Static Analysis Complements Machine Learning: A Type Inference Use Case

Version of August 21, 2023

Lang Feng

Static Analysis Complements Machine Learning: A Type Inference Use Case

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Lang Feng
born in Harbin, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Static Analysis Complements Machine Learning: A Type Inference Use Case

Author: Lang Feng
Student id: 5446511
Email: L.feng-2@student.tudelft.nl

Abstract

Type inference plays a pivotal role in modern software development as it aids in understanding code, detecting errors, and facilitating code completion. Two main approaches, static analysis, and machine learning, contribute to this process. Each approach has its own benefits and limitations. This thesis investigates the potential of combining static analysis techniques and machine learning (ML) approaches to enhance type inference capabilities.

The research initially demonstrates how static analysis and ML complements each other in the context of type inference. It utilizes two static analysis tools, Pyre [9] and Pyright [11], to showcase their effectiveness in inferring types. To enhance type inference, a hybrid approach is proposed, integrating the machine learning approach Type4Py [53] with static analysis techniques. Additionally, a ranking system is considered to determine the order of results within this combined approach. Both the naive approach and learning to rank models are implemented within this ranking system.

Finally, the research evaluates its findings using a newly created dataset, which is an updated version of ManyTypes4Py [52]. The outcomes of this research emphasize the potential for improving type inference by integrating static analysis and ML approaches. Moreover, the evaluation reveals that, in the ranking system, the naive approach proves to be more effective compared to the learning-to-rank models.

Thesis Committee:

Chair:	Dr. G. Gousios, Faculty EEMCS, TU Delft
University supervisor:	Dr. G. Gousios, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft

Preface

I am pleased to introduce my master's thesis titled "Static Analysis Complements Machine Learning: A Type Inference Use Case." This research endeavor was undertaken under the guidance of Professor Georgios Gousios, and with valuable support and supervision from my daily advisor, Amir Mir. The thesis was carried out as a part of my academic program in the Faculty of Electrical Engineering, Mathematics, and Computer Science at TU Delft.

The motivation for selecting this topic arose from my initial meeting with Amir Mir, where he introduced me to the key aspects of type inference and potential approaches in the field. From that point onwards, we worked together closely, and in our weekly meetings, he guided me through the research journey. Additionally, I am extremely grateful for the guidance and support of my supervisor, Professor Georgios Gousios, during the second stage of the research. Despite his busy schedule, he provided me with valuable suggestions and mentorship on a weekly basis. Our weekly meetings served as a platform for productive brainstorming sessions. I am truly fortunate to have received guidance from both of them throughout this research project.

I would like to express my heartfelt appreciation to Committee Member Casper Poulsen for his invaluable contribution as a member of my committee. I am also immensely grateful to my parents for their unwavering support, which has been a constant source of courage and determination throughout my life. Their encouragement and belief in me have been instrumental in shaping my journey. Furthermore, I want to sincerely express my deep appreciation to my girlfriend, Chang Ge, for her unwavering support and constant companionship during my master's studies at TU Delft. Her presence and encouragement have been truly invaluable, and I am profoundly grateful for her unwavering support at every step of this journey. Lastly, I extend my deep gratitude to the Delft University of Technology for fostering an exceptional learning environment and providing invaluable resources. Their commitment to excellence has greatly contributed to my academic growth and development.

Lang Feng
Delft, the Netherlands
August 21, 2023

Contents

List of Figures

Chapter 1

Introduction

An introduction to type systems and dynamic type languages is provided in this section. We continue by outlining the terms *type annotation* and *type language* and providing examples. We also explore type inference's benefits and significance in the context of dynamic type languages. We next go over the current literature on type inference and its drawbacks before outlining our probable strategy. Finally, we state the thesis's guiding research questions.

1.1 Type Inference

Type System and Dynamic Type Language

Each programming language has its own type system, according to the time when it begins type check, program languages could be divided into dynamic type languages and static type languages [45]. For instance, dynamic type languages like Python [37] and Ruby [60], do not perform the data type check until run-time. In other words, when programming in a dynamically typed language, there is no need to assign a data type to any variable; the language records the data type internally when the data is first assigned to the variable. This pattern for variables in dynamic type languages is also consistent for both function parameters and return types [38]. Unlike static type languages where the types for parameters and return values need to be specified in the function signature, dynamic type languages do not require any such specification.

To illustrate the contrast between these language types, consider the following code snippet that demonstrates the difference between *Java* and *Python* in Figure 1.1 and Figure 1.2. The first Java code snippet demonstrates that when defining the sum function, we must specify the data types for both parameters *a* and *b*, as well as for the return value, before compiling the code. Additionally, to use the function, we must specify the data type for the output variable *result*, which is assigned the value returned by the sum function. However, the pattern for *Python*, which is a dynamically typed language, differs from that of *Java*. The first print command outputs 3, whereas the second one outputs 'Hello World'. This illustrates that the sum function in Python does not specify argument types during compilation, but rather determines them during run-time.

Main.class

```java
public class Main {
  public static void main(String[] args) {
    int result = sum(1, 2);
    System.out.println(result);
  }

  public static int sum(int a, int b) {
    return a + b;
  }
}
```

Figure 1.1: Sum function in *Java*. This code snippet shows the implementation of the sum function in Java. In order to use this function, it is necessary to specify the types of both variables, as well as the parameters and return values.

sum.py > ...

```python
def sum(a, b):
    return a + b

print (sum(1,2))
print (sum("Hello ", "Word"))
```

Figure 1.2: Sum function in *python*. No need to specify the types.

Type Annotation and Type Inference

Although we know that type information is not required in dynamically typed languages like Python, we can still include it in the programming code as a hint to the expected data types of variables and function arguments. This can be helpful in facilitating type checking and ensuring successful compilation [55]. For example, we can define type restrictions for input parameters and output results of a Python function to provide additional guidance for the interpreter. The code snippet below is extracted from the basic_type.py file on Github, and it demonstrates the usage of type annotations for a simple code block. Initially, we set the variable *hello* as a string and then assigned the value *"hello word!"* to it. Later on, we specified the expected data types of the input arguments *x* and *y*, and also indicated that the function will return an integer using the *-int* syntax.

In this code block, the data types such as "str" and "int" are defined using ***type annota-***

basic_types.py > ...

```python
hello: str = "hello world!"

def add(x: int, y: int) -> int:
    return x + y

new_val: int = add(7, 4)
```

Figure 1.3: Define Datatypes in Python

tions [54], which are syntaxes introduced in Python3.5 for explicitly indicating the expected data types of variables, arguments, and return values in Python code. In cases where there are no type annotations in the code, we need to infer the data types from other information such as the variable names, context, and environment. This process, automatically determining the data type of a variable or expression is defined as ***type inference*** [47]: an expression E is opposed to a type T, formally written as $E: T$; *type inference* is to solve the $E: \_?$ problem, in other words, only the expression is known and try to derive a type for E. For example, in the code implementation in *fib.py* file in Figure 1.4, the parameter types and return type are inferred, demonstrating the type inference process.

```python
def fib(n):
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

