% Weighted Metrics		
			All	Common	Rare	All	Common	Rare	F1-score	Recall	Precision
Our dataset	Combined prediction	Type4Py	**72.07**	92.17	**38.91**	**76.74**	94.27	**47.84**	**69.42**	**72.07**	**69.17**
		TypeWriter	68.82	**95.88**	24.17	74.57	**94.60**	30.94	61.20	68.82	57.06
		Typilus	59.19	85.40	26.59	69.29	88.60	45.27	55.74	59.19	54.95
	Argument prediction	Type4Py	**75.73**	93.81	**46.25**	**79.30**	95.50	**52.91**	**73.22**	**75.73**	**72.77**
		TypeWriter	71.73	**96.55**	31.27	75.85	**97.58**	40.44	63.65	71.73	58.71
		Typilus	62.27	86.46	29.88	70.37	89.41	44.89	59.00	62.27	58.27
	Return prediction	Type4Py	**60.30**	91.35	**24.47**	**68.31**	94.49	38.11	**57.31**	**60.30**	**56.91**
		TypeWriter	52.58	**92.40**	6.64	61.60	**95.08**	22.98	41.14	52.58	39.32
		Typilus	51.85	82.56	19.72	66.71	86.46	**46.06**	47.81	51.85	46.52
Typilus' dataset	Combined prediction	Type4Py	**67.20**	93.16	**33.72**	**72.16**	95.37	42.24	**64.49**	**67.20**	**64.89**
		TypeWriter	67.06	**95.55**	30.32	71.63	**96.70**	39.32	58.76	67.06	55.05
		Typilus	63.40	88.21	32.79	72.21	90.39	**49.77**	60.18	63.39	59.43
	Argument prediction	Type4Py	**71.32**	95.16	**41.78**	**74.71**	96.50	47.73	**68.99**	**71.32**	**69.05**
		TypeWriter	68.32	**96.27**	33.71	71.84	**97.09**	40.56	60.44	68.32	57.26
		Typilus	67.61	89.62	38.51	73.73	91.43	**50.34**	64.77	67.60	64.62
	Return prediction	Type4Py	**55.71**	93.59	19.62	64.50	96.31	34.19	**52.88**	**55.71**	**52.40**
		TypeWriter	47.01	**95.45**	0.86	56.57	**97.44**	17.63	34.07	47.01	30.54
		Typilus	52.33	84.03	**19.64**	**68.19**	87.34	**48.44**	48.13	52.33	46.36

performance in the prediction of common types comparing to Type4Py.

To further analyze the effectiveness of our proposed approach, we performed experiments on a smaller dataset from the work of Allamanis et al. [15]. The performance evaluation of the approaches on Typilus' dataset is shown at the bottom half of Table V. Considering the results of all tasks on Typilus' dataset, the performance difference between Type4Py and Typilus is less significant than the obtained results on Type4Py's dataset. We attribute this mainly to the fact that Typilus' dataset is much smaller than Type4Py's dataset (see Table III). On the other hand, when our Type4Py model trained on a much larger dataset, it generalizes better and significantly outperforms Typilus both at the prediction of common and rare types. Based on the obtained results on Typilus' dataset and ours, it appears that Type4Py's DSL-based hierarchical neural network has a better learning capacity than that of Typilus' GNN model for the type prediction task, when trained on a significantly larger dataset. In other words, The performance of Type4Py is significantly improved on the larger dataset whereas Typilus' performance is decreased.

B. Top-n predictions performance (RQ_2)

Research has shown that developers are more likely to use the first suggestion by a tool [49]. It is therefore important that an ML-based type inference approach shows satisfying results in the top-n predictions, especially the top-1 prediction. To evaluate RQ_2, we consider the combined prediction task on Type4Py's dataset.

Results: Table VI shows performance evaluation of Type4Py, TypeWriter and Typilus while considering top-n predictions. Overall, we observe that the Type4Py model outperforms both Typilus and TypeWriter in all considered top-n predictions. As can be seen from Table VI, there is a negligible difference between the top-5 and the top-10 predictions. This suggests that the list of top-3 predictions may often contain the correct type, assuming that the model can correctly predict a given argument type or the return type of a function. When comparing the top-1 and the top-10 predictions, the performance of Type4Py decreases significantly whereas the performance loss for Typilus is more significant (i.e. 9.9% decrease in the precision score). Also, the results of top-1 prediction show that the first predicted type may not always be correct with the assumption that the expected type exists in the top-3 or the top-5 predictions. In summary, the ability of the Type4Py model to make a correct prediction is quite better than that of TypeWriter and Typilus' as the Type4Py's precision score is significantly higher than both models by a margin of 13.49% and 19.33% in the top-1.

