Combining Type4Py's Deep Similarity Learning-based Type Inference with Static Type Inference for Python

Anhar Al Haydar
Supervisors: Amir M. Mir and Sebastian Proksch
EEMCS, Delft University of Technology, The Netherlands
24-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Combining Type4Py's Deep Similarity Learning-based Type Inference with Static Type Inference for Python

Anhar Al Haydar[1]
Supervisor(s): Amir M. Mir[1] **, Sebastian Proksch**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

Dynamic programming languages (DPLs), such as Python and Ruby, are often used for their flexibility and fast development. The absence of static typing can lead to runtime exceptions and reduced program understandability. To overcome these problems, some DPLs have introduced optional static typing. Because of the tedious effort of adding type annotations to existing projects, different approaches have been employed to generate type annotations. Static type inference methods are sound in their suggestions, but the dynamic nature of DPLs, combined with insufficient satisfied static dependencies, can cause imprecision. Other proposed approaches used machine learning (ML)-based type inference to predict type annotations. ML-based methods don't have the limitations of static type inference, however, their performance depends on the training set's quality and they cannot guarantee type correctness because of their probabilistic techniques. One of such ML-based inference approaches, is the state-of-the-art Type4Py model. Type4Py suffers from some of the same limitations of other learning-based approaches, e.g. it cannot predict types outside of its pre-defined type clusters. To this end, this paper presents `hpredict`, a tool that combines type prediction of Type4Py's pre-trained model with static type inference. `hpredict` runs Type4Py's learning-based inference and static type inference on different copies of type slots and combines the predictions from both methods. Experiments on the test set of the ManyTypes4Py dataset show that `hpredict` outperforms Type4Py significantly by 11% regarding Top-10 prediction. The findings of this research, lend evidence that `hpredict` can increase Type4Py's general type prediction performance by employing static type inference as well.

1 Introduction

Dynamically typed programming languages (DPLs) have become very popular in software development. In particular, the dynamic programming language Python is ranked as most popular by the IEEE Spectrum [9]. DPLs have some useful features, for example, increased flexibility and fast prototyping. However, the absence of static typing like languages can lead to problems as unforeseen exceptions during run-time and reduced program understandability. To combat these problems, some dynamically typed languages have introduced optional static typing. Nevertheless, static typing on existing projects requires manually adding type annotations, which can be tedious [3]. Static type inference methods can reduce this tedious effort by inferring part of the project's type annotations. Additionally, such methods can infer annotations soundly when sufficient static dependencies are satisfied, which means that suggested type annotations are mostly guaranteed to be type-correct. Still, the dynamic nature of DPLs, such as allowing lists to contain differently typed elements or dynamic evaluation, can be problematic for static type inference methods. Some issues of static type inference on DPLs include potential imprecision [7] and being relatively slow.

Recently, researchers have developed a number of *Machine Learning* (ML) based type inference models to predict types for dynamic languages [1, 4, 8], based on type hints and features extracted from the source files. It has been shown that such approaches can be more precise than static type inference methods on DPLs [1, 8], although, the approaches' performance is dependent on the quality of the used training set. One of these ML-based type inference models is the state-of-the-art deep similarity learning-based hierarchical neural network (HNN) model Type4Py [4], which can predict types for variables, function arguments, and return values in Python files.

However, Type4Py does have some limitations [4]. Firstly, Type4Py cannot give sound predictions since it is a probabilistic approach and not a static one, i.e. it provides type predictions with various confidence scores instead of a guaranteed annotation. Secondly, it is not able to predict types outside of its pre-defined type clusters, e.g. user-defined types that have not been seen before.

To overcome the individual issues from static type inference and the ML-based type inference model Type4Py, this paper presents `hpredict`, which combines the type prediction of Type4Py's pre-trained model with static type inference from the Pytype tool [13]. `hpredict` is supposed to utilize the strengths of both ML-based type inference and static type inference methods, while reducing the effects of both strategies' limitations. This paper aims to answer if the general type prediction performance of Type4Py can be improved if combined with static type inference, i.e. in the form of `hpredict`, using the original dataset ManyTypes4Py [5].

This main research question is divided into two sub-questions that will be answered. Namely:

- Sub-RQ1: How does Type4Py perform compared to the static type inference tool Pytype?
- Sub-RQ2: How does `hpredict` perform compared to Type4Py alone?

The experiments' results show that `hpredict` outperforms Type4Py with a margin of 11%.

All in all, this paper makes the following contributions:

- `hpredict`, which stands for *hybrid predict*, a combination of Type4Py's ML-based type prediction and Pytype's static type inference that can predict type annotations for Python code.
- Empirical evidence that `hpredict` increases Type4Py's general type prediction performance.