```python
def fib(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b
```

Figure 1.4: Type Inference for Function Signature

Significance of Type Inference

As discussed previously, dynamic-type languages do not impose any restrictions on the types of expressions during their definition. However, this can lead to various problems and inconveniences [57]. Firstly, there is a higher likelihood of bugs and compilation issues since variables and function results are not explicitly typed, unlike static-type languages. The absence of type checking can lead to more bugs, which, in turn, may result in longer debugging times and delays in the CICD process [59]. Secondly, the lack of type definition makes it harder for developers to understand the code and can pose issues during code refactoring [32]. The absence of type structure definitions makes it challenging for developers to change and update the code when they take over a project since they cannot rely on the internal structure of data (i.e., variables).

Therefore, the significance of type inference lies in the following three aspects:

- improve code performance: with inferred data types, the compiler could optimize the generated machine code [39]. In particular for complex systems that require a lot of data processing, this can lead to faster and more effective code execution.
- improve code quality and maintainability: inferred types can save developers time and effort in reading, debugging, and testing, as well as assist avoid bugs from being introduced into production [23].
- lessen the need for explicit type annotations: by automatically determining the data type of variables and expressions based on the context in which they are used, programmers may create more clear, succinct, and type-safe code.

1.2 Current research for Type Inference

A summary of current mainstream type inference methods and their limitations are presented in this section and will be further discussed in the subsequent Chapter 2.

1.2.1 Mainstream implementations of type inference

The current mainstream implementations of type inference can be broadly classified into three categories: static analysis, machine learning, and hybrid approaches. ***Static analysis*** for type inference involves analyzing code without executing it, and inferring types by looking at the structure and syntax of the code [21]. This can be achieved using various techniques such as abstract interpretation, constraint-based analysis, or type rule inference [31]. Current technologies that use static analysis for type inference include *Mypy* [46], *Pyre* [9], *Scalpel* [13], and others. There are also several data-driven approaches that use ***machine learning methods*** to learn patterns in code and deduce types. For instance, Type4Py [53] employs a hierarchical neural network and nearest neighbor search, while DiverseTyper [42] uses a pre-trained TypeBert [41] model and multi-task deep learning approach to infer types in TypeScript [15]. A recently emerged trend, which could be referred to as ***hybrid approach***, combines static approaches with machine learning technologies. For instance, HiTyper [55] and TypeT5 [62] utilize static analysis to identify useful information, which is then fed into a deep model for pattern learning. This approach results in increased accuracy and better performance compared to using single machine learning. approach [55].

1.2.2 Limitations and Challenges of Type Inference Research

We previously discussed various approaches for type inference. Machine learning and hybrid approaches typically outperform current static analysis tools. Despite the advantages of machine learning, type inference still faces two major problems.

The first problem is the ***rare types issue***, which referred to types that occurred in a limited times in the overall dataset, for example, user-defined types. Machine learning approaches tend to exhibit a significant drop in accuracy when dealing with such types, in comparison to their overall performance. Due to their infrequency, these types cannot be effectively predicted based on the learned features. In Figure 1.5, we could see even for the

current state of art machine learning technology, the accuracy on rare types is still very low, lower than 20%.

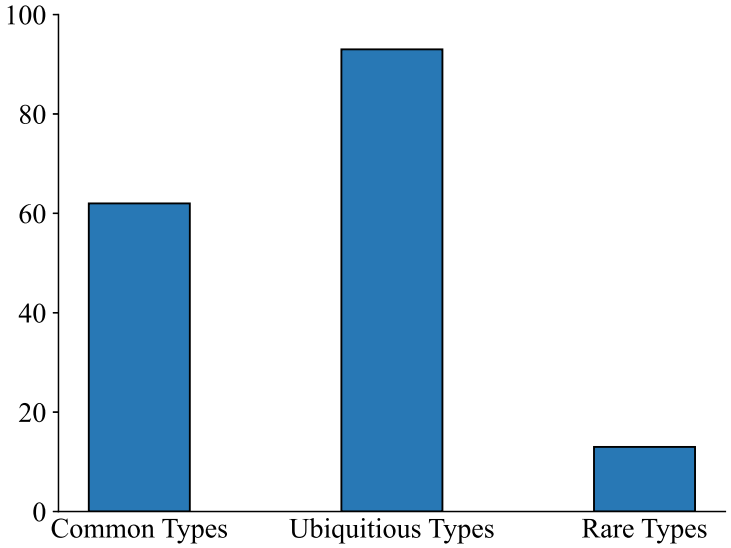


Figure 1.5: Performance on Rare Types for Type4py

The second problem is the ***cross-domain problem***, as highlighted by Gruner [35]. When training a model in one domain and testing it in another, a discrepancy arises that can result in a 10-20% decrease in accuracy. In real-world scenarios, this can lead to decreased performance because it is impossible to ensure that the domain of the project has been seen during the model training process.

1.3 Approach Overview

To solve the issue of rare types and enhance the model's cross-domain capabilities, we suggest a hybrid strategy that blends machine learning and static analysis methods. We intend to incorporate the precise predictions from static analysis on rare types into our final hybrid predictions, in contrast to existing hybrid systems that only use static analysis to extract important information for the neural network. For the static analysis phase, we recommend utilizing two tools: Pyre [9] and Pyright [11]. As for the machine learning component, we suggest employing the Type4Py model and training it on our dataset through a project-based approach. Due to the Type4Py model is based on the dataset from 3 years ago [52], we may first start with training that on a new version of the dataset. After that, we will start evaluating the Type4Py model as well as static analysis on the test projects.

Furthermore, given that our hybrid approach involves two models, it is possible that different models may yield different results. In light of this, we propose implementing a ranking system to establish an order of preference. To accomplish this, we have selected a learning-to-rank approach as the primary component of our ranking system. We will train

this model independently and integrate it into our hybrid approach to evaluate its effectiveness. The overall structure of our approach is shown in Figure 1.6.

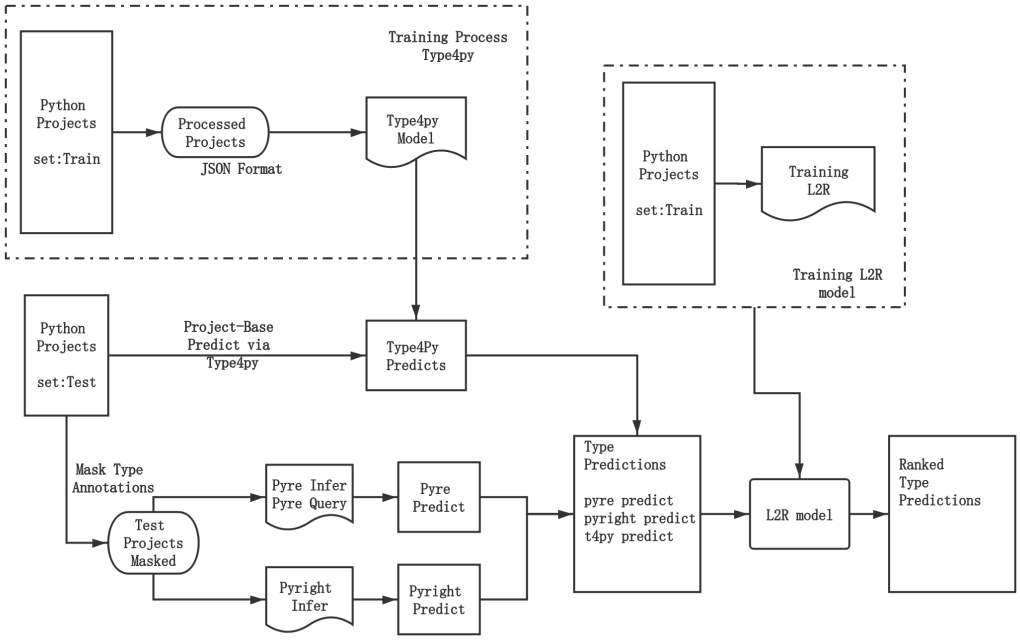


Figure 1.6: Approach Overview

1.4 Research Questions

In light of the existing research on type inference that has already been done, we created a hybrid strategy that combines static analysis and machine learning models, as described in the previous section. Our thesis seeks to explore the following research questions for this approach:

RQ1: What is the general performance of this hybrid approach?

We suggest a hybrid strategy meant to capitalize on both of their advantages for static analysis and machine learning. We intend to contrast our approach's performance with that of several baseline models in order to judge its effectiveness. We conduct this to see if the hybrid strategy can outperform other state-of-art techniques in terms of overall performance.

As previously mentioned, type inference tasks typically involve determining the types of variables, parameters, and return values. We are interested in examining potential distinctions between these tasks; for example, we might look into whether our suggested method performs better at determining variable types than the other two tasks. To this end, we plan to analyze the accuracy of our approach for each of these tasks, in order to gain insight into their relative strengths and weaknesses.

Moreover, we aim to determine how much the performance of our suggested hybrid strategy benefits from static analysis and whether it can address some of the drawbacks and restrictions of type inference based on machine learning. We will investigate how static analysis can assist in overcoming the limitations of machine learning-based type inference.

RQ2: How to rank among static analysis and machine learning result in hybrid approach?

In the hybrid approach, which incorporates both static analysis and machine learning results, running on a single type slot may yield different outcomes. For instance, consider the expression:

$$x = 3.0$$

static analysis may give us a result as x : '$float$', while machine learning may give us a result as x : 'int', it is essential to establish the correct answer order and provide recommendations between these two options. To address this challenge, we have implemented a ranking system within our hybrid approach. This system takes into account the varying results obtained from different static analysis tools, as well as the list of predictions generated by the machine learning component. By employing a learning-to-rank model, we have trained several neural networks to solve this ranking problem.

The structure of our methodology, which includes the implementation details of the ranking system, will be discussed in Chapter 3. Furthermore, we will evaluate the performance of our approach in Chapter 4.

RQ3: What is the scalability and time efficiency of this approach?

In addition to exploring the theoretical potential of our proposed hybrid approach, we also aim to assess its practical viability. Specifically, we will investigate whether our approach can be effectively applied to large-scale projects with numerous files. In order to do this, we will analyze the time required to perform type inference using our hybrid approach and compare it to other existing approaches. In order to assess the practical viability of our approach in real-world circumstances, we will compute the inference time for each type slot.

Chapter 2

Related Work

The core focus of our research centers around two key areas: *type inference* and *learning-to-rank problem*. Within these domains, we have implemented multiple type inference approaches and employed a learning-to-rank model to establish an order among them. This chapter serves to discuss the various technologies associated with these fields and their significance in our research.

2.1 Type Inference

As mentioned in Chapter 1, we discussed the current landscape of type inference approaches, highlighting three main streams: static analysis, machine learning, and hybrid approaches. The static analysis relies on basic types and existing type annotations within the file, performing recursive analysis to infer unknown types [11]. Machine learning approaches leverage program context, such as program names and patterns, as input to predict the most likely results [22]. The hybrid approach combines static analysis with techniques like user-usee trees to extract valuable information, which is then inputted into a neural network for inference [55] [62]. These approaches represent the state-of-the-art methodologies used in type inference, and their exploration forms a significant part of this research.

2.1.1 Static Type Inference

Static type inference involves utilizing static type analysis techniques to deduce unknown types without the need for code execution. Prominent static analysis tools in the field include Mypy [6], Pyre [9], Pyright [11], Pytype [12], Scapel [13], among others. These tools are capable of analyzing the program context and inferring types based on their recognition capabilities. By leveraging static type inference, researchers and developers can enhance code understanding, identify potential errors, and improve overall program correctness.

Static Analysis

At the core of static type inference lies static analysis, which involves examining code without its execution. During the static analysis process, specialized tools are employed to

detect potential errors, identify violations of coding standards, and attempt to infer missing type annotations.

Consider the example of Pyre: when Pyre is applied to a program, it initiates static analysis on the program's foundation. It scrutinizes variable definitions, establishes call-callee relationships, and generates reports highlighting any issues encountered. These reports may include findings such as import cycles, unused variables, and missing type annotations.

Static Type Inference

We will take advantage of static analysis for type inference, type inference includes three tasks, variables, parameters as well as return types.

For **variables**, the static analysis tool will infer its type based on its variable assignment within the scope, either *built-in*, or the *class* or *project* scope.

code_snippet.py

```
1   var1 = [p for p in [1, 2, 3]]
2   # var1: `list[int]`
3
4   class Foo:
5       def __init__(self):
6           self.var = ""
7
8       def do_something(self, val: int):
9           self.var = val
10
11  if __debug__:
12      var2 = None
13  else:
14      var2 = Foo()
15  # var2: `Foo | None`
```

Figure 2.1: Example Code Snippet for Variables in Python

In Figure 2.1, when analyzing this code snippet, the static analysis tool will infer the **var1** as '*list* [*int*]', this inferred type comes from the type of the source expression for this variable. In the case of **var2**, which is assigned in two places, in line 10, and line 12, and refers to different types, therefore, it will be a union of these types. In line 10, it is easy for static analysis to infer it as '*None*', and in line 12, it will refer to the callee class *Foo* in line 2, which finally gives the inference '*Foo* | *None*'. This is just a simple example, however, when it comes to more complex definition expressions including user-defined types and parametric types, static analysis tools will not be able to infer the correct types and in some cases, it will give an '*Any*' type instead.

For **parameters**, in most cases, it will be hard for static analysis to infer, for example, Pyright will infer the types of parameters in two cases: 1) if this is a function in an instance and the corresponding function in base class has the annotation, the static analyzer will infer the types correspondingly via *inherit* [11]. 2) if there is a default value for the parameter, the static analyzer could infer the type by its assignment.

For **return types**, types can be inferred from the return statements found within that function, all the types of return statements found in the function will gather a *Union* as the overall return type. For example, in Figure 2.2, when analyzing **func1**, static analyzer will recognize "*empty_str*", '*True*' as '*str*' and '*bool*', and the inferred return type will be '*str* | *bool* | *None*'. However, when it comes to functions with parameters unannotated, it will become harder for the static analyzer to infer, for example in **func2**, there is no annotation for either *a*, *b* or *c*, in this case, the inferred return type could only be '*Unknown* | *None*'.

code_snippet1.py