C. Ablation analysis (RQ_3)

We investigate the performance of Type4Py model with different configurations on its own dataset considering the combined prediction task. Table VII indicates the performance of Type4Py with/without visible type hints, code context and identifiers. It can be observed that visible type hints significantly improve the performance of Type4Py model. For instance, considering exact match criteria, the performance of Type4Py is substantially improved for both common and rare types by 12.99% and 4.68%, respectively. In contrast, Pradel et al. [14] indicated that visible type hints did not improve the performance of TypeWriter. We attribute this difference to the fact that we perform a deeper analysis for extracting visible type hints from source code files. Unlike the work of Pradel et al. [14], our analysis considers a set of visible types from the dependencies of a source code file and it also resolves the

TABLE VI: Performance evaluation of neural models on Type4Py's dataset considering top-n predictions

Top-n predictions	Approach	% Exact Match			% Parametric Type Match			% Weighted Metrics		
		All	Common	Rare	All	Common	Rare	F1-score	Recall	Precision
Top-1	Type4Py	**66.09**	**87.17**	**31.32**	**73.35**	92.38	**41.96**	**64.13**	**66.09**	**64.34**
	TypeWriter	60.43	85.79	18.61	69.92	**92.61**	32.25	54.04	60.43	50.85
	Typilus	48.60	71.99	19.51	55.75	75.04	31.75	45.76	48.60	45.01
Top-3	Type4Py	**70.17**	90.39	**36.81**	**75.87**	93.82	**46.25**	**67.81**	**70.17**	**67.72**
	TypeWriter	65.52	**92.18**	21.81	72.94	**96.10**	34.74	58.48	65.62	54.67
	Typilus	56.42	82.06	24.53	66.01	85.72	41.49	53.23	56.42	52.28
Top-5	Type4Py	**71.34**	91.47	**38.13**	**76.41**	94.10	**47.23**	**68.82**	**71.34**	**68.64**
	TypeWriter	67.22	**94.04**	22.98	73.69	**96.77**	35.62	59.85	67.22	55.87
	Typilus	58.17	84.17	25.84	68.16	87.67	43.91	54.86	58.17	54.15
Top-10	Type4Py	**72.07**	92.17	**38.91**	**76.74**	94.27	**47.84**	**69.42**	**72.07**	**69.17**
	TypeWriter	68.82	**95.88**	24.17	74.30	**97.23**	36.46	61.20	68.82	57.06
	Typilus	59.19	85.40	26.59	69.29	88.60	45.27	55.74	59.19	54.95

TABLE VII: Performance evaluation of Type4Py with different configurations on its own dataset

Approach	% Exact Match			% Parametric Type Match			% Weighted Metrics		
	All	Common	Rare	All	Common	Rare	F1-score	Recall	Precision
Type4Py	**72.07**	**92.17**	**38.91**	**76.74**	**94.27**	**47.84**	**69.42**	**72.07**	**69.17**
Type4Py (w/o visible type hints)	62.22	79.18	34.23	65.89	80.89	41.14	56.99	62.22	55.11
Type4Py (w/o code context)	69.99	90.46	36.22	74.85	93.01	44.89	68.26	69.99	68.76
Type4Py (w/o identifiers)	54.55	81.52	10.06	60.74	85.86	19.30	54.34	54.55	56.96

name aliasing for imported visible types (i.e. `import numpy as np`).

To observe the effect of code context type hints on the performance of the Type4Py model, we removed the input sequences of code context and its corresponding RNN. As shown in Table VII, the performance of the Type4Py model without code context type hints is slightly decreased in comparison with the complete model. In addition, we performed the same experiment for identifier type hints. Considering the exact match criteria, the overall performance of the model without identifiers is decreased by 17.52%, which is quite substantial. This suggests that identifier names chosen by developers may have a correlation or similarity with the type of an argument or a function's return value. In other words, identifier type hints aid the neural model the most to discriminate between similar and dissimilar types in the type cluster. Overall, we conclude that identifiers are the strongest type hint for our type prediction model.