The rest of this paper is structured in the following way. In Section 2, related work is discussed. Moreover, in section 3, the study's methodology is described. The setup of the evaluation are presented in Section 4. Furthermore, Section 5 gives the results of the evaluation. This study's approach to Responsible Research is outlined in Section 6. Section 7 discusses the achieved results. Finally, conclusions and directions for future work are given in Section 8.

2 Related Work

Static type inference for Python: Since the introduction of type hints to Python, multiple static type checkers and type inference tools have been released for Python. These include static type checker MyPy [11] and the tools Pytype [13], PyRight [12], and Pyre [14], that can perform both static type checking and type inference. As mentioned earlier, such static type inference approaches can be imprecise on DPLs.

ML-based type inference: In 2019, Malik et al. [3] presented NL2Type, which is a model based on neural networks that can predict type annotations for JavaScript functions. To do this, it makes use of natural language information found in source code.

Inspired by NL2Type, Pradel et al. [8] proposed TypeWriter. The TypeWriter model is based on hierarchical deep neural networks that can infer type annotations for Python code. For the inference, it utilizes not only natural language information, but also code context. Additionally, TypeWriter performs combinatorial search strategies and an external type checker to confirm its predictions.

These works use small and fixed-size type vocabularies, which can result in problems in inferring user-defined and rare types. To address this, Allamanis et al. [1] proposed Typilus, which is a graph-based neural network model. Typilus makes use of things like identifiers, syntactic patterns, and data flow to infer types for Python. Typilus has been shown to outperform models with small type vocabularies.

However, as outlined by Mir et al. [4], the discussed works have a couple of drawbacks. Firstly, the neural models are trained and evaluated on datasets containing non-type-checked developer-given type annotations. This can be problematic for obtained model results because these annotations are not guaranteed to be correct. Secondly, the discussed works have decent performance for predictions being in the Top-10 suggestions, whereas a focus on Top-1 can be more effective for developers as they tend to pick a tool's first prediction [6].

Considering the mentioned drawbacks of some tools, Mir et al. [4] proposed the Type4Py model. Type4Py is based on a deep similarity learning-based hierarchical neural network (HNN) model, which can differentiate between different types by linking Python programs to high-dimensional type clusters. Type4Py is trained on a large type-checked dataset [5], which enables it to work with large type vocabularies. Yet, Type4Py has some limitations such as not being able to provide sound type predictions and not being able to predict types outside of its pre-defined type clusters, e.g. generated types like `List[List[Tuple[int]]]`. In [4], Type4Py has been shown to achieve an MRR of 77.1%, outperforming Typilus and TypeWriter by 8.1% and 16.7%, respectively.

Very recently, researchers developed hybrid type inference approach, called HiTyper [2]. HiTyper works with type dependency graphs (TDGs) that record type dependencies among variables. Static type inference is largely used to infer the gaps in the TDG. For types that cannot be statically inferred, HiTyper accepts recommendations from deep learning type inference models. Wrong predictions on a TDG are removed by building and using a series of type rejection rules, after which static type inference is performed again on remaining correct type predictions. The researchers outline a benefit of this approach being that the static type inference rules are insensitive to type frequencies, which overcomes the ML-models' issue with predicting rare type annotations.

As a hybrid type inference approach, HiTyper combines static type inference with type inference from ML-based models. In concept, this is similar to this research's topic. However, this research specifically centers around improving the general type prediction performance of Type4Py by combining its type inference with static type inference in a hybrid manner.

3 Methodology

This section discusses the proposed approach for solving the two research sub-questions presented earlier. Firstly, in Section 3.1, an outline is given for the pipeline developed for answering sub-RQ1. Secondly, Section 3.2 details the changes made to the pipeline to include `hpredict` and its evaluation.

3.1 Sub-RQ1

For the first sub-question, the pipeline consists of two sub-pipelines that are run independently from each other. Namely, a sub-pipeline for Type4Py and another one for static type inference (STI), as can be seen in the overview in Figure 1.

The pipeline starts by going through the processed JSON files of ManyTypes4Py [5]. The JSON files include tags that classify source files as either train, validation, or test files. The dataset's test set is recreated by selecting all source files that are classified as test files. After this process, ManyTypes4Py's test set contains projects (repositories) containing a number of Python source files. Two copies of the resulting test set are created, where each sub-pipeline uses another copy.

The Type4Py sub-pipeline receives its copy of the test set and starts by handing it to its first component. This part of the pipeline performs type annotation prediction by calling the Type4Py API on each of the test set's source files. Errors can occur if Type4Py reaches a timeout or if the source file contains type mistakes. Such errors are logged and then the process continues with the next source file. Consequently, the Type4Py API's returned JSON objects are formatted to have a similar structure to the output of the static type inference pipeline. This last step produces a JSON file per Python project (repository) containing type annotation predictions for each of the project's files.