```
1  # func1: `str | bool | None`
2  def func1(val: int):
3      if val > 3:
4          return ""
5      elif val < 1:
6          return True
7
8  # func2: `Unknown | None`
9  def func2(a, b, c):
10     if c:
11         return a
12     elif c > 3:
13         return b
14     else:
15         return None
```

Figure 2.2: Example Code Snippet for Functions in Python

Static analysis for type inference offers certain advantages. It excels in inferring correct types for assigning expressions within simple built-in scopes. Furthermore, it can also recognize and accurately infer user-defined types in class scopes, as demonstrated by the example of *Foo* mentioned earlier. However, static analysis may face limitations when dealing with code that lacks sufficient type annotations, for example, inferring the return types without the annotations of parameters. In such cases, the inference process may not yield the desired results, as the absence of explicit type information makes it challenging for static analysis to deduce the types.

2.1.2 Machine Learning Techniques for Type Inference

In the realm of type inference, machine-learning approaches have also gained prominence. Two notable ML-based approaches for type inference are Typilus [22] and Type4Py [53]. These approaches utilize neural networks to encode the code context into a high-dimensional space. Subsequently, they employ K-nearest neighbors (KNN) search to perform type inference.

Typilus, released in 2020, emerged as the cutting-edge machine learning technique in this field. This approach employs a Graph Neural Network (GNN) [44] [49] to map type information to a high-dimensional type space. To enhance the learning process, Typilus

utilizes a variant triplet loss [29] as the loss function in the GNN network. By calculating the Euclidean distance between type slots, Typilus performs a K-nearest neighbors search. It predicts the types that are closest to the ones observed in the training set. This process enables Typilus to make accurate type predictions based on the proximity of the type slots in the learned type space.

Another state-of-the-art machine learning approach for Python type inference is Type4Py, which was introduced in 2022. Similar to Typilus, Type4Py is developed using a deep similarity learning-based hierarchical neural network architecture. Type4Py employs various components to make accurate type predictions, including the extraction of *identifiers*, *code context*, and *visible type hints* for each type slot. The extracted tokens are then passed through an embedding model, which maps them to a high-dimensional space (256-dimensional). To accomplish this, Type4Py employs a hierarchical neural network (HNN) consisting of two LSTM-based recurrent neural network (RNN) models. After the tokens are embedded, a linear layer is applied to the resulting 256-dimensional embeddings. The final step involves utilizing a K-nearest neighbors (KNN) search, where the machine learning model compares the embeddings to similar examples in the training data. Based on this comparison, Type4Py outputs its final prediction for the type of the given code snippet. The overall structure is shown below in Figure 2.3.

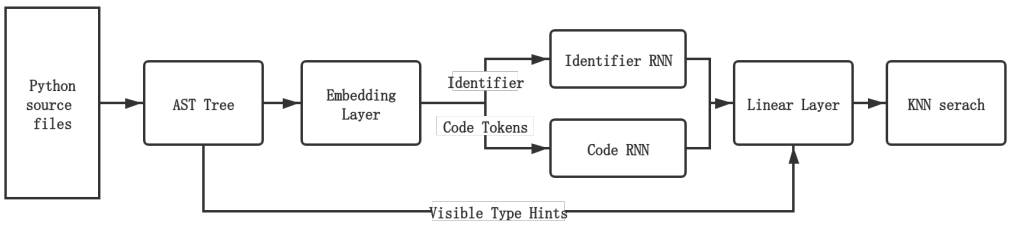


Figure 2.3: Architecture of Type4Py Type Inference System

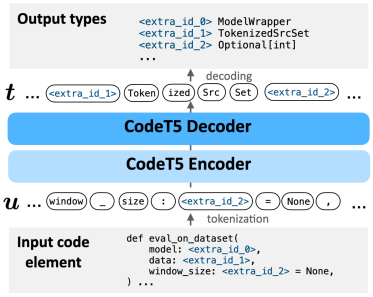
According to the results, Type4Py not only outperforms the traditional static type inference techniques but also outperforms other machine learning techniques including the GNN-based model, Typilus. Moreover, for the machine learning approaches, it could output top-n predictions, which will be helpful in the actual practice compared to static analysis.

2.1.3 Hybrid Approach for Type Inference

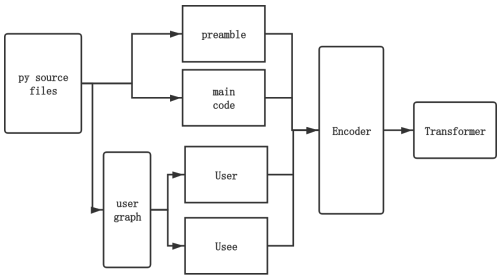
Recently, there have been hybrid approaches that combine both static analysis and machine learning techniques for Python type inference. Two notable examples are HiTyper [55] and TypeT5 [62]. These approaches utilize static analysis tools to extract relevant and valuable information, which can provide better contexts for the feature learning process. The extracted features are then fed into a neural network for learning and prediction. By combining static analysis and machine learning, HiTyper and TypeT5 aim to leverage the strengths of both approaches to enhance the accuracy of type inference in Python.

Hityper, released in 2022, is a hybrid type inference approach based on the *Type Dependency Graph(TDG)* [55]. The TDG is a graph that records the type dependencies between

variables in the function scope. The type prediction task is changed into a graph-filling task by Hityper using this graph. In order to fill in the types in the graph, Hityper leverages both static analysis techniques and deep learning models. To ensure accurate predictions, Hityper builds a series of *type rejection rules*. These rules help eliminate incorrect type predictions by considering factors such as conflicting type information and semantic inconsistencies. By combining static analysis, deep learning, and type rejection rules, Hityper strives to provide precise type inference results.



(a) Type Inference as Code Infilling Task



(b) TypeT5 Architecture

Figure 2.4: TypeT5: Seq-2seq Type Inference using Static Analysis

TypeT5 [62], built upon the foundation of CodeT5 [61] with improvements designed for type inference, is considered one of the most accurate and powerful models available. TypeT5 leverages the transformer architecture commonly used in sequence-to-sequence models. By transforming the type inference task into a code-infilling task in Figure 2.4a, TypeT5 capitalizes on the strengths of the transformer architecture. In addition, TypeT5 recognizes that type inference often requires non-local information beyond the immediate code context, extending to the entire file or even external dependencies. To address this, TypeT5 utilizes static analysis techniques to construct a comprehensive *usage graph*, as shown in Figure 2.4b. This graph stores valuable information from both within and outside the surrounding code, enabling TypeT5 to extract relevant details about potential users and usees. This wealth of information contributes to more accurate type inference results.

With its combination of the Transformer architecture and the usage graph, TypeT5 demonstrates remarkable precision in type inference, making it a leading model in the field.

2.2 Rank system: Learning to Rank

As discussed in the preceding section, the hybrid approach incorporates outputs from both static analysis and deep learning techniques. Consequently, there may arise situations where the two models yield different results. To establish a correct ranking among these predictions and ensure proper ordering, the implementation of a rank system becomes necessary. In this regard, we propose employing a *learning-to-rank* model as the core component of this ranking system. In the following subsections, we will introduce the concepts related to learning to rank and discuss the techniques that we may employ in our approach.

2.2.1 Ranking Model and learning to Rank

Ranking models are widely used to sort objects based on their relevance, preference, or importance. This task is crucial in information retrieval and is encountered in various domains [50]. For instance, in search engines, the goal is to present a sorted list of web pages based on their relevance to a given query. Similarly, in recommendation systems, the aim is to provide a list of potentially interesting products based on user profiles and order histories [43].

Ranking models typically work by predicting a ***relevance score***, denoted as:

$$s = f(x)$$

for each input $x = (q,d)$, where q represents a ***query*** and d represents a ***document***. The relevancy score reflects how closely the query and the document are related. By obtaining the relevance scores for all documents, we can then sort, or rank, the documents based on these scores. This ranking process enables us to prioritize and present the documents in an order that reflects their relevance to the given query. To compute the ***relevance score S***, we will employ the learning-to-rank approach. This methodology involves training a model to predict a score s when provided with an input $x = (q,d)$ during the training phase. The learning-to-rank model learns the underlying patterns and relationships between the *query:* q and the *document:* d to estimate their relevance score. By optimizing the model through training, it becomes capable of accurately predicting the relevance score for new inputs. This approach enables us to effectively rank and prioritize documents based on their relevance to a given query.

We can also make use of the learning-to-rank approach in the context of our type inference ranking system. We provide the model with a query that represents the code context and anticipate that it will produce an ordered list of predictions from several models. This enables us to meaningfully and effectively rate and recommend the type predictions.

2.2.2 ML models in Learning to Rank

To construct a machine learning model for calculating relevance, we need to define the ***loss function***. Considering the input and output mentioned earlier, the remaining aspect to determine is the appropriate loss function. In the field of learning to rank, there are three primary categories of loss functions based on different approaches: point-wise, pair-wise, and list-wise [63] [27]. As shown in Figure 2.5, given a query and a list of documents:

- Pointwise: In this method, the L2R model treats ranking as a regression or classification problem. The loss function is typically defined based on the discrepancy between the predicted relevance score S_i and the ground truth relevance label y_i for each individual instance $x_i = (q,d_i)$.

- Pairwise: In this method, instances are compared in pairs $x_i = (q,d_i)$, $x_j = (q,d_j)$ with the goal of learning a model that appropriately ranks instances inside each pair. The loss function calculates the discrepancy between the actual pair order and the

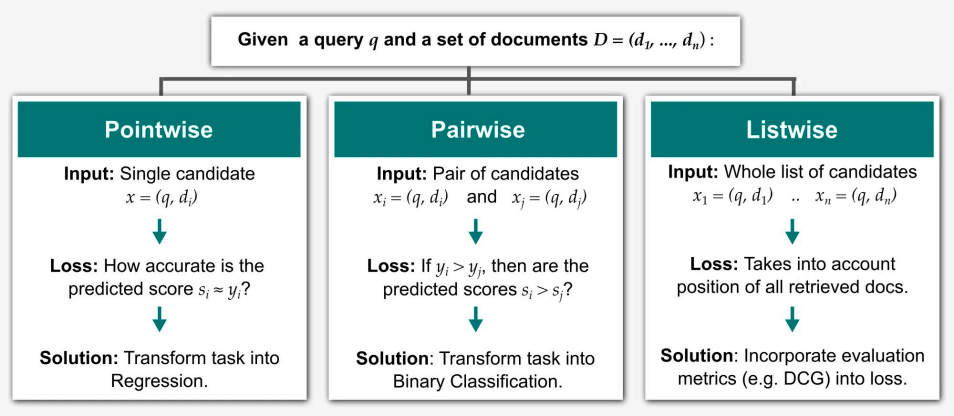


Figure 2.5: Machine Learning Approaches on Learning to Rank: Pointwise, Pairwise, Listwise [3].

expected pair order. This method is used in RankNet [24] and lambdaMART [26], which is an improvement compared to the pointwise approach.

- Listwise: In this method, ML models consider the entire list of instances and directly optimize a ranking metric, such as NDCG (Normalized Discounted Cumulative Gain) [40] or MAP (Mean Average Precision) [27]. The loss function is designed to directly optimize the ranking performance on the entire list. This method is used on LambdaRank [25] model and has better results compared to pointwise and pairwise.

After considering the different approaches, we have decided to adopt the point-wise approach for its simplicity in embedding the query and training process. By abstracting the problem as a classification task, we can leverage various techniques such as XGBoost [28] and neural networks. The specific details regarding the problem abstraction and implementation will be elaborated in Chapter 3.

Chapter 3

Methodology

This chapter outlines the methodology employed in our research. We begin by discussing the utilization of static analysis and the specific tools employed to perform type inference on a project-based level. Subsequently, we introduce the Type4Py model and elaborate on the enhancements made to the training process. Additionally, we describe the integration of static analysis results and the ML model, showcasing our hybrid approach. We further outline the pipeline employed in the project-based context. Finally, we introduce our ranking system and discuss various approaches that were explored within this system.

3.1 Static Type Inference

This section provides an overview of our approach to utilizing static analysis tools for performing type inference in Python projects. The process consists of several key steps: *type extraction*, *annotation removal*, *static type analysis*, and *inference extraction*. These steps primarily rely on the Abstract Syntax Tree (AST) and operating system (OS) commands. In the subsequent subsections, we will delve into each step in detail, explaining their importance and methodology. The overall structure is shown in the following Figure 3.1.

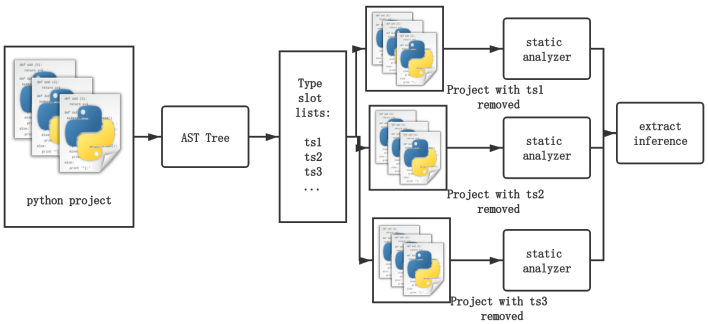


Figure 3.1: Static Analysis Pipeline for Type Inference

3.1.1 Type Extraction

The first step in the static inference pipeline is to extract all the annotated types in the Python project to collect ground truth and type inference. In this step, we will mainly use LibCST [5] for type slot extraction. The procedure is shown in the following Algorithm 1.

Algorithm 1 Type Extraction Algorithm