D. Computational Efficiency (RQ_4)

We compare our proposed approach with TypeWriter and Typilus in terms of training and prediction speed on Type4Py's dataset considering the combined prediction task. For the Type4Py model, a single training epoch takes around 157 sec. whereas it takes 13 sec. for the TypeWriter model. This significant difference in the training speed can be explained by two reasons. First, the Type4Py model has substantially more trainable parameters than that of TypeWriter (see Table IV). Second, Type4Py is a DSL-based model and hence it has to predict the output of three data points for every single batch (see Eq. 2). Overall, the training and validation phases for the Type4Py model is finished in about 165 minutes, which is acceptable for a deep learning model. However, the Typilus model completes a single training epoch in 267 seconds.[3]

Considering inference time, the Type4Py model predicts a single training batch (w/ 2536 data points) in 12.7 seconds (i.e. 5 ms for a data point on average). On the other hand, the TypeWriter model does prediction in less than 1 ms for a single training batch. Unlike TypeWriter, our Type4Py approach also performs KNN search in a high-dimensional space, which is why its inference speed is quite significantly slower. To compensate for this, one can decrease the output dimension of the model's linear layer. By reducing the linear layer's output dimension to 512, we obtain less than 1 ms inference time for a single data point at the cost of 1 to 2 percent decrease in overall performance. Also, Typilus does inference in less than 1 ms providing that its output vector's dimension is 64 and considering the fact that GNN-based models are substantially faster than biRNN-based models [50]. In summary, we believe that our proposed approach is practical for type-annotating an average Python project.

E. Qualitative Evaluation (RQ_5)

To qualitatively demonstrate the predictions of Type4Py approach and the state-of-the-art approaches, namely, Typilus and TypeWriter, we selected four different functions from the test set of Type4Py's dataset. Here, our goal is not to be overarching, rather present a few prediction examples that

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

A: https://github.com/bheklilr/python-workshop/blob/master/workshop1/pycalc/pycalc/ui.py

```
class PyCalcUI(tk.Frame):
    def __init__(self, calculator: Calculator, parent: tk.Tk=None) -> None:
        super().__init__(parent)
        self.calculator = calculator
        self.pack()
        self.create_widgets()
```

Target: Argument `parent`
Expected: `tk.TK`

Type4Py:	**Typilus:**	**TypeWriter:**
1) `tk.TK`	1) `QWidget`	1) `QObject`
2) `Optional[Node]`	2) `QObject`	2) `Resource`
3) `QObject`	3) `object`	3) `Tag`
4) `QApplication`	4) `Optional[QApplication]`	4) `Dict[str, Any]`

B: https://github.com/asiddhant/taskonomy-nlp/blob/master/allennlp/semparse/contexts/sql_table_context.py

```
class SqlVisitor(NodeVisitor):
    def __init__(self, grammar: Grammar) -> None:
        self.action_sequence: List[str] = []
        self.grammar: Grammar = grammar
```

Target: Argument `grammar`
Expected: `Grammar`

Type4Py:	**Typilus:**	**TypeWriter:**
1) `Grammar`	1) `RawKernelType`	1) `str`
2) `Fifo[T]`	2) `Table`	2) `Symbol`
3) `Dict[str, Entity]`	3) `object`	3) `Optional[str]`
4) `Optional[Any]`	4) `Dict`	4) `Graph`

C: https://github.com/pbougue/tartare/blob/master/tartare/interfaces/common_argrs.py

```
def get_current_date(self) -> date:
    args = self.parsers.parse_args()
    return args.get('current_date')
```

Target: Return type
Expected: `date`

Type4Py:	**Typilus:**	**TypeWriter:**
1) `datetime`	1) `Text`	1) `datetime`
2) `date`	2) `datetime`	2) `date`
		3) `Namespace`
		4) `str`

D: https://github.com/robertmrk/aiocometd-chat-demo/blob/master/aiocometd_chat_demo/conversation.py