Similarly to the Type4Py pipeline, the static type inference pipeline starts after receiving its copy of the test set. In its first component, pre-existing type annotations are removed from the received source files. This step is performed because Pytype appeares to only infer types for expressions without pre-existing annotations. Performing type annotation removal on the test set's source files, results in a type annotation coverage of 24%, which should actually be 0%. The reason was found to be 11,790 files (see Table 1) without type slots but with non-zero coverage. To overcome this issue, source files with non-zero coverage are logged and ignored in the remainder of the static type inference pipeline. After type annotation removal, static type inference is performed by Pytype on the unannotated projects' source files. During this process, Pytype creates `pyi` files that contain its suggested type annotations per source file. Next, the inferred type annotations from the `pyi` files are merged back into the respective Python source files. Measuring the type annotation coverage gives 43%, which is a coverage improvement of 19% from the original test set. Lastly, the annotated source files are passed to the LibSA4Py [10] tool's extraction functionality, which produces JSON analyses of a project's Python files. If any errors are encountered while removing annotations, running static type inference, applying the inferred types, or performing extraction, then the error is logged and the process continues with the next source file.

In the pipeline's next step, the JSON files from both sub-pipelines are merged into new JSON objects. Since all JSON files have a similar structure by now, Type4Py's predictions and Pytype's suggestions can be combined in the following way. First, an intersection is taken of both tools' JSON file names to avoid the processing of projects without inferred types from both Type4Py and Pytype. This is done because errors could have occurred in different components of the two sub-pipelines that prevented files or entire projects from being processed. Next, the source file paths in Type4Py's JSON files are iterated over while it is checked if the file path exists in Pytype's JSON file. If this is the case, then variables, parameters, and return values are considered among all global variables, functions, and classes in the Type4Py analyzed source file. During this process, for every variable, parameter, and return value, it is checked if both developer-provided type annotations (ground truth) and a Pytype suggestion exist for the type slot. If so, then Pytype's suggestion is added as a new JSON entry along with Type4Py's predictions. If either ground or Pytype's suggestion is not available, then the expression and file path are logged and processing continues with the next type slot. This way, inconsistencies are avoided and both tools are evaluated on predictions for the same type slots. In this process, new JSON files are created, where each file contains a list of JSON objects. The JSON objects include the ground truth, whether or not ground truth is a base type, the prediction task (variable, parameter, or return value), Type4Py's predictions, and Pytype's suggestions for type slots. The format of the JSON files largely follows Type4Py's output format to mostly stay compatible with Type4Py's evaluation functionality.

Table 1: Dataset characteristics at different stages of the pipeline

Metrics in pipeline stage	Amount
Projects before running pipeline	5,203
Projects in test set	3,296
Test files	57,783
... after type annotation removal	57,743
...... with non-zero type coverage	11,790
Processed projects after STI pipeline	4,354
Processed projects after Type4Py pipeline	4,086
Processed projects before merging JSON files	4,086
Type slots before evaluation	9,329
Type slots evaluated for Type4Py and Pytype	8,870
Type slots evaluated for hpredict	8,901
No ground truth during evaluation	699,074
No Pytype inferred type during evaluation	109,007

Finally, the evaluation functionality of Type4Py is extended to enable processing of Pytype's suggested type annotations as well, which is done by also keeping track of correct predictions for Pytype. During evaluation, the ground truth, Type4Py's predictions, and Pytype's suggestion are transformed to be correctly comparable. This transformation is done by performing three steps. First, types are made consistent by converting fully qualified names (like `typing.List`) to a base name by removing the prefix (in this example `typing.`). Second, in the case of ground truth, it is checked to be neither `Any` nor `None` because evaluation of such cases is not useful. Third, resolving of a type alias is performed on the consistent type, which is done to prevent different type aliases for the same type to be incorrectly compared to each other. Evaluation is performed for every type slot by iterating over the list of JSON objects and separately evaluating Type4Py's predictions and the single Pytype suggestion. This way, the performance of each tool is computed separately on ManyTypes4Py's test set.

Characteristics of the dataset at various stages of the sub-pipeline can be found in Table 1, where one can see a decreasing number of files, projects, etc. in next stages due to occurred errors.