```
1:  procedure A PYTHON PROJECT(project_test)
2:      typeslots ← []
3:      for all python file: file in project_test do
4:          typeslots_file ← []
5:          parsed_model = cst.parse_module(file)
6:          for all node ∈ parsed_model and node is not visited do
7:              if node.type == cst.FuncDef then
8:                  # Extract Return Type
9:                  if node.return.annotation is not None then
10:                     typeslot = (node.return.annotation, node.loc)
11:                     typeslots_file.add(typeslot)
12:                 end if
13:                 # Extract Parameters
14:                 for all param in node.params do
15:                     if param.annotation is not None then
16:                         typeslot = (param.annotation, param.name, node.loc)
17:                         typeslots_file.add(typeslot)
18:                     end if
19:                 end for
20:             end if
21:             # Extract Variables
22:             if node.type == cst.AnnAssign then
23:                 typeslot = (node.annotation, node.name, node.loc)
24:                 typeslots_file.add(typeslot)
25:             end if
26:         end for
27:         typeslots ← typeslots ∪ typeslots_file
28:     end for
29:     return typeslots
30: end procedure
```

Input: A python project contains a list of files: *project_test*
Output: A list of type slots extracted from the files: *typeslots*

For each file in the Python project, we will utilize the builtin function: *cst_parser* [2] to parse the program as a CST (Concrete Syntax Tree) module [1]. Within the module, we will visit *funcDef* and *AnnAssign* nodes to extract the return types of all functions, function parameters, and variables. Furthermore, to proceed with the next step of locating the node,

removing the annotation, and inferring the types, we need to store the location information within each type slot in the project-level *typeslot_list*. Each type slot in this list will appear like the following in Figure 3.2: For the tasks for *variables* and *parameters* the *name* key

Figure 3.2: Example Type Slot after Type Extraction

will refer to the name of that variable and parameter.

3.1.2 Annotation Removal

After extracting all the type slots within the entire project, the next step is to remove the type annotations and perform static analysis to infer the correct types for those specific type slots. We can utilize the LibCST [5] package for this task, similar to the previous step. However, instead of visiting each node, we will make changes to the specific nodes that require processing with *with_change* method [18]. In each iteration, we will make changes to a single node within one specific file, while keeping all other type slots and files unchanged. The three types of changes related to return types, parameters, and variables will be as follows:

- For **Return Types**, consider the function node as *node*, what we just need to do is change the annotation in *node.return* to *None*:

$$return\ node.with\_changes(node.return.annotation = None)$$

- For **parameters**, the objective is to remove the specific annotation for one parameter at a time while keeping all other parameters and the return expression unchanged. To achieve this, we will create a new parameter list within the $funcDef$ node. The modified $funcDef$ node with the updated parameter list will be returned. The process is shown in the following Algorithm 2

Algorithm 2 Remove Parameter Algorithm

Input1: Type slot of specific param: $type\_slot$
Input2: The module of function which contains that param: $func\_node$
Output: The updated module of function which removes the annotation of that param: $func\_node_{update}$

```
Initialize updated_param_list ← []
for all param in func_node.params do
    if param.name != type_slot.name then
        updated_param_list.add(param)
    else
        new_param = param.with_changes(param.annotation = None)
        updated_param_list.add(new_param)
    end if
end for
func_node_update = func_node.with_changes(params = updated_param_list)
return func_node_update
```

- For **Variables**, when encountering an annotated assignment *node*, we need to transform the *node* from *Annotated Assignment* into *Assignment* type, while keeping the target expression unchanged. The transformation can be performed as follows:

$$return\ cst.Assign(target = node.target)$$

3.1.3 Static Type Inference

Once we have the project with only the targeted type slots masked, we can utilize static analysis tools to perform type inference. In our experiments, we explore the use of two static analysis tools: Pyre [9] and Pyright [11], for static type inference. The underlying strategy is similar for both tools. We use their command-line interface to generate type stubs, which display the types recognized by the tools in the stub files (.pyi). Subsequently, we parse the .pyi files again and extract the relevant types using the *libcst* transformer.

First, we will first introduce the pipeline for Pyre static type inference. We utilize the command tool $pyre\ infer$ [10] for this purpose. The process begins by starting the Pyre server within the project base. Then, we use a subprocess to execute the command tool and generate the type stubs. The pipeline for running the Pyre server is illustrated in Figure 3.3. In order to perform pyre analysis, it is necessary to have the Watchman server [17]. Therefore, we must initialize and start the watchman server beforehand in every project's base.

Moreover, we experimented with another static analysis tool called Pyright, which is considerably more straightforward compared to Pyre. To utilize Pyright, we will employ the $pyright\ --createstub$ command in the operating system [20]. However, prior to executing this command, we need to install the project using the *pip install* command within the virtual environment. Once this command is executed, Pyright will generate type stubs

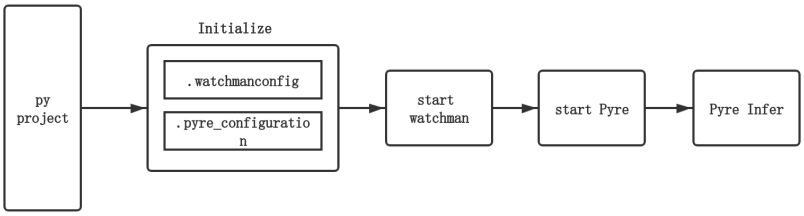


Figure 3.3: Pyre Static Analysis Process

within the Python project, encompassing all the recognized types. We will then examine the type stub folder and extract the type inference from it.

3.1.4 Inference Extraction

During this phase, we will carry out the last step, which involves extracting the relevant type inference from the type stub file generated during the static type analysis. The following two steps will be used to extract this inference:

- First, we will verify the existence of a type stub file that corresponds to the source file containing the target type slot. The type stub file will reside in the same relative path but with a filename suffix of ".pyi".
- Next, upon locating the type stub file, we will parse it and utilize the Concrete Syntax Tree (CST) to navigate and identify the corresponding type slot.
- Finally, we will add the static type inference we extracted from the type stub file into the type slot in Figure 3.2, with a key as *pyre_result* or *pyright_result*

Following the completion of the preceding four steps, our goal is to obtain a type slot list for each project. Each type slot will consist the following components: human annotation as the label, Pyre and Pyright results as type inferences, the corresponding source filename, and its location. By utilizing these type slot lists, we can evaluate the match rate individually, and they can also be employed in subsequent experiments that involve combining machine learning approaches and ranking systems.

3.2 Machine Learning Approach

To implement the machine learning approach for type inference, we primarily build upon the Type4Py [53] framework and introduce several minor enhancements. These enhancements aim to adapt the training process to larger datasets and enable the handling of project-based type inference tasks. The first section of our implementation focuses on introducing the baseline functionality of Type4Py, while the second subsection delves into the details of our specific enhancements.

3.2.1 Type4Py Pipeline

The baseline implementation of Type4Py consists of three main phases: *feature extraction*, *embedding*, and the *learning process*. In the feature extraction step, Type4Py selects three components as features for each type slot prediction: natural language information, code context, and type hints [53]. These features are then passed through the embedding phase, where a Word2Vec [51] model is utilized to convert them into embeddings. This process vectorizes the input features, allowing models to learn from them effectively. The resulting vectors are subsequently fed into the core model. Within this model, both code tokens and identifier tokens are inputted into separate recurrent neural networks (RNNs) with LSTM units [34] to generate vectors. These vectors are then combined with visible type hints and passed through a linear layer. This transformation sets up the type inference task as a K-nearest neighbors (KNN) [56] search task.

Feature Selection & Embedding

As previously mentioned, the ML approach selects three features for each type slot as inputs for the inference model. The first feature is ***natural information***, which varies depending on the specific task. For different tasks, it refers to:

- Variables: variable name N_{var}
- Parameters: parameter name N_{arg} + function name N_{func} + the remaining parameter names N_{arg0}, N_{arg1}...
- Return types: function name N_{func} + parameter names N_{arg0}, N_{arg1}...

The extraction of these names is a straightforward process using the Abstract Syntax Tree (AST). Once the names are extracted, they are then passed as input to a Word2Vec model, which generates vector representations as N_{var}, N_{arg}, N_{func}. As for parameters and return types, the concatenated vector length is limited to 1000 dimensions, where any excess beyond this length is omitted and not used as input for the model.

The second feature we choose to utilize as input is the ***code context***. This includes the code expressions corresponding to each type slot. We feed the selected code snippets into the code Word2Vec model and concatenate the resulting output vectors based on different tasks, as outlined below:

- Variables: All the occurrences for the variable within the scope $\{C_{var}, C_{var}, C_{var}...\}$
- Parameters: All the occurrences for the parameter within the function scope $\{C_{param}, C_{param}...\}$
- Return types: All the return expressions: $\{Return_1, Return_2\}$

We will consider the third feature as ***visible type hints***, which encompass all the imports present in the source project. This information can be helpful in providing hints to an ML model about the potential types that might occur and be used within the code context.

During the embedding process, for each project, the visible type hints can be represented as a binary vector consisting of 0s and 1s. 0 indicates that a particular type has not occurred in the project, while 1 indicates the presence and location of the type within the project in relation to the overall type libraries.

Neural Model

Once we have obtained the input vector for each type, we will feed the vector into a neural model for deep similarity learning. This model consists of a hierarchical neural network (HNN) comprising a code RNN and an identifier RNN. Additionally, we employ a triplet model to calculate the triplet loss in the type space as shown in Figure 3.4. The triplet model

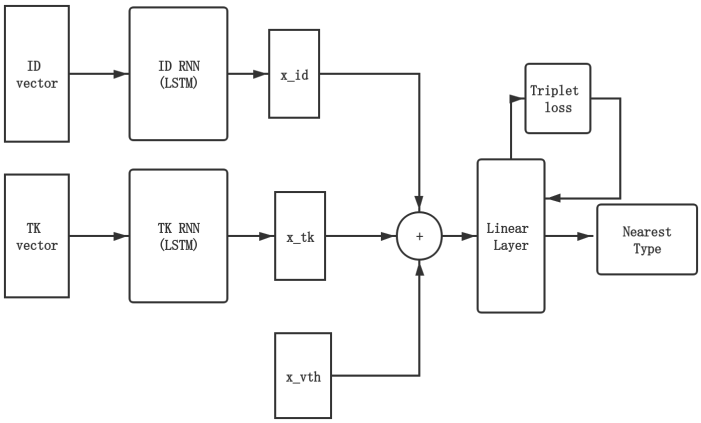


Figure 3.4: Type4Py Model for Deep Similarity Learning

is designed to learn a mapping that makes the results of type A closer together compared to results from a different type B. For example, in Figure 3.5, let's consider examples A and B, both of which belong to the "str" type. In this case, we desire that A and B are closer to each other in the embedding space, as compared to example C, which belongs to the "int" type. In the implementation, the RNNs are implemented using the LSTM [4] package in PyTorch. Additionally, the triplet loss can be calculated using the TripletMarginLoss [14] module available in PyTorch.

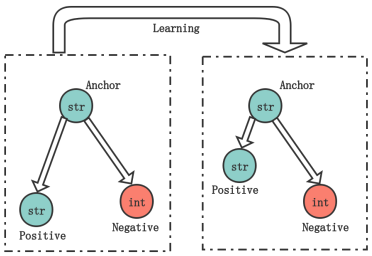


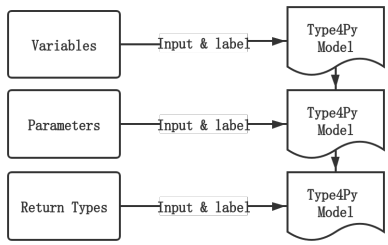
Figure 3.5: Triplet Loss for Types

Consequently, in the prediction phase, when presented with a new example, the trained model will be able to predict the closest type cluster based on the learned mappings.

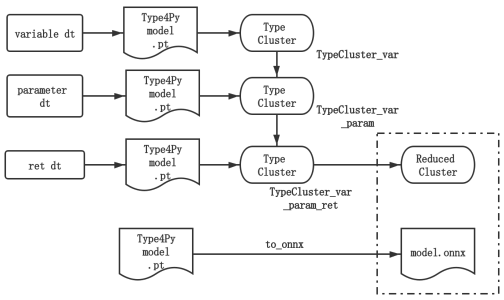
3.2.2 Enhancement on Training for Larger Dataset

We have added a modification to the Type4Py model that was initially discussed earlier in order to handle larger datasets more effectively. We found that it is not possible to load the complete dataset for training, as we observed while working with large datasets (which will be discussed in more detail in Chapter 4). Consequently, we have devised a strategy as ***Sequential Training***: to train the model sequentially on three different datatypes: variables, parameters, and return types as shown in Figure 3.6a. With the help of this sequential training method, we can overcome the difficulties larger datasets present. We can efficiently analyze and train on subsets of the data at a time by separating the training process into these three different data types. While taking into account the limitations of larger datasets, this approach enhances the manageability and scalability of training.

Furthermore, during the training phase, it is necessary to save type clusters for KNN (K-Nearest Neighbors) search in the prediction phase. Considering our sequential training strategy, we load and save type clusters accordingly. Additionally, we employ a reduced type cluster with fewer dimensions, reducing it from 1024 to 256. Moreover, to facilitate prediction, we convert our Type4Py model to ONNX [7] format. This allows for efficient and optimized inference during the prediction phase. The training process and outputs are shown in Figure 3.6b.



(a) Train the Model Sequentially on Data Types



(b) Training Output: Reduced Type Cluster, Model.onnx

Figure 3.6: Enhancement on Training for Larger Dataset

3.3 Integration on Project-Base

In the preceding sections, we discussed the utilization of static analysis and ML approaches for type inference. However, the results obtained from these different approaches may be present in separate files or formats. To obtain a consolidated type list that includes hybrid results for each project, we will now provide additional details on how we achieve project-based type inference.

3.3.1 Implement Type4Py model on Project-base

Once we have obtained the *type cluster* and *Type4Py model* in the previous step, we can utilize them for project-level type inference. This can be achieved using the libsa4py [52] tool, which facilitates the extraction of relevant information from the projects.

For each project, libsa4py generates a JSON-like file containing comprehensive information about the types. This JSON file includes detailed information about variables, parameters, return types, and their corresponding locations within the code. In the case of variables, which can be found in files, classes, and functions, the ML model utilizes both the variable name and its occurrences as input. The model then adds the predicted types to the JSON file accordingly. For each variable, the model generates a list containing the predicted types, represented as var_p. Similarly, for parameters and return types occurring in functions, the extracted information is fed into the model, and the predicted types are outputted as $param_p$ and $return\_type_p$ respectively.

Subsequently, we clean up unnecessary information in the JSON file, retaining only the type slots consisting of the original type, predictions, and locations in each file in the project. As a result, the ML approach generates a type list file that contains all the type slots with ML predictions.

3.3.2 Merge Results as Type List

After completing the previous step, for each project, we obtain a type slot list that includes annotated types along with predictions from both static analysis and the ML approach (Type4Py). Using the location information within the file, we can merge these results together, resulting in a consolidated output as depicted in Figure 3.7. In this example types lot output, the term *original_type* corresponds to the human annotations, serving as the ground truth for comparison. The *pyre_predict* field represents the predictions generated by the static analysis tool Pyre, while *pyright_predict* represents the predictions from another static analysis tool, Pyright. Additionally, the *t4py_predictions* field contains a list of type predictions obtained using the Type4Py approach.

3.4 Ranking System

Having obtained results from both the ML approach and static analysis, it is desirable to establish an ordering among these approaches for ranking purposes. Initially, a straightforward approach involves placing the static analysis results above the ML approach results. Moreover, we also aim to explore the use of a learning-to-rank model to determine if we can further improve the ranking process.

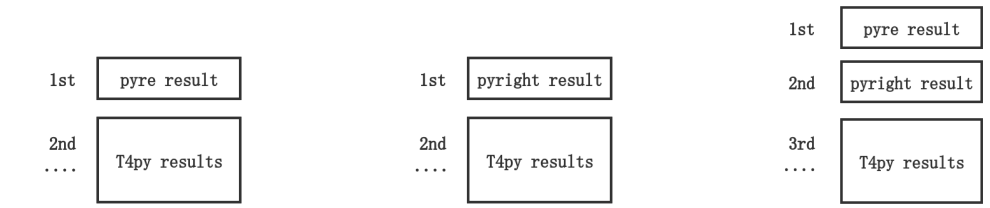
3.4.1 Naive Approach

Now, let's consider the results obtained from both the static analysis and ML approach. Initially, we can adopt a straightforward approach where the results from the static analysis are placed above the results from the ML approach. This is based on the intuition that