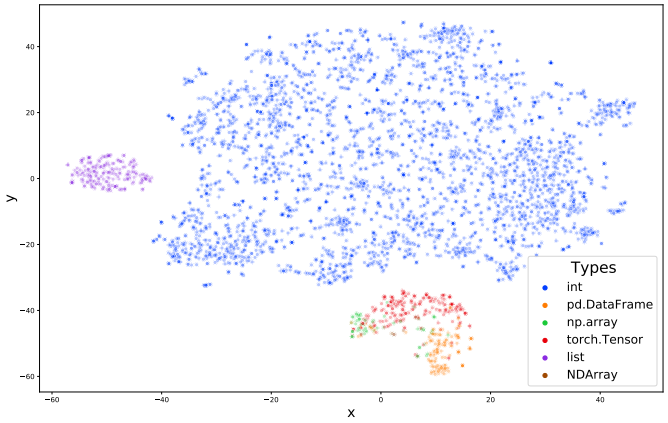
```
def rowCount(self, parent: Optional[QModelIndex] = None) -> int:
    return len(self._messages)
```

Target: Argument `parent`
Expected: `Optional[QModelIndex]`

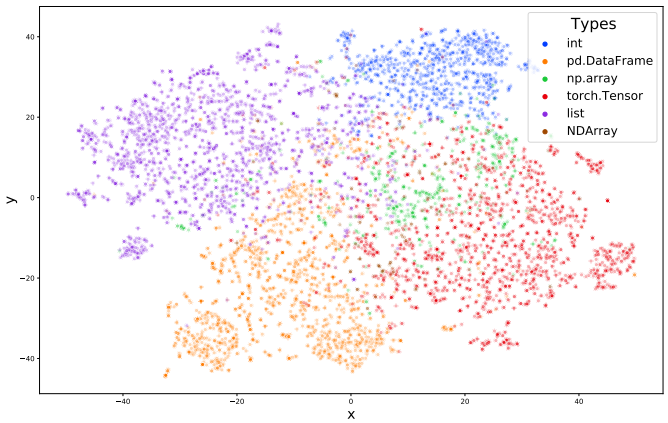
Type4Py:	**Typilus:**	**TypeWriter:**
1) `QModelIndex`	1) `Optional[QModelIndex]`	1) `int`
2) `ProcessType`	2) `QWidget`	2) `Iterable[int]`
3) `List[T]`	3) `Optional[QMainWindow]`	3) `Optional[int]`

Fig. 3: Qualitative evaluation of neural models on four different code snippets (not seen during the training phase)

show the kind of types that Type4Py can predict. Figure 3 shows predictions of neural models for the four functions from code snippet A, B, C and D. Starting with the code snippet A, the first suggestion of Type4Py is correct for argument `parent` while the other two approaches cannot make a correct prediction and instead they suggest `QWidget` and `QObject` types from PyQT5 framework. From the code snippet A (see Fig. 3), neither the name of the to-be-typed argument nor its usage give a hint for the prediction. Yet Type4Py model's first suggestion is correct. For the argument `grammar` in the code snippet B, Type4Py makes a correct prediction in its first suggestion while the other approaches' predictions are all incorrect. The name of the argument `grammar` and its usage give a strong hint about its type. For TypeWriter, it is impossible to predict argument `grammar` since the type `Grammar` is out of its type vocabulary. Consdering the code snippet C, the task is to predict the return type of function `get_current_date`. Both Type4Py and TypeWriter predict the type `date` in their second suggestion, which is correct. However, Typilus predicts a completely irrelavent type, `Text` in its first suggestion whereas the first suggestion of both Type4Py and TypeWriter is `datetime`, which is somewhat relevant to the expected return type. Lastly, we consider the code snippet D where Typilus makes a correct prediction while Type4Py's first suggestion is still relevant and somewhat correct. Once again, TypeWriter's predictions are all incorrect



(a) Type4Py



(b) Typilus

Fig. 4: t-SNE plot of learned type clusters in $\mathbb{R}^2$

as `Optional[QModelIndex]` is not present in its type vocabulary.

