Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR’S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Предсказание типа переменных в коде Python /** **Variables type prediction in Python code**

Работу выполнил / Thesis is executed by	**Хузин Айдар Илдусович / Khuzin Aidar Ildusovich**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Иванов Владимир Владимирович / Ivanov Vladimir Vladimirovich**	подпись / signature

Иннополис, Innopolis, 2023

Contents

List of Tables

List of Figures

Abstract

The main advantage of dynamically typed programming languages such as Python is fast prototyping, which makes programming faster, whenever static typing makes code more understandable and maintainable, in addition to providing earlier bug detection. With PEP484 the community added the opportunity to optionally type variables and functions to complement Python with the advantages of static typing. Using type inference tools makes typing easier, that is why researchers developed various Machine Learning and Deep Learning techniques for type inference based on CRFs, PGMs, LSTMs, RNNs, and GNNs. The Current SoTA type inference model for TypeScript is TypeBert, which is based on BERT model and treats type inference as a sequence annotation task. However, no research has investigated the performance of BERT-like models on type inference task in Python code. To address this research gap, I fine-tuned CodeBERT model for the type prediction task in Python code. Additionally, the model was trained under Deep Similarity Learning (DSL) objective, to resolve the closed vocabulary issue, which is presented in TypeBert. Thus, the obtained model could learn to predict new types without retraining. In this work I observed how well DSL could be applied to CodeBERT model and how the choice of mining strategy and margin value impacts the performance. The obtained model did not outperform SoTA solutions, however it presented comparable performance with the general drop in accuracy of 5.4% compared to Type4Py.

Chapter 1

Introduction

Dynamic typing has gained popularity in recent years, and Python is identified as the most widely used programming language in 2022 according to IEEE Spectrum [1]. One of the primary advantages of dynamic languages is their ability to enable fast prototyping, facilitating faster programming [2]. This means that developers can write and test code rapidly without worrying about type annotations. The dynamic nature of these languages allows for more flexible programming since variables can change type at runtime.

However, while dynamic typing may be beneficial during the initial stages of the software lifecycle, it can lead to issues with code maintainability and bug detection. As the codebase grows and becomes more complex, it can be challenging to keep track of variable types, leading to errors that may not be detected until runtime. This is where static typing can help. By explicitly annotating the types of variables and function parameters, developers can make code more understandable and maintainable, and can also help to detect bugs earlier in the development process [3].

To complement Python with the advantages of static typing, the community added the optional typing of variables and functions with PEP484 [4]. Developers can use the features introduced with this standard to gradually type code, without having to make a full transition to a static typing. This has led to the development of several static analysis-based type inference tools for Python, such as Pytype [5], Pyre [6], Pysonar2 [7], and PyRight [8]. Static analyzers predict types based on predefined rules and constraints. This technique is already used in some statically typed languages, such as Scala and Kotlin [9], [10]. While static analysis-based type inference tools can be useful, they have limitations. In some cases, there may not be enough static constraints on a variable to predict a type deterministically. Additionally, static type checkers have an over-approximation problem [11].

To address these limitations, recent research have explored the use of machine learning and deep learning solutions based on conditional random fields, probabilistic graphical models, LSTMs, RNNs, and GNNs. These models have demonstrated superior performance over static analyzers [12].

The current SoTA model for type inference in Python is HiTyper [13]. HiTyper is a novel approach that combines static analyses with neural networks to infer types. For TypeScript, the leading SoTA model is TypeBert [14], which treats type inference as a sequence annotation task. However, despite the advancements in type inference research, there is a lack of studies that have explored the performance of BERT-like models specifically on type inference tasks in Python code.

To address this research gap, I fine-tuned CodeBERT [15] for type prediction on the ManyTypes4Py [16] dataset. Additionally, I tackled the closed vocabulary issue presented in TypeBert through the use of Deep Similarity Learning (DSL). This work explores the following research questions.

RQ1: What is the performance of CodeBERT trained under the DSL objective compared to other type inference models for Python?

RQ2: How does the choice of triplet mining strategies impact performance?

RQ3: How does the choice of margin value affect performance?

Chapter 2

Literature Review

There are a number of type checkers for python based on static analysis, such as Pytype [5], Pyre [6], Pysonar2 [7], PyRight [8]. Static approaches are inferring types based on a set of rules and constraints. However, these methods tend to be imprecise [17] because of the dynamic nature of Python. Additionally, static methods over-approximate [11] programs' behavior.

Rachev et al. [18] presented a learning based model, which can predict variable names and types in JavaScript code using conditional random fields (CRFs) called JSNice. The core idea of JSNice is to build a network to capture dependencies between the program elements and learn statistical correlations.

Xu et al. [12] used probabilistic graphical model (PGM) to infer variable types in Python. They identified four type hints: "data flow between variables, attribute accesses, variable names, and explicit type checks". By their assumption, some attributes may be uncertain and merging type hints using a probabilistic method can resolve the issue. The authors have shown the superiority of their model over PySonar2.

Hellendoorn et al. [19] treated type inference as a sequence annotation task like POS tagging or NER. But in their case, the model gets a sequence of types as a result. Presented DeepTyper model can capture a wide context. However, it treats every token occurrence as a new variable, so the same variable may have different type annotations in the same context.