```
{
    "original_type": "str",
    "pyre_predict": "str",
    "pyright_predict": "str",
    "t4py_predictions": [
        [
            "Dict[str, Any]",
            0.5920554500729136
        ],
        [
            "Dict[str, Dict[str, Any]]",
            0.2096957774505871
        ],
        [
            "Optional[popxl.dtypes.dtype]",
            0.10095022639515874
        ],
        [
            "Dict[str, List[Any]]",
            0.09729854608134055
        ]
    ],
    "dt": "param",
    "is_parametric": false
}
```


Figure 3.7: Hybrid Result for an Example Type Slot

static analysis is generally believed to be more accurate compared to machine learning approaches. Therefore, when dealing with the results from Pyre, Pyright, and Type4Py, the ranking order will be as follows in Figure 3.8:



(a) Pyre & Type4Py (b) Pyright & Type4Py (c) Order for three results

Figure 3.8: Naive Ranking Order: Static Analysis before ML Approach

3.4.2 Learning to Rank Approach

We are also interested in exploring a novel approach to solving this problem by utilizing a learning-to-rank model. Our idea is to abstract the problem as a query-document challenge, where we consider the code context as the query and type inferences as the document. In this subsection, we will first present the problem abstraction and then introduce our proposed solutions.

Problem Abstraction

Currently, we possess a list of predictions for a specific type slot generated by Pyre, Pyright, and Type4Py. Our aim is to transform the rank-ordering problem into a learning-to-rank problem. To achieve this, we need to determine the query for the problem. We define the query as a ***type query***, representing the expression for the type without any annotation. This could be either the variable definition or the function context. Consequently, the documents in our learning-to-rank framework will consist of type predictions from different models.

Let's consider the following function in Figure 3.9 as an example.

test_func.py

```
1  def unsigned_hex_to_signed_int(hex_string: str) -> int:
2      """
3      Converts a 64-bit hex string to a signed int value
4      """
5      v: int = struct.unpack("q", struct.pack("Q", int(hex_string, 16)))[0]
6      return v
```

Figure 3.9: Example Function for Type Query

It has a function signature that includes the return type *int* and parameter *hex_string*. To rank the predictions for the return type, which is *int*, we will remove the type annotation for the return value while keeping all other contexts intact, including annotations for parameters and variables within the function, as shown in Figure 3.10a. Similarly, when inferring the type of a parameter, we will only remove the annotation for that specific parameter while preserving all other annotations and contexts as shown in Figure 3.10b. Additionally, we have observed that this function includes a function-level variable *v* which is another target for type inference. The type query for this variable is straightforward: it will be the assignment of *v* without any type annotation in Figure 3.11

```
def unsigned_hex_to_signed_int(hex_string: str):
    """Converts a 64-bit hex string to a signed int value.
    param hex_string: the string representation of a zipkin ID
    returns: signed int representation
    """
    v: int = struct.unpack("q", struct.pack("Q", int(hex_string, 16)))[0]
    return v
```

(a) Type Query for Return Type

```
def unsigned_hex_to_signed_int(hex_string) -> int:
    """Converts a 64-bit hex string to a signed int value.
    param hex_string: the string representation of a zipkin ID
    returns: signed int representation
    """
    v: int = struct.unpack("q", struct.pack("Q", int(hex_string, 16)))[0]
    return v
```

(b) Type Query for Parameter

Figure 3.10: Type Query for Parameter and Return Type