3.2 Sub-RQ2

In the pipeline for sub-RQ2, a new encapsulating component, i.e. `hpredict`, is added that combines Type4Py's prediction

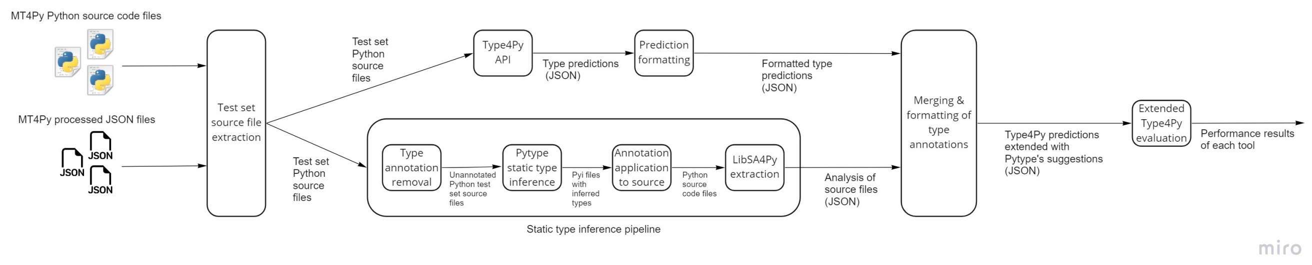
MT4Py Python source code files
MT4Py processed JSON files
JSON
JSON
JSON
Test set source file extraction
Test set Python source files
Type4Py API
Type predictions (JSON)
Prediction formatting
Formatted type predictions (JSON)
Test set Python source files
Type annotation removal
Unannotated Python test set source files
Pytype static type inference
Pyi files with inferred types
Annotation application to source
Python source code files
LibSA4Py extraction
Analysis of source files (JSON)
Static type inference pipeline
Merging & formatting of type annotations
Type4Py predictions extended with Pytype's suggestions (JSON)
Extended Type4Py evaluation
Performance results of each tool
miro

Figure 1: Pipeline overview for research sub-question 1

functionality with Pytype's static type inference. Similarly to the pipeline for sub-RQ1, two different copies of ManyTypes4Py's test set are used, which are required by `hpredict` for running the two sub-pipelines. In Figure 2, the pipeline overview for sub-RQ2 is presented.

The first component of this pipeline is identical to the one discussed for sub-RQ1. Namely, it goes through the processed JSON files and produces ManyTypes4Py's test set. Once again, the test set is copied and the two copies are passed to the next step.

Next, in `hpredict`, the two test set copies are used internally to run both sub-pipelines. The two sub-pipelines are identical to the ones presented in Section 3.1. To summarize, test set copies are used as input, after which `hpredict` runs these these sub-pipelines to produce each tool's predictions in the form of JSON files.

Afterward, the pipeline differs from sub-RQ1's, starting from the *Merging & combination of type annotations* step. In this component, for type slots that have ground truth and a Pytype suggestion available, Pytype's suggested type is added to Type4Py's list of predictions. This is different from sub-RQ1's approach, where new Pytype entries are created next to Type4Py's predictions for type slots that have ground truth.

Type4Py's predicted annotations for a type slot are sorted by their *confidence score*, i.e. a value between 0 and 1.0 representing Type4Py's confidence in the prediction. Since Pytype performs static type inference, its suggestions are guaranteed to be type-correct, as opposed to Type4Py's probabilistic predictions. This means that Pytype's inferred types can be considered to have a full confidence score of 1.0. As 1.0 is the highest possible confidence score, Pytype's suggestion is placed at the head of Type4Py's prediction list during the merging step. Using this approach, the merging component of `hpredict` produces files containing lists of JSON objects. The JSON objects hold ground truth, whether or not ground truth is a base type, the prediction task, and Type4Py's list of predictions, prepended with Pytype's suggestion. This format of the JSON objects is almost entirely compatible with Type4Py's original evaluation implementation.

Lastly, the resulting JSON files are passed to a version of Type4Py's original evaluation function, which is modified by adding the transormation of types discussed in the last section. For sub-RQ2, evaluation is performed on the lists of combined predictions to compute the performance of `hpredict`. This determined performance can then be compared to Type4Py's or Pytype's single performance from sub-RQ1.

4 Evaluation Setup

This section presents the baseline tools for performance comparison, `hpredict`'s implementation details, and the evaluation metrics to measure the tools' performance.

4.1 Baseline tools

`hpredict` is compared with the deep similarity learning-based HNN-model Type4Py [4] and static type checking and inference tool Pytype [13].

Regarding the static inference tool, there are a couple of alternatives to Pytype that were considered. Namely, Pyre [14] and Pyright [12]. For this research, the runtime should not be too high and the inference tool should be able to infer types for variables, parameters, and return values. Pyright and Pyre both claim to be fast and meant for large code bases, which seems like a good fit for ManyTypes4Py [5]. However, a problem with Pyre is that, at the moment, it only infers types for variables, which is not sufficient as inference for parameters and return values are also needed. Regarding Pyright, it was not used because it couldn't be well integrated into the static type inference pipeline. Pytype didn't seem to have this problem, although, Pytype is described to slow down noticably once a source file is larger than approximately 1,500 lines. Because of the significant size of the ManyTypes4Py dataset (see Table 1), this was a potential deal-breaker. However, during experiments it was found that most source files in ManyTypes4Py's test set were being processed relatively quickly when using multiprocessing. All in all, relative ease of Pytype's integration into the pipeline and its sufficient speed are why Pytype is used as static type inference tool for this research.