To qualitatively inspect the quality of learned type clusters, we employ the t-SNE technique [51]. It is a dimensionality reduction technique that allows us to visualize high-dimensional type clusters in a 2-dimensional space. For this evaluation, we select common types such as `int`, `list` and conceptually similar types, i.e. `mxnet.nd.NDArray` and `torch.Tensor`. Figure 4 shows t-SNE plot of 6 learned type clusters from the training set of Type4Py's dataset for both Type4Py and Typilus. Overall, It can be seen that the learned type clusters by Type4Py have better separation and are more compact while Typilus has substantial overlap between all of its selected learned type clusters (see Fig. 4b). For instance, common type clusters such as `int` and `list` have perfect separation whereas in the case of Typilus they have significant overlap with other type clusters. This complements our quantitative evaluation (subsection V-A) which shows the excellent performance of the Type4Py approach at predicting common types in comparison with Typilus. In addition, we observe that there is some overlap between the clusters of conceptually similar types (CSTs) such as `torch.Tensor`, `mxnet.nd.NDArray`, and `np.array` for Type4Py. The same can be observed for Typilus, which is in line with the finding of Allamanis et al. [15], which shows that Typilus confuses conceptually similar types. However, Type4Py has less confusion in separating CSTs as it can be observed from its learned type clusters of CSTs (see Fig. 4a).

VI. Discussion and Future Work

Based on the formulated RQ_1-RQ_5 and their evaluation in section V, we give the following concluding remarks:

- The superiority of Type4Py over TypeWriter could be attributed to the fact that Type4Py maps to-be-typed arguments and functions' return into a high-dimensional space (i.e. type clusters). This not only enables a much larger type vocabulary but also significantly improves the prediction of rare types.
- According to the results of ablation analysis (RQ_3), the three considered type hints, identifiers, code context, and visible type hints are all effective and positively contribute to the performance of Type4Py. Different from the work of Pradel et al. [14], for this work we opted for more precise extraction of available types, including dealing with type aliasing. The results show that our richer type hints significantly contribute to the model performance, both for Type4Py *and* TypeWriter. Moreover, 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., autocompletion purposes.
- As indicated in the qualitative evaluation (RQ_5), the HNN-based model of Type4Py has a better discriminative ability than the GNN-based model of Typilus, which results in a substantially higher type prediction performance. We attribute this to the inherent bottleneck of GNNs which is over-squashing information into a fixed-size vector [52] 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 their long-range dependencies.
- Considering the top-n predictions performance (RQ_2), we believe that our proposed approach is more suitable as a machine-aided tool for assisting developers in typing their Python codebases, given that Type4Py has an acceptable top-1 prediction performance and that developers are more likely to use the first suggestion by a tool [49]. To validate this, we plan to implement type suggestion as a feature for modern IDEs in future work.

Despite the superiority of Type4Py over state-of-the-art approaches Typilus and TypeWriter, Type4Py cannot make a correct prediction for types beyond its pre-defined (albeit large) type clusters. For example, it currently cannot synthesize types, meaning that it will never suggest a type such as `Optional[Map[str, int]]` if it does not exist in its type clusters. To address this, future research can explore

pointer networks [53] or a GNN model that captures type system rules.

VII. Summary

ML-based type inference is a challenging task as the type vocabulary is quite large, which makes traditional neural network-based approaches ineffective for predicting rare types. In this paper, we present Type4Py, a deep similarity learning-based type inference model for Python. Specifically, the Type4Py model is based on the DSL-based hierarchical neural network that efficiently maps 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 types of arguments and functions' return types. Overall, the results of the quantitative and qualitative evaluation show that the Type4Py model outperforms other state-of-the-art approaches. Most notably, the results of top-1 prediction show that our proposed approach achieves significantly higher precision than that of Typilus and TypeWriter's by a margin of 19.33% and 13.49%, respectively. This indicates that our approach is more capable of predicting rare and user-defined types than the other two state-of-the-art approaches.

Acknowledgment

This research work was funded by H2020 grant 825328 (FASTEN).

References

[1] [n. d.], "Ieee spectrum's the top programming languages 2019," https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019.

[2] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, 2014.

[3] A. Stuchlik and S. Hanenberg, "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*, 2011, pp. 97–106.

[4] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "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*, 2014, pp. 155–165.

[5] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: quantifying detectable bugs in javascript," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 758–769.

[6] G. Van Rossum, J. Lehtosalo, and L. Langa, "Pep 484–type hints," *Index of Python Enhancement Proposals*, 2014.

[7] J.-P. Ore, S. Elbaum, C. Detweiler, and L. Karkazis, "Assessing the type annotation burden," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 190–201.