```
1   v = struct.unpack("q", struct.pack("Q", int(hex_string, 16)))[0]
```

Figure 3.11: Type Query for Variable

Therefore, we abstract the problem to the learning-to-rank model as:

$$x = (\, q : type\ query,\ d : type\ predicts)$$

Embedding model

In order to facilitate learning in our machine learning model, it is necessary to obtain vectorized representations of the code tokens for both queries and documents. For example, when we have a code fragment, specifically a function code fragment that serves as our type query, our goal is to generate code embeddings that capture code-related understanding. To achieve this, we explore various options such as codeBERT [30], Unixcoder [36], and Starcoder [48]. After consideration, we decide to utilize the pre-trained model of Unixcoder for our task. By leveraging Unixcoder, we can encode the type queries and type predictions into a 768-dimensional vector representation. This allows us to capture the semantic information and characteristics of the code tokens effectively.

Pointwise Solution

After embedding the type queries and type predictions into vectors, as described in the previous section, we can now explore a pointwise solution to transform the ranking problem into a **classification problem**. To achieve this, we concatenate the query and document vectors together, forming a single vector representation. The label assigned to each training sample is either 0 or 1, indicating whether the document (type prediction) matches the ground truth.

By training a classification model using these training samples, we aim to learn the ability to classify whether the query and type predictions are a match. In the prediction phase, we can then rank the documents based on the scores outputted by the classification model. These scores serve as the ranking order, allowing us to determine the most relevant matches for the given query.

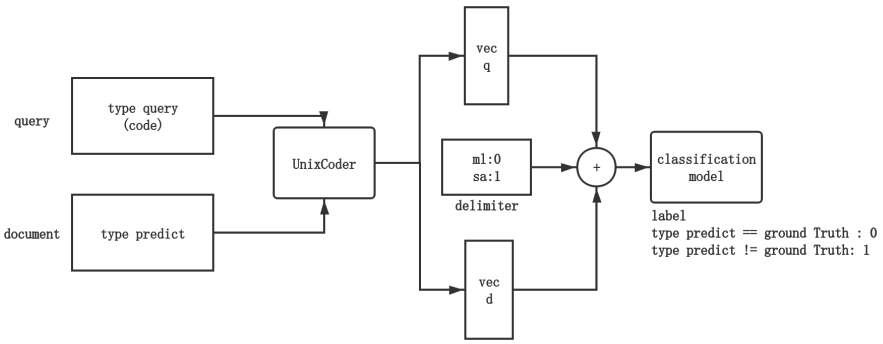


Figure 3.12: Learning to Rank Model

As illustrated in Figure 3.12, the input vector for our classification model is constructed by concatenating the query vector (vec q in the figure), the document vector (vec d in the figure), and a delimiter that serves to differentiate between different approaches. In this context, the delimiter takes the value of 1 when the documents (type predictions) originate from static analysis and 0 when they come from a machine learning approach. Consequently, the input vector becomes a 1537-dimensional vector (768 * 2 + 1). Along with

the corresponding label, which is either 1 or 0, we feed these inputs into the classification model for learning purposes.

Regarding the classification model, we conducted experiments with three different models: **XGBoost**, **Linear-layer Model**, and **Multi-layer Model**:

- XGBoost: XGBoost is a gradient boosting algorithm that utilizes decision trees [28]. It trains multiple decision trees, with each tree focusing on samples that were misclassified or had higher prediction errors in previous trees. The final prediction is formed by combining the predictions from each tree. In our experiment with XGBoost, we used the Scikit-Learn API [19] of it. Given the large input size, we adjusted the parameter settings in the API to include more gradient-boosted trees during the training process.

- Linear-layer Model: we implemented a neural network model with a single layer for this classification problem. The input layer of the model has a dimension of (1537, 1). We used the PyTorch package to build and train this model. During the training process, we employed the Binary Cross Entropy Loss [58] as the loss function and the Adam optimizer [64] for parameter optimization.

- Multi-layer Model: In addition, we conducted experiments with a multi-layer model for this classification problem. The model consists of hidden layers with dimensions (1537, 512) and (512, 256), and an output layer with dimensions (256, 1). Within the hidden layers, we utilized the ReLU activation function. Similar to the linear-layer model, we used the Binary Cross Entropy Loss as the loss function and employed the Adam optimizer for parameter optimization in the multi-layer model. This ensured consistency in the training process across both models.

Chapter 4

Evaluation

In this section, we will initially present the evaluation setup, encompassing the dataset, our baselines, and evaluation metrics. Subsequently, we answer the following research questions introduced in Chapter 1:

RQ1: What is the general performance of this hybrid approach?

RQ2: How to rank among static analysis and machine learning results in our hybrid approach?

RQ3: What is the scalability and time efficiency of this approach?

4.1 Evaluation Setup

The implementations and preparations leading up to the evaluation are described in this section. We will first display the dataset construction process and its characteristics. The introduction of the baselines and evaluation metrics follows.

4.1.1 Dataset

A dataset with enough annotated type information is needed to train and test type annotation predictions. *Typilus' Dataset*, created by Allanmanis [22], *ManyTypes4Py*, created by Mir [52], and the *BetterTypes4Py* [62], a subset of ManyTypes4Py released this year, are the most frequently used datasets for Python type inference. These datasets were produced two or three years ago, and for the advantage of training and assessing purposes, we would prefer a more recent dataset. Therefore, we created a new version of ManyTypes4Py, which we shall refer to *MT.v0.8* in the subsequent discussions. Our dataset can be downloaded on Zenodo[1].

Build Dataset

The selection of *MT.v0.8* basically based on the dependents for *mypy* [6] package in Github. Our hypothesis is that projects built based on mypy, the most well-liked static type checker

[1] https://zenodo.org/record/8255564

for Python, would contain more precise type annotations. In order to filter the 57k results returned by our search for mypy-dependent Python projects over the last four years (2018–2022), we used the selection criteria listed below in Table 4.1. Furthermore, we

Table 4.1: Dataset Selection

Selection Criteria	Number of Projects
None	57201
50+ stars & 5+ forks	2246
100+ stars & 10+ forks	1456

chose the 2k repositories among 57k Python projects with more than 50 ratings and 5 forks as our dataset. In the end, we remove none python files and exact-duplicated Python files by using the hash-deduplicate method [33].

Dataset Augmentation

We were inspired by Allamanis [22] and Mir [53] to use a static analysis tool for dataset augmentation by adding more type annotations. We chose to use the static analysis tool *Pyre* for this purpose. Unlike Mir's approach, we used both *pyre query* and *pyre infer* for augmentation. The updated version of Pyre [10] can now add type annotations for return types and parameters. We acknowledge that some projects or files couldn't be processed by either Pyre or LibSA4Py [52] due to parse issues in Table 4.2, so we ended up selecting 2220 projects as our final dataset. After filtering with useless type information, we have gained the results for 290k files with 5.1M extracted useful types, compared to type4py with 3.3M types in total for 288k files as shown in Table 4.3, there is a raise about ***54.90%*** in annotated types per file.

Table 4.2: Process of Enhancing Datasets. *The number refers number remaining.*

Process	Number of Projects	Number of Files
LibSA4Py	2230	332,284
pyre query	2228	328,716
pyre infer	2220	290,040

Table 4.3: Types & Files between ***MT V0.8*** and ManyTypes4Py.

	Files	Available Types	Types per File
ManyTypes4Py	288,760	3.3M	11.43
MT V0.8	290,040	5.1M	17.69

Dataset Properties

To look into the new dataset, ***MT V0.8***, we are two outperforming advantages:

Table 4.4 below shows the type coverage for variables, parameters, and return types. By comparing these results with ManyTypes4Py, we can observe that the new feature for pyre results in better type annotation performance for parameters and return types.

Table 4.4: Type coverage between ***MT V0.8*** and ManyType4Py.

Tasks	ManyTypes4Py	***MT V0.8***
variables	47.90%	42.01%
parameters	14.50%	45.73%
return types	10.70%	23.66%
overall	37.76%	39.50%

Figure 4.1 demonstrates that ***MT V0.8*** has a better-balanced type distribution than ManyTypes4Py, the percentage of return types and parameters have raised to 10.83% and 26.62% in the overall types.

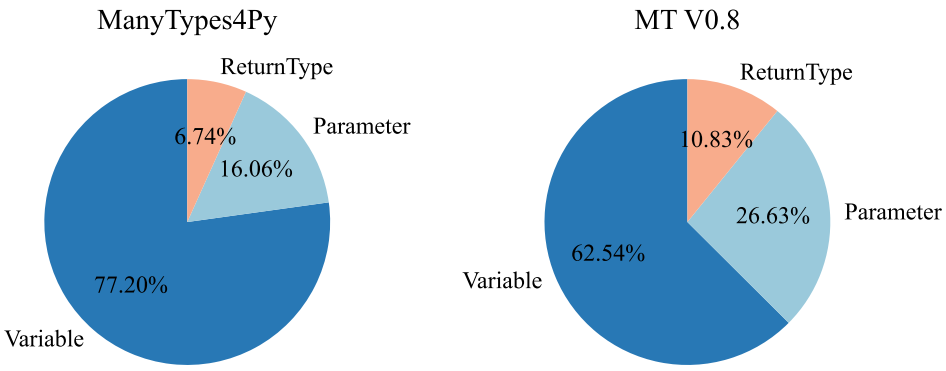


Figure 4.1: Datapoint percentage between ***MT V0.8*** and ManyType4Py.

We divided the entire dataset by projects, with a ratio of 0.7, 0.1, and 0.2 for training, validation, and testing. The model is developed and tested using training and validation data, and then it is assessed using test projects. The fundamental statics for the three sets are displayed in the following Table 4.5, and the top-10 most frequent types in the dataset is shown in the following Figure 4.2. However, during the final project-based evaluation, certain factors such as Python versions and system requirements led us to obtain a total of 80 module-based projects that could be successfully installed in the Python 3.10 virtual environment on the Ubuntu 18.04 system. We proceeded to test both the baselines and our approaches on these 80 projects.

Table 4.5: Basic Statistics in ***MT V0.8***.

	train	*valid*	*test*
Project numbers	1554	224	442
File numbers	211,414	31,912	55,102
Lines of code	32.4M	5.24M	7.48M
Types	3,734,316	554,313	842,864

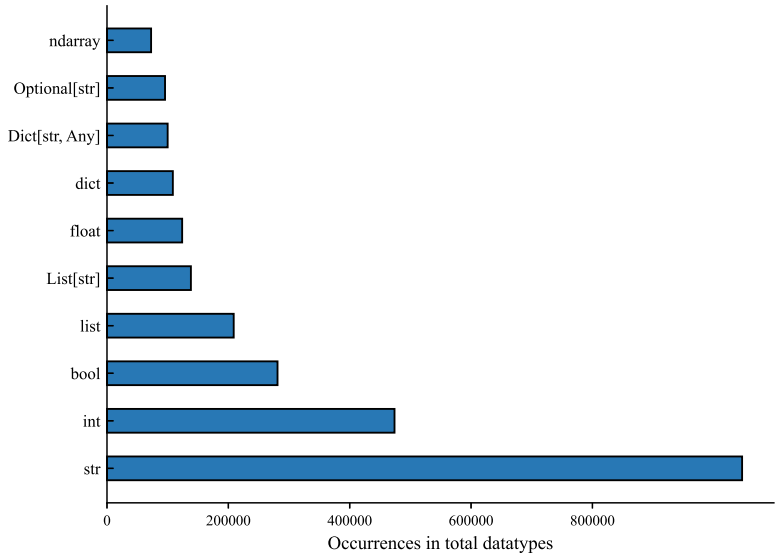


Figure 4.2: Top-10 frequent types in ***MT V0.8*** dataset.

4.1.2 Baseline Approaches

Our aim is to compare the performance of our own approach with mainstream approaches such as static analysis, machine learning, and hybrid approaches. We have selected *Pyre* and *Pyright* as the static type inference tools for static analysis. For the machine learning approach, we have opted for *Type4Py*. However, we have made some modifications to the Type4Py training process and used the reduced type cluster due to out-of-memory issues. Additionally, we have implemented project-based predictions for this approach as mentioned in the previous Chapter 3. For the hybrid approach baseline, we have chosen *TypeT5* which is a mixing approach that includes both *usage graph* and *CodeT5* [61].

For *Type4Py*, we trained the model on our training dataset and made some slight changes in the Type4Py training process. With the trained model, we proceeded to deploy it for type inference in the project base. The resulting predictions were then saved in a JSON-like file, which contains *"original_type"* and *"predictions"* information.

In order to differentiate between human-annotated type information (considered as the ground truth) and type inferences generated by static analysis, we follow a specific procedure as mentioned in the Chapter 3. Firstly, we extract all the type information from the

original test projects. Subsequently, we utilize pyre and pyright CLI commands to predict types for each type slot within the projects. We extract the pertinent type information from the stub file and combine the results into a type slot list. This allows us to calculate accuracy and evaluate the performance of Pyre and Pyright's static analysis.

Type inference for TypeT5 is performed using the pre-trained TypeT5 model, which can be accessed on Huggingface [16]. To execute the model and generate type inferences on the cleaned projects, we use the "evaluate_on_projects" API provided by TypeT5. The resulting output is converted to CSV format and evaluated using our own evaluation metrics.

4.1.3 Evaluation Metrics

To simplify the evaluation process, we first conduct a preprocessing stage for the type predictions. Following this, we define the evaluation metrics and several terms used in the evaluation phase.

Preprocess

The type indexes used among different type inferences may differ, for example Pyre used the PEP484 [8] type system which includes types such as *"typing.List"* and *"builtins.str"*. To ensure consistency between the type predictions generated by different approaches, we first remove type information from the results. We then resolve type aliases by mapping symbols such as *"{...}"* to the type *dict* and *"[...]"* to the type *list*.