4.2 Implementation details

`hpredict` and its internal Type4Py and static type inference sub-pipelines are mostly implemented in Python 3 and its ecosystem. A minor part of the system is implemented in *bash* scripts, e.g. for running static type inference on a project or applying inferred types to Python source files. Regarding Type4Py and Pytype, their public implementation is used from GitHub [13, 15]. Multiple components of the pipeline like type annotation removal, static type inference, type annotation application, and Type4Py prediction

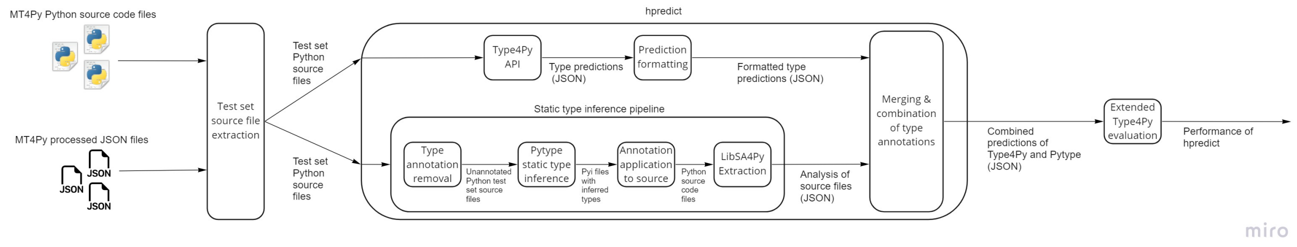
MT4Py Python source code files
MT4Py processed JSON files
JSON
JSON
JSON
Test set source file extraction
Test set Python source files
Test set Python source files
hpredict
Type4Py API
Type predictions (JSON)
Prediction formatting
Formatted type predictions (JSON)
Static type inference pipeline
Type annotation removal
Unannotated Python test set source files
Pytype static type inference
Pyi files with inferred types
Annotation application to source
Python source code files
LibSA4Py Extraction
Analysis of source files (JSON)
Merging & combination of type annotations
Combined predictions of Type4Py and Pytype (JSON)
Extended Type4Py evaluation
Performance of hpredict
miro

Figure 2: Pipeline overview for research sub-question 2

are parallelized using the parallel execution functionality of LibSA4Py [10], which itself uses the *joblib* package. Another use of LibSA4Py regards type annotations are removed from Python source files. The Type4Py API is used in a local form using a Docker image. Lastly, information is extracted from JSON objects using the *jq* package.

All experiments were performed on a computer running a Linux operating system (Ubuntu 18.04.5 LTS).

4.3 Evaluation metrics

In evaluation of Type4Py, Pytype, and `hpredict`, some of the metrics and general approach used by Mir et al. [4] are followed in this research. Some of the same evaluation metrics are used to ensure that the results of this study can be compared to those of Type4Py. Type predictions are evaluated for variables, function parameters, and return values.

Comparison of type prediction from Type4Py or Pytype with ground truth is performed using the *Exact Match* and *Base Type Match* criteria originally proposed by Allamanis et al. [1]. In Exact Match, two types have to be exactly the same for a match to exist. Base Type Match is more lenient as it only matches base types and ignores type parameters, e.g. `int` is ignored in `List[int]`.

As evaluation metric, the *Top-n* metric is used, which classifies a prediction as correct if one of the `n` predictions with highest probability is true. For this study, Top-1 and Top-10 are used as evaluation metrics. Top-1 is used to enable comparison between Pytype and Type4Py since Pytype gives only one suggestion per type slot. Top-10 is used to compare `hpredict`'s type prediction performance with Type4Py's. Unlike Mir et al. [4], this research does not use *Mean Reciprocal Rank* (MRR@n), as an evaluation metric. MRR@n is not used for two reasons. First, Pytype infers only one type per type slot, which means that MRR@1 would be equivalent to the Top-1 metric. Second, in `hpredict`, Pytype's suggestion is prepended to Type4Py's list of predictions, which makes MRR@n unsuitable in this case. MRR@n might become usable with a kind of normalization of the prediction, but this is outside the scope of this study's research questions.

Finally, like in the evaluation methodologies of Mir et al. [4] and Allamanis et al. [1], types are considered *ubiquitous* if the type is in str, int, list, bool, float,

5 Evaluation Results

In this section, the evaluation results are outlined for both research sub-questions. Sections 5.1 and 5.2 discuss the results for sub-RQ1 and sub-RQ2, respectively.