[8] M. Hassan, C. Urban, M. Eilers, and P. Müller, "Maxsmt-based type inference for python 3," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 12–19.

[9] M. Furr, J.-h. An, J. S. Foster, and M. Hicks, "Static type inference for ruby," in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1859–1866.

[10] M. Madsen, "Static analysis of dynamic languages," Ph.D. dissertation, Aarhus University, 2015.

[11] Z. Pavlinovic, "Leveraging program analysis for type inference," Ph.D. dissertation, New York University, 2019.

[12] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 152–162.

[13] R. S. Malik, J. Patra, and M. Pradel, "Nl2type: inferring javascript function types from natural language information," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 304–315.

[14] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "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*, 2020, pp. 209–220.

[15] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 91–105.

[16] S. Chopra, R. Hadsell, and Y. LeCun, "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, 2005, pp. 539–546.

[17] J. Lehtosalo *et al.*, "Mypy-optional static typing for python," 2017.

[18] [n. d.], "Pytype," https://github.com/google/pytype.

[19] ——, "Pyright," https://github.com/microsoft/pyright.

[20] ——, "Pyre: A performant type-checker for python 3," https://pyre-check.org/.

[21] M. Salib, "Faster than c: Static type inference with starkiller," *in PyCon Proceedings, Washington DC*, pp. 2–26, 2004.

[22] E. Maia, N. Moreira, and R. Reis, "A static type inference for python," *Proc. of DYLA*, vol. 5, no. 1, p. 1, 2012.

[23] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 111–124.

[24] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 607–618.

[25] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," in *International Conference on Learning Representations*, 2019.

[26] I. V. Pandi, E. T. Barr, A. D. Gordon, and C. Sutton, "Opttyper: Probabilistic type inference by optimising logical and natural constraints," *arXiv preprint arXiv:2004.00348*, 2020.

[27] D. Jurafsky and J. H. Martin, *Speech and Language Processing (2nd Edition)*. USA: Prentice-Hall, Inc., 2009.

[28] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[29] R. J. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural computation*, vol. 1, no. 2, pp. 270–280, 1989.

[30] F. Liu, L. Zheng, and J. Zheng, "Hienn-dwe: A hierarchical neural network with dynamic word embeddings for document level sentiment classification," *Neurocomputing*, vol. 403, pp. 21–32, 2020.

[31] J. Zheng, F. Cai, W. Chen, C. Feng, and H. Chen, "Hierarchical neural representation for document classification," *Cognitive Computation*, vol. 11, no. 2, pp. 317–327, 2019.

[32] Y. Du, W. Wang, and L. Wang, "Hierarchical recurrent neural network for skeleton based action recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1110–1118.

[33] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[34] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[35] G. Rao, W. Huang, Z. Feng, and Q. Cong, "Lstm with sentence representations for document-level sentiment classification," *Neurocomputing*, vol. 308, pp. 49–57, 2018.

[36] W. Liao, M. Ying Yang, N. Zhan, and B. Rosenhahn, "Triplet-based deep similarity learning for person re-identification," in *Proceedings of the IEEE International Conference on Computer Vision Workshops*, 2017, pp. 385–393.

[37] D. Cheng, Y. Gong, S. Zhou, J. Wang, and N. Zheng, "Person re-identification by multi-channel parts-based cnn with improved triplet loss function," in *Proceedings of the iEEE conference on computer vision and pattern recognition*, 2016, pp. 1335–1344.

[38] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.

[39] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," *arXiv preprint arXiv:1812.06469*, 2018.

[40] C. D. Manning, H. Schütze, and P. Raghavan, *Introduction to information retrieval*. Cambridge university press, 2008.

[41] E. Loper and S. Bird, "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*, 2002, pp. 63–70.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[44] [n. d.], "Annoy," https://github.com/spotify/annoy.

[45] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[46] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[47] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[48] [n. d.], "Typilus' public implementation," https://github.com/typilus/typilus.

[49] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[50] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–43, 2019.

[51] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[52] U. Alon and E. Yahav, "On the bottleneck of graph neural networks and its practical implications," *arXiv preprint arXiv:2006.05205*, 2020.

[53] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in neural information processing systems*, 2015, pp. 2692–2700.