Measurements

We have adopted two criteria from Typilus [22] to evaluate the performance of both the baseline methods and our own approaches: exact match and basic type match. Additionally, we modify the match rules to include the union match. For instance, in some circumstances, the ground truth (human annotation) may only contain the type for a single return expression, whereas the results of static analysis may include the ground truth in a union of results. As a result, we see a union match as an exatch match. To be specific, here are explanations and examples of their definitions:

1) **Exact match**: for an original type t_{label} and a type prediction $t_{prediction}$, we consider this is exact match only when t_{label} and $t_{prediction}$ are exactly identical:

$$t_{label} ==_e t_{prediction}$$

Also, when the original type is an union includes a *None* type, if the type prediction is same as the rest types in the original type, we consider it is an exatch match. Moreover, when the type prediction $t_{prediction}$ consists the original type t_{label}, we consider it is an exatch match, for example:

$$t_{label} : Union(str, None) ==_e t_{prediction} : str$$

$$t_{label} : str ==_e t_{prediction} : Union(str, float)$$

2) **Basic Type match** When a type contains a "[]" argument, such as *"Dict[str]"* or *"List[int]"*, it is considered to be ***parametric***. If the original type t_{label} is parametric, we adopt a broader matching criteria refered to as parametric match. This involves ignoring the type parameters outside of the "[]" argument and considering t_{label} and $t_{prediction}$ to be a parametric match only if their corresponding basic types are identical, for example:

$$Dict[str] ==_p Dict[Any]$$

$$Set[str] ==_p Set[...]$$

In the basic type match, we also provide union match rules. For instance, if the original type has a union with multiple types and the prediction matches one or more types in the union, we consider the match to be parametric, as follow:

$$t_{label} : Union(str, float) ==_p t_{prediction} : str$$

Terms

During the evaluation phase, we will look into three **subtasks** and three types of **data types** to conduct a comprehensive and detailed investigation. The subtasks in the type inference task are defined as follows:

1) ***variables***: the subtask of variable inference encompasses inferring types for variables within four domains: file-based variables, class-based variables, function-based variables within the file, and function-based variables within the class.

2) ***parameters***: The parameter subtask involves inferring types of parameters in two contexts: function parameters at the file level and function parameters at the class level.

3) ***return types***: The return type subtask is similar to the parameter subtask, which also involves two contexts: function returns at the file level and the class level.

The three kinds of data types are defined as follows:

1) ***Ubiquitous Types***: We define the following types in the set as the most common types:

$$\{str, int, list, bool, float\}$$

2) ***Common Types***: For types that appear more than 100 times in the dataset and are not part of the ubiquitous type set are categorized as common types.

3) ***Rare Types***: For types in the dataset that appear less than 100 times are classified as rare types.

4.2 RQ1: What is the general performance of this hybrid approach?

Our objective is to assess the overall performance of our approach in comparison to mainstream approaches. To achieve this, we have deployed the three baselines as well as our own approaches on the project-based test dataset. Our approaches, include the merging of Type4py and Pyre, referred to as *Type4Pyre*; the merging of Type4py and Pyright, referred to as *Type4Pyright*; and the merging of the three, referred to as *Type4SA* in the table.

4.2.1 Performance among different data types

Table 4.6 below shows the performance of these approaches among data types on the test projects. The common types are referred to as *comT*, ubiquitous types are referred as *ubT*, and rare types are referred to as *rareT* in the table.

Table 4.6: Overall Performance on Project-based Test Dataset.

Approach	Exact Match %				Parametric Match %		
	All	ComT	UbT	RareT	All	ComT	RareT
Pyre	29.80	23.82	55.52	11.36	30.12	24.01	12.05
Pyright	35.04	39.70	47.64	24.08	37.07	40.23	26.01
Type4Py	45.29	54.87	78.67	17.59	47.28	57.02	20.34
TypeT5	50.35	43.94	90.52	20.47	53.36	45.04	27.91
Type4Pyre	**52.78**	61.90	90.97	24.48	**54.01**	62.10	25.33
Type4Pyright	**56.40**	68.58	87.54	35.84	**59.87**	70.01	37.29
Type4SA	**63.30**	72.57	93.50	41.05	**67.34**	74.02	45.20

In terms of accuracy, Type4Py is outperformed by our methods, including our three approaches in the table, for both exact match and parametric match. As shown in Figure 4.3, our approach combing static analysis and type4py *Type4SA*, with a 20% boost on both match metrics on rare type datasets, our approach also significantly solves the problem of rare types. In addition, as compared to Type4Py, our technique improves on ubiquitous types by 15% and common types by 18%. Our technique also outperforms another baseline, TypeT5, which asserts to have high accuracy on its dataset, with a 20% improvement in accuracy for both match metrics. This might be because our hybrid technique uses static analysis to infer types, whereas TypeT5 relies on static analysis to acquire information needed for deep models.

4.2.2 Performance among different tasks

As previously stated, our type inference task includes three subtasks: variable subtask, parameter subtask, and return type task. We aim to evaluate the performance of type inference on each subtask and examine if there are still significant differences between them. In Table 4.7, we compare baselines and our own approaches. We observe a discrepancy among the

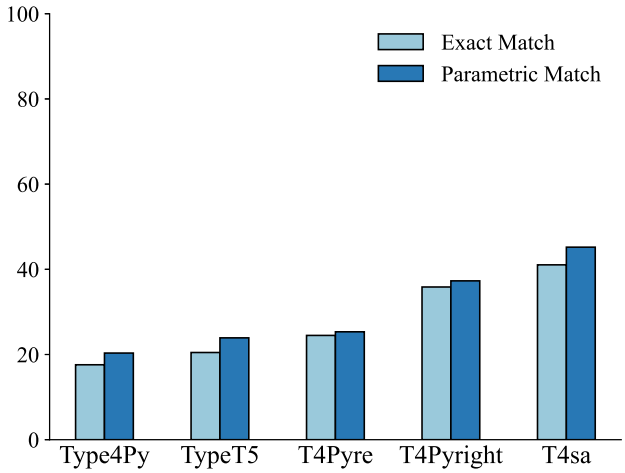


Figure 4.3: Overall Performance on **Rare Types**.

three subtasks, with accuracy on variables, including exact match and parametric match, outperforming the other two subtasks. One of our baseline approaches, TypeT5, demonstrates a more balanced performance; however, there is still a 20% difference between the variable task and the parameter task. This suggests that in type inference tasks, predicting parameters is more challenging for models to capture patterns, even with the assistance of static analysis.

We could also see that our hybrid method, which is the combination of Type4Py, Pyre and Pyright, *Type4SA*, significantly outperforms Type4Py as a baseline. In particular, we see a **20%** increase in variable accuracy, a **7%** increase in parameter accuracy, and a **20%** increase in return type accuracy. Notably, our method achieves return type accuracy for exact match and parametric match of 65.79% and 67.01%, respectively. The return type problem has, in our opinion, been somewhat addressed thanks to the use of static analysis.

Table 4.7: Performance on Three Subtasks (%)

Approach	Variables		Parameters		Return Types	
	Exact	Parametric	Exact	Parametric	Exact	Parametric
Pyre	26.94	27.02	20.13	21.76	39.33	39.45
Pyright	42.24	43.20	19.72	20.12	46.65	47.23
Type4Py	49.84	52.26	45.01	47.76	42.15	44.57
TypeT5	57.31	64.24	47.19	50.45	50.85	59.10
Type4Pyre	56.66	57.27	50.37	52.69	**60.84**	62.36
Type4Pyright	70.51	72.19	47.80	49.44	**63.92**	65.10
Type4SA	71.60	73.61	**52.24**	53.01	**65.79**	67.01

Additionally, we analyzed the accuracy of our approach on three data categories: common types, ubiquitous types, and rare types. The detailed results can be found in Table 4.8

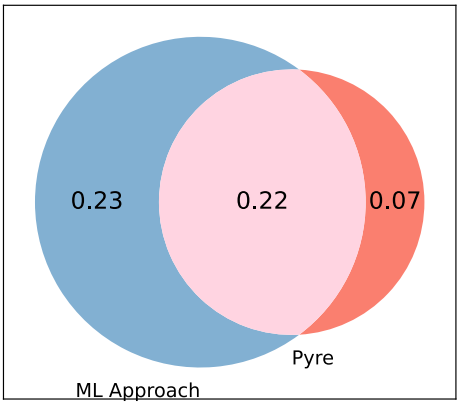
below. It was observed that ubiquitous types showed no variations across the three subtasks. However, for common types and rare types exhibited a decrease of 10 to 20 percent in the exact match between variable subtasks and the other two. The current approach still faces difficulties in accurately predicting rare types for function parameters and return values. The task of predicting rare types becomes even more challenging when it comes to function parameters.

Table 4.8: *Type4SA* Performance(exact match) on Three Subtasks (%)

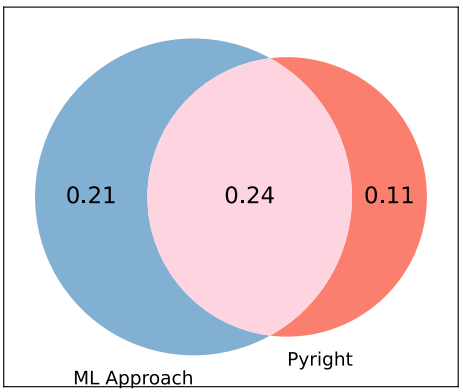
	Variables	Parameters	Return Type	All
Common Types	0.81	0.63	0.71	0.72
Ubiquitous Types	0.94	0.93	0.94	0.93
Rare Types	0.56	0.25	0.37	0.41

4.2.3 Contribution of the static analysis and ML approach to the hybrid approach

The hybrid approach we adopt consists of two components: a static analysis part and a deep model part. We aim to determine the extent to which static analysis contributes to the overall performance and whether it assists in addressing the weak points in machine learning.



(a) Contribution of Static Analysis: Pyre in Exatch Match

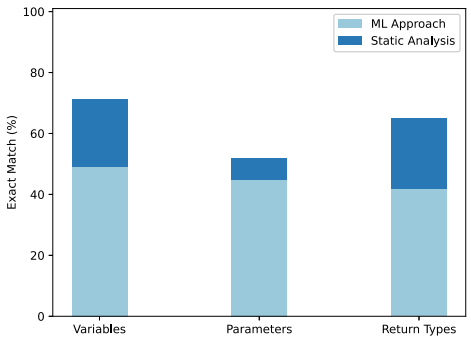


(b) Contribution of Static Analysis: Pyright in Exatch Match

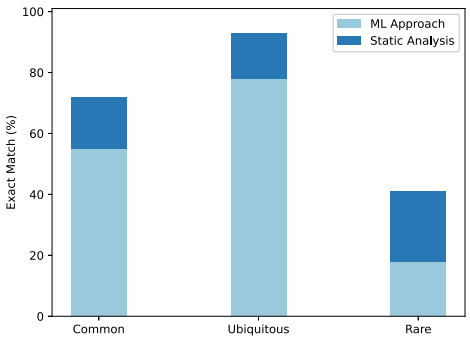
Figure 4.4: Contributaion of Static Analysis and ML Approach

Figure 4.4 illustrates the impact of ML approach and static analysis on the exact match of the test projects. The graphic specifically shows that for *Type4Pyre*: Type4Py (ML Technique) accounts for 23% of the 52% of precise matches, demonstrating the deep model's impressive pattern recognition ability without involving Pyre. The graphic, however, reveals that pyre's self-prediction represents 7%, which means that static analysis accounts for around one-seventh of the overall performance of type inference. Additionally, exam-

ining *Type4Pyright*, which exhibits a higher exact match rate of 56%, we observe that the static analysis tool Pyright plays an even more complementary role. Pyright contributes to approximately one-fifth of the overall match rate, further enhancing the performance of the type inference system.



(a) Contribution of Static Analysis and ML among Tasks



(b) Contribution of Static Analysis and ML among Data Types

Figure 4.5: Contributaion of Static Analysis and ML Approach

In the depicted Figure 4.5, we examine the impact of static analysis on the performance of the three subtasks and data types. Notably, there is a substantial enhancement for rare type prediction on variables and return types. Specifically, applying static analysis on rare types in these two tasks results in an almost twofold increase in accuracy compared to the baseline. We could conclude that static analysis can enhance the weaknesses, specifically the *rare type issue* and *return type issue*, of machine learning to a certain extent.

4.3 RQ2: How to rank among static analysis and machine learning results in hybrid approach?

As explained in the Chapter 3, a ranking system may be necessary when there are discrepancies between the results obtained from static analysis and the ML Approach for the same type slot. In the table below, we compare the Naive Approach (static analysis before ML approach) with three models using the learning-to-rank approach. Unfortunately, we are disappointed to observe that the learning-to-rank approaches did not surpass the naive approach. This suggests that when faced with a discrepancy between the ML approach and static analysis, it may be preferable to initially trust the static analysis result.

Additionally, we discovered that using the neural network model instead of XGBoost for the classification mode resulted in a better Mean Reciprocal Rank (MRR). This implies that when dealing with classification problems that involve large input dimensions (such as our case with 1537 dimensions), neural networks are more powerful than boosted trees.

Table 4.9: Performance Evaluation of Rank System (%)

Approach	Exact Match	Naive Approach	Learning to Rank		
			XGBoost	Linear-layer	Multi-layer
Type4Pyre	52.78	**49.23**	40.33	42.50	43.10
Type4Pyright	56.40	**52.01**	41.30	40.93	42.01
Type4SA	63.30	**60.34**	51.01	54.33	55.20

4.4 RQ3: What is the scalability and time efficiency of this approach?

In this research question, we would like to examine the practicality and time efficiency of our approaches. Specifically, we want to compare the inference time per type slot among our approaches and baselines. Figure 4.6 below displays the results, which include Pyre and Pyright, Type4Py, TypeT5, and our hybrid approaches.