- Sub-RQ1: How does Type4Py perform compared to the static type inference tool Pytype?
- Sub-RQ2: How does `hpredict` perform compared to Type4Py alone?

5.1 Performance of Type4Py vs. Pytype (sub-RQ1)

In this subsection, Type4Py's type prediction performance is compared to that of Pytype.

Method: The local Type4Py API and Pytype are used to perform type prediction for the Python source files from ManyTypes4Py's [5] test set. Both tools' inferred type annotations are evaluated by considering their Top-1 and Top-10 predictions, respectively. For this research question, type predictions for variables, function parameters, and return values are taken into account.

Results: Tables 2 and 3 show the overall performance when considering Top-1 for Pytype and Top-1, 10 for Type4Py. Regarding Top-1 prediction, Pytype outperforms Type4Py based on the Exact Match and Base Type Match of all types. At the Top-1 prediction, Pytype performs better than Type4Py with a margin of 5.1% and 13.8% for Exact Match and Base Type Match, respectively. The only case for which Type4Py performs better than Pytype, is predicting types for ubiquitous types in Exact Match, where Type4Py performs 4.3% better. It should be noted that Type4Py's results are significantly lower than the ones found by Mir et al. [4].

5.2 Performance of hpredict (sub-RQ2)

In this subsection, `hpredict`'s type prediction performance is compared to that of Type4Py and Pytype.

Method: Similar to sub-RQ1, the local Type4Py API and Pytype are used to perform type prediction for the Python source files from ManyTypes4Py's [5] test set. However, now the sub-pipelines are encapsulated by `hpredict` alongside modified pipeline components for merging JSON files and performing evaluation. `hpredict`'s inferred type annotations are evaluated by considering its Top-1 and

Table 2: Performance evaluation of neural model Type4Py

Top-n prediction	Metrics	%
Top-1	Exact Match All	21.8
	Exact Match Ubiquitous	54.1
	Exact Match Common	5.8
	Exact Match Rare	0.1
	Base Type Match All	24.9
	Base Type Match Common	12.9
	Base Type Match Rare	3.7
Top-10	Exact Match All	32.1
	Exact Match Ubiquitous	73.5
	Exact Match Common	18.0
	Exact Match Rare	0.6
	Base Type Match All	39.9
	Base Type Match Common	37.0
	Base Type Match Rare	9.3

Table 3: Top-1 performance evaluation of Pytype

Top-n prediction	Metrics	%
Top-1	Exact Match All	26.9
	Exact Match Ubiquitous	49.8
	Exact Match Common	18.7
	Exact Match Rare	9.7
	Base Type Match All	38.7
	Base Type Match Common	45.9
	Base Type Match Rare	23.7

Top-10 predictions, respectively. Like for sub-RQ1, type predictions for variables, function parameters, and return values are taken into account.

Results: Table 4 shows `hpredict`'s overall performance when considering Top-1 for Top-10. These results are compared with Type4Py and Pytype's results in Tables 2 and 3, respectively.

Regarding Top-1 prediction, it can be seen that `hpredict` is approximately matching Pytype's performance for Exact match and Base Type Match. For most of these cases, it performs the same as Pytype, and for the other ones, it performs around 0.1% better or worse. In comparison with Type4Py, `hpredict` outperforms Type4Py based on the Exact Match and Base Type Match of all types with a margin of 5.0% and 13.8% for all types, respectively. This is almost identical to Pytype's performance in sub-RQ1.

For Top-10 prediction, one can see much more notable differences between the tools' performances. `hpredict` outperforms Type4Py in every case. For Exact Match of all types, `hpredict` outperforms Type4Py with a significant margin of 11.0%. Breaking it down, shows that `hpredict` outperforms Type4Py on rare types with a margin of 9.5% on the low end and on common types Type4Py is outperformed by 13.3% on the high end. For Base Type Match, the differences are more notable. Namely, for all types, `hpredict` outperforms Type4Py with a margin of 19.2%. When breaking it down for the different types, one can see that Type4Py

Table 4: Performance evaluation of hpredict

Top-n prediction	Metrics	%
Top-1	Exact Match All	26.8
	Exact Match Ubiquitous	49.8
	Exact Match Common	18.7
	Exact Match Rare	9.6
	Base Type Match All	38.7
	Base Type Match Common	45.8
	Base Type Match Rare	23.8
Top-10	Exact Match All	43.1
	Exact Match Ubiquitous	84.9
	Exact Match Common	31.3
	Exact Match Rare	10.1
	Base Type Match All	59.1
	Base Type Match Common	67.8
	Base Type Match Rare	29.2

is outperformed by 30.8% on common types and 19.9% on rare types, respectively.

6 Responsible Research

During this research, significant effort has been spent on performing the experiments responsibly, honestly, and transparently.