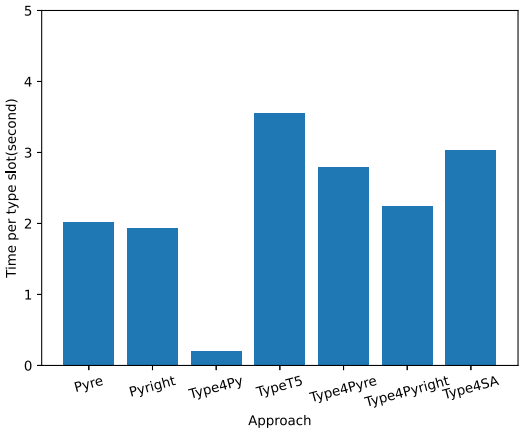


Figure 4.6: Time Efficiency among Approaches

For each type slot, Pyre and Pyright static analysis approach take about 2 seconds, although this also includes the time needed to connect to and initiate the Pyre or Pyright server and extract the type from the stub file. Another baseline, type4py, is the most time-efficient approach, taking approximately 0.20 seconds per type slot. Our hybrid approach, *Type4SA* which combines static analysis and type4py, takes less than 3 seconds per type slot, accounting for the time taken to wait for the multiprocessors of static analysis and ML approach. However, the other baseline, the hybrid approach typeT5, takes significantly longer, approximately 4 seconds per type slot, because it first extracts the callers-callees graph in the project base. It can be inferred that utilizing static analysis to supplement machine learning predictions rather than extracting information for it not only leads to improved results but also saves time in practical applications.

Chapter 5

Conclusions and Future Work

We will summarize the contributions of our work in this chapter. Following this summary, we will consider the findings and reach conclusions. A few suggestions for future development will then be explored.

5.1 Contributions

In this thesis and our research, we suggest a hybrid strategy that uses static analysis tools to augment the classic ML approach. We experimented with two static analysis tools, Pyre and Pyright, and developed algorithms to employ these tools for type inference. A more recent version of the ManyTypes4Py dataset is also gathered, and Type4Py's training procedure is modified to make it fit a larger dataset. Finally, we test if it is possible to create an order among the outcomes of static analysis and machine learning using the learning-to-rank approach. We experiment with several approaches to address the learning-to-rank problem with the goal of incorporating the rank issue into type inference. Our contributions can be summarized as follows:

- Proved complementary between static analysis and the ML approach
- Utilized static analysis tools to perform type inference
- Created an improved dataset and implemented enhancements to the Type4py model
- Evaluated the effectiveness of Learning to Rank in the type inference ordering system

5.2 Conclusions

Based on our research experiments and the evaluation, we can draw the following conclusions:

1) ***The combination of static analysis and the ML approach proved to be complementary***: By leveraging static analysis tools for type inference and incorporating

the Type4py model, the research demonstrated the effectiveness of integrating these two methodologies.

2) ***Learning to Rank is not as effective as the naive approach in the type inference rank system***: The results obtained indicate that the naive approach, which does not involve complex machine learning techniques, outperforms the learning to rank method in terms of MRR. It highlights the importance of carefully considering the applicability and suitability of machine learning methods in different problem domains.

5.3 Future work

Future work can involve exploring the potential of other static analysis tools, such as Mypy, Scapel, and others, to enhance the type inference ability. While the research experiment focused on Pyre and Pyright, there are a number of other static analysis tools that may be looked into to further increase the efficacy and accuracy of type inference.

Additionally, the current research primarily considered the pointwise approach in learning to rank. Future investigations could explore the pairwise and listwise approaches in order to assess their effectiveness in the context of type inference rank systems. Implementing and comparing these different approaches can provide valuable insights into the optimal learning-to-rank strategy for improving type inference order systems. Another area for further investigation is broadening the evaluation to incorporate various codebases, programming languages, and software domains. This broader analysis would provide a more comprehensive understanding of the strengths and weaknesses of Learning to Rank in various contexts.

Future research can deepen our understanding of the complimentary nature of static analysis and ML in type inference by taking into account more static analysis tools, investigating various learning-to-rank methodologies, and expanding the evaluation to diverse codebases and programming languages. These initiatives will help create more efficient and flexible type inference systems for a variety of software applications.

Bibliography

[1] Concrete syntax trees (cst), . URL https://libcst.readthedocs.io/en/latest/why_libcst.html#concrete-syntax-trees-cst.

[2] Parsing and visiting, . URL https://libcst.readthedocs.io/en/latest/tutorial.html#Parse-Source-Code.

[3] Learning to rank: A complete guide to ranking using machine learning. URL https://towardsdatascience.com/learning-to-rank-a-complete-guide-to-ranking-using-machine-learning-4c9688d370d4.

[4] torch.nn.lstm: Pytorch. URL https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html.

[5] A concrete syntax tree (cst)parser and serializer library for python. URL https://libcst.readthedocs.io/en/latest/#.

[6] mypy: Optional static typing for python. URL https://github.com/python/mypy.

[7] Open neural network exchange: The open standard for machine learning interoperability. URL https://onnx.ai/.

[8] Pep 484 – type hints for python. URL https://peps.python.org/pep-0484/.

[9] Pyre: A performant type-checker for python 3, . URL https://pyre-check.org/.

[10] Pysa: Coverage increasing strategies, . URL https://pyre-check.org/docs/pysa-coverage/.

[11] Pyright: Understanding type inference, . URL https://microsoft.github.io/pyright/#/type-inference?id=type-inference.

[12] Pytype: A static type analyzer for python code. URL https://github.com/google/pytype.

[13] Scalpel: The python static analysis framework. URL https://github.com/SMAT-Lab/Scalpel.

[14] torch.nn.tripletmarginloss: Pytorch. URL https://pytorch.org/docs/stable/generated/torch.nn.TripletMarginLoss.html.

[15] Typescript: Javascript with syntax for types, . URL https://www.typescriptlang.org/.

[16] Typet5: Seq2seq type inference using static analysis, . URL https://huggingface.co/MrVPlusOne/TypeT5-v7.

[17] Watchman:a file watching service. URL https://facebook.github.io/watchman/.

[18] A convenience method for performing mutation-like operations on immutable nodes. URL https://libcst.readthedocs.io/en/latest/nodes.html#libcst.CSTNode.with_changes.

[19] Scikit-learn interface: Xgboost. URL https://xgboost.readthedocs.io/en/stable/python/python_intro.html#scikit-learn-interface.

[20] Generating type stubs. URL https://github.com/microsoft/pyright/blob/main/docs/type-stubs.md.

[21] Ole Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis: First International Static Analysis Symposium, SAS'94 Namur, Belgium, September 28–30, 1994 Proceedings 1*, pages 78–100. Springer, 1994.

[22] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

[23] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to c ˆ. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24*, pages 76–100. Springer, 2010.

[24] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.

[25] Christopher Burges, Robert Ragno, and Quoc Le. Learning to rank with nonsmooth cost functions. *Advances in neural information processing systems*, 19, 2006.

[26] Christopher JC Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[27] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007.

[28] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, et al. Xgboost: extreme gradient boosting. *R package version 0.4-2*, 1(4):1–4, 2015.

[29] De Cheng, Yihong Gong, Sanping Zhou, Jinjun Wang, and Nanning 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*, pages 1335–1344, 2016.

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[31] Michael Furr, Jong-hoon An, Jeffrey S Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866, 2009.

[32] Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in javascript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 758–769. IEEE, 2017.

[33] David Geer. Reducing the storage burden via data deduplication. *Computer*, 41(12): 15–17, 2008.

[34] Alex Graves and Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.

[35] Bernd Gruner, Tim Sonnekalb, Thomas S Heinze, and Clemens-Alexander Brust. Cross-domain evaluation of a deep learning-based type inference system. *arXiv preprint arXiv:2208.09189*, 2022.

[36] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[37] John V Guttag. *Introduction to computation and programming using Python: With application to understanding data*. MIT press, 2016.

[38] Marijn Haverbeke. *Eloquent javascript: A modern introduction to programming*. No Starch Press, 2018.

[39] Kazuaki Ishizaki, Takeshi Ogasawara, Jose Castanos, Priya Nagpurkar, David Edelsohn, and Toshio Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 169–180, 2012.

[40] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[41] Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. Learning type annotation: is big data enough? In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1483–1486, 2021.

[42] Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. Learning to predict user-defined types. *IEEE Transactions on Software Engineering*, 2022.

[43] Alexandros Karatzoglou, Linas Baltrunas, and Yue Shi. Learning to rank for recommender systems. In *Proceedings of the 7th ACM Conference on Recommender Systems*, pages 493–494, 2013.

[44] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[45] Shriram Krishnamurthi. *Programming languages: Application and interpretation*. Brown University, 2012.

[46] Jukka Lehtosalo, G v Rossum, Ivan Levkivskyi, Michael J Sullivan, David Fisher, Greg Price, Michael Lee, N Seyfer, R Barton, S Ilinskiy, et al. Mypy-optional static typing for python. *URL: http://mypy-lang. org/[cited 2021-11-30]*, 2017.

[47] Jukka Lehtosalo, G v Rossum, Ivan Levkivskyi, Michael J Sullivan, D Fisher, G Price, M Lee, N Seyfer, R Barton, S Ilinskiy, et al. Mypy: Optional static typing for python. *URL: http://mypy-lang. org/[cited 2021-11-30]*, 2021.

[48] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[49] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[50] Tie-Yan Liu et al. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval*, 3(3):225–331, 2009.

[51] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[52] Amir M Mir, Evaldas Latoškinas, and Georgios Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 585–589. IEEE, 2021.

[53] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: Practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.

[54] John-Paul Ore, Carrick Detweiler, and Sebastian Elbaum. An empirical study on type annotations: Accuracy, speed, and suggestion effectiveness. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–29, 2021.

[55] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: a hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2019–2030, 2022.

[56] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[57] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages: 10th International Workshop, IFL'98 London, UK, September 9–11, 1998 Selected Papers 10*, pages 169–185. Springer, 1999.

[58] Usha Ruby and Vamsidhar Yendapalli. Binary cross entropy with deep learning technique for image classification. *Int. J. Adv. Trends Comput. Sci. Eng*, 9(10), 2020.

[59] Andreas Stuchlik and Stefan 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*, pages 97–106, 2011.

[60] David Thomas, Chad Fowler, and Andrew Hunt. *Programming ruby*. Pragmatic, 2004.

[61] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

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

[63] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. Listwise approach to learning to rank: theory and algorithm. In *Proceedings of the 25th international conference on Machine learning*, pages 1192–1199, 2008.

[64] Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*, pages 1–2. Ieee, 2018.