The topic of this research hasn't required any direct involvement of other people or their data. For example, neither human participation was needed nor private data was collected. The only data used in the research is in the form of the ManyTypes4Py [5] dataset. ManyTypes4Py is constructed from public GitHub repositories, which should avoid any potential problems with developers' private information or using work without consent.

Regarding the implementation of `hpredict` and its internal sub-pipelines, all of it is located on a private GitHub repository of the Software Analytics Lab at TU Delft. If and/or when this organization wants to make this repository public, then the implementation should be accessible and usable for other experiments.

In Section 3, the approach is discussed for both of this study's research questions. This discussion has been made as detailed as possible to (along with a potential public implementation) ensure reproducibility and verifiability of the used methodology.

7 Discussion

Based on the entire study's process there are a number of noteworthy remarks to be discussed.

- During experimentation, it was found that Pytype tends to sometimes add additional classes and/or functions in eiter the static inference or type annotation application step. It is not clear why and in which step exactly this can happen.
- In Section 3.1, it was mentioned that after static type inference by Pytype, type annotation coverage of the test set arrived at 43%. ManyTypes4Py [5] contains two

kinds of datasets, namely, one is the original one and the other is augmented with additional type annotations using the Pyre [14] tool. In this study, the original (unaugmented) dataset was used to start with, however, because of time limitations, it was not possible to change over to the augmented dataset.

- In Section 5, Pytype's static type inference has been found to be valuable in improving the ML-based type inference of Type4Py. However, perhaps other static type inference tools could perform better for `hpredict` (in certain cases), but his has not been researched yet.
- As can be seen in Table 1, a significant number of files and projects have been left out between stages of the pipeline due to errors. A lot of the errors were caused by programming mistakes in ManyTypes4Py's source files. It might be possible to somehow create fixed versions of these files before running the system, which could save more files and projects for type prediction and evaluation. This could possibly lead to more reliable evaluations of the tools' performances.
- In Section 5.1, it was mentioned that the determined performance of Type4Py is significantly lower than that found by Mir et al. [4]. For Top-10 Exact Match on all types, the difference amounts to 43.7%. It could possibly be that the ManyTypes4Py dataset has changed somewhat between both experiments, which can explain relatively small differences in performance, but 43.7% is so high that it seems highly improbable to be caused by (a relatively small number of) changes to the dataset. It is reasonable to think that there could be mistakes in the modified evaluation functionality used to determine the tools' performance. Namely, before transforming types (see Section 3.1), performance of the tools was almost non-existent. It could be that some other mistakes were overlooked in the evaluation's implementation, specifically in the transformation and comparison of types, which might have decreased performance significantly for all evaluated tools. If this is true, then the performance of each tool could actually be substantially higher than what was found this in this study. However, since the evaluation was the same for each tool, the differences in performance between the tools should still be relatively reliable.
- Given the results of sub-RQ1, Pytype was found to be more effective than Type4Py in most cases for Top-1. Of course, some of these results could be caused by faulty evaluation, however, for some cases it could be expected that Pytype outperforms. Namely, rare types or more difficult types such as `Dict[str, Optional[List[int]]]` will less likely be inferred by Type4Py since these types might not exist in its type clusters, while Pytype's static type inference might be more successful. The only case where Type4Py performs better than Pytype, is in Exact Match on ubiquitous types. This could be explained by ML-based inference models' generally good prediction of ubiquitous types since their training sets probably contain a large number of such types.
- `hpredict`'s performance was found to approximately match that of Pytype for Top-1 predictions (sub-RQ2). It could be expected that `hpredict`'s should at least perform as well as Pytype since it uses Pytype internally and has its suggestion in its list of predictions. The 0.1% difference between the two tools could be caused by potential mistakes in evaluation since, for Top-1, `hpredict` should perform the same as Pytype. During evaluation, only the first inferred type is considered for Top-1 and since Pytype's suggestion is prepended to `hpredict`'s list of predictions, it should evaluate like Pytype. Still, even with the 0.1% difference, both tools evaluate almost identically, which does support the above reasoning.
- Regarding Top-10, `hpredict` outperforms Type4Py significantly. Namely, for Exact Match on all types, Type4Py is outperformed by 11%. This significant improvement could possibly be explained by `hpredict`'s combination of Type4Py and Pytype's static type inference. In cases where Type4Py's learning-based approach could struggle, e.g. rare types, static type inference could provide `hpredict` with better predictions than Type4Py. The biggest performance improvement (30.8%) can be found in Base Type Match on common types. This might be caused by the combined predictions of Type4Py and Pytype, where there could be more often a correct type prediction from `hpredict`.

8 Conclusions and Future Work

The overarching research question of this paper has been if the general type prediction performance of Type4Py can be improved if combined with static type inference, i.e. in the form of `hpredict`, using the original dataset ManyTypes4Py [5].

To this end, this paper makes the two contributions. First, `hpredict`, which stands for *hybrid predict*, which combines Type4Py's type prediction functionality with Pytype's static type inference to predict type annotations for Python code. And second, empirical evidence that `hpredict` can increase Type4Py's general type prediction performance by using static type inference.

To answer this question, it was split in the following sub-questions:

- Sub-RQ1: How does Type4Py perform compared to the static type inference tool Pytype?
- Sub-RQ2: How does `hpredict` perform compared to Type4Py alone?

For sub-RQ1, it was found that Pytype outperforms Type4Py in most cases. Regarding Top-1 prediction, Pytype outperformed Type4Py with a margin of 5.1% and 13.8% for Exact Match and Base Type Match, respectively. Because of the large difference in evaluated performance of Type4Py between this study and Mir et al. [4], it is suspected that the used evaluation implementation could contain some mistake(s).

For sub-RQ2, regarding Top-10, it was found that `hpredict` outperforms Type4Py significantly by 11% on all types and Exact Match. Moreover, `hpredict` was found to approximately match Pytype's performance for Top-1 prediction.

To answer the main research question, it appears that Type4Py's type prediction performance indeed can be improved when combined with static type inference in the form of something like `hpredict`. Namely, an improvement of 11% was found over Type4Py's purely learning-based approach. It appears that when Type4Py's learning-based approach is combined with static type inference, the approaches complement each other with their strenghts and reduce each others' weaknesses to some extent.

During this study, a couple of new research questions arose that could be worked on next.

- Sub-RQ3: What combination strategy, i.e. the way Pytype's suggestion is added to Type4Py's list of predictions, maximizes `hpredict`'s performance?
- Sub-RQ4: Can an incremental combination of Type4Py and Pytype lead to better type prediction performance? Here incremental, means running one tool after the other on the same source files for some amount of time.

For future work, there may be a couple of noteworthy things. First, one could look into running the pipeline on ManyTypes4Py's augmented dataset instead of the original one. With the augmented dataset, a higher type annotation coverage could be achieved for the original test set, thus providing a static type inference tool like Pytype with more type information, which could potentially result in different findings.

Furthermore, one could research other static type inference tools and their performance to check if another tool is more suitable than Pytype.

It is important to also look into the implemented evaluation functionality since it might contain some mistakes that are impacting the results significantly. If any changes are made or mistakes are removed, then evaluation should be performed again to gain new reliable results. Additionally, if possible, it would be useful to run the evaluation of Mir et al. on the current ManyTypes4Py test set. This way, the results can be double-checked in comparison with this study's findings.

Lastly, it could be beneficial to make sure whether or not parts of the pipeline can be improved to increase things like efficiency and/or type prediction performance.

References

[1] Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. 2020. Typilus: neural type hints. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 91-105).

[2] Gao, B., Goa, C., Li, Z., Lo, D., Lyu, M., Peng, Y., Zhang, Q. 2022. Static Inference Meets Deep Learning: Approach for Python.

[3] Malik, R. S., Patra, J., and Pradel, M. 2019. NL2Type: inferring JavaScript function types from natural language information. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, (pp. 304–315).

[4] Mir, A. M., Latoskinas, E., Proksch, S., and Gousios, G. 2022. Type4py: Deep similarity learning-based type inference for python. In 44th International Conference on Software Engineering (ICSE '22).

[5] Mir, A. M., Latoskinas, E., and Gousios, G. 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, (pp. 585-589).

[6] Parnin, C. and Orso, G. 2011. Are automated debugging techniques actually helping programmers? In Proceedings of the 2011 international symposium on software testing and analysis, (pp. 199–209).

[7] Pavlinovic, Z. 2019. Leveraging Program Analysis for Type Inference. Ph.D. Dissertation. New York University.

[8] Pradel, M., Gousios, G., Liu, J., and Chandra, S. 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 (pp. 209-220).

[9] [n. d.]. [n.d.]. IEEE Spectrum's the Top Programming Languages 2021. https://spectrum.ieee.org/top-programming-languages.

[10] [n.d.]. [n.d.]. LibSA4Py's public implementation. https://github.com/saltudelft/libsa4py/.

[11] [n. d.]. [n.d.]. Mypy: A static type checker for Python 3. https://mypy.readthedocs.io/.

[12] [n. d.]. [n.d.]. PyRight's public implementation. https://github.com/microsoft/pyright/.

[13] [n.d.]. [n.d.]. Pytype's public implementation. https://github.com/google/pytype/.

[14] [n. d.]. [n.d.]. Pyre: A performant type-checker for Python 3. https://pyrecheck.org/.

[15] [n.d.]. [n.d.]. Type4Py's public implementation. https://github.com/saltudelft/type4py