Malik et al. [20] introduced the NL2Type model that works on top of an LSTM-based recurrent neural network that can predict function arguments and return types. The backbone of their approach is to extract natural language information, such as function names and comments. Authors have shown that NL2Type has superiority over both JSNice and DeepTyper.

In TypeWriter Pradel et al. [21] used both natural language and code context to infer types. Their model gathers natural language type hints "in the form of identifier names and comments", and code context type hints as usage sequences. Then the model encodes each type hint with RNNs. Additionally, TypeWriter extracts available types and represents them as a type mask. To predict types, the model concatenates all the obtained information into one vector and passes it into one fully connected layer. Additionally, TypeWriter validates produced types with an external gradual type checker to get results, acceptable to that tool. The authors have shown that their model outperforms NL2Type and DeepTyper.

Wei et al. [22] presented LambdaNet, a probabilistic type inference algorithm for TypeScript. Their approach is to obtain a type dependency graph and compute vector representation for each variable using graph neural network (GNN). To predict types, LambdaNet uses a pointer network-like architecture [23] that is capable of predicting unseen types. LambdaNet also outperforms DeepTyper, moreover it presents predictions consistency, because it makes variable-level predictions, in contrast to token-level predictions of DeepTyper.

Pandi et al. [24] introduced the idea of combining logical and natural constraints and formulated the problem as an optimization task. To test their approach, the authors have implemented OptTyper model, which predicts variable types in TypeScript. To extract logical constraints, OptTyper uses the TypeScript compiler [25]. OptTyper produces compilable types like TypeWriter. But in contrast to it, the presented model does not need to validate results after prediction, as it gathers logical constraints before inference. As for natural constraints, OptTyper only exploits identifier names and encodes them using char-level LSTM. Additionally, natural constraints can be retrieved using other learning-based tools, for example, with LambdaNet. However, Pandi et al. did not test that. The authors have shown that OptTyper has 4% higher accuracy than LambdaNet. The comparison with TypeWriter was not provided.

Allamanis et al. [26] presented the GNN based Typilus model for predicting types in Python. To construct a graph representation of a code, their model utilizes various sources such as syntactic patterns, data flow, and token names. Using such a graph, the model maps tokens to *TypeSpace*. The variable types are then obtained using the KNN algorithm. The idea of *TypeSpace* enables adding unseen types to the type vocabulary without retraining. Afterward, inferred data types are validated with type checker to remove false positives.

In 2021 Jesse et al. [14] adopted BERT to predict types for TypeScript. The authors have shown that presented TypeBert model outperforms LambdaNet and OptTyper. By the authors' assumption, their model implicitly learns the semantics of code, even without direct construction of graphs. TypeBert is SoTA model for type inference in TypeScript. However, there is a possible improvement, because the model makes predictions on fixed-size type vocabulary. Applying a pointer network-like architecture can solve this issue.

Mir et al. [27] presented a model for type inference in Python. Their Type4Py model extracts three kinds of type hints. The first is natural information. It is the names of variables and functions along with the names of the function arguments. The second type hint is code context, for this the model extracts all usages of a variable. The last kind of hints is visible type hints. Type4Py gathers all import statements to get possible data types. Then the model encodes type hints using HNN. The authors treated the type prediction as a deep similarity learning problem in order to make the prediction on unbound vocabulary. In the presented paper, the authors have shown that Type4Py outperforms both Typilus and TypeWriter. Additionally, the model can be improved by utilizing pointer-network-like architecture.

Recently, Peng et al. [13] introduced the SoTA approach to predict types in Python. Their HiTyper tool consists of three main components: "type dependency graph generation, static type inference, and neural type prediction". They developed a static type inference module that is capable of predicting 2-3 times more annotations with higher exact match score compared to Pytype and Pyre Infer. Along with it, the authors obtained type rejection rules to extend the static analyzer with types inferred by the neural type prediction module. Thus, inferred types pass type checkers. The authors did not develop their own neural model and used Typilus and Type4Py instead. In general, combining their tool with Typilus and Type4Py improved the performance of underlying models by 11-15%. Especially great improvements of 22-39% and 30% were achieved on return value type inference and rare types prediction respectively. However, inferring subtypes is still a problem for HiTyper.

In 2023 Wei et al. announced TypeT5 [28] model, that outperformed HiTyper. Additionally, OpenAI released Large Language Model called ChatGPT [29], that is capable to annotate Python code. My work was mainly done before the announcement of these models. That is why I did not compare my model with the mentioned new approaches.

Chapter 3

Methodology

I Type inference as a sequence annotation task

Despite the fact that the program code has a sophisticated structure and there are known ways to present it as graphs, graph-based models for both Python and TypeScript languages were outperformed by other models. TypeBert [14] - The SoTA type inference model for TypeScript is based on BERT [30] and treats type inference as a sequence annotation task.

To understand what performance a BERT-based model can introduce in Python specifically, CodeBERT [15] was adopted for the type inference task. Additionally, the model was fine-tuned with the Deep Similarity Learning (DSL) objective to resolve the closed vocabulary issue.

II The base model

This section describes CodeBERT. The backbone of this model is a multilayer bidirectional Transformer [31]. Specifically, the model is the same as RoBERTa [32]. CodeBERT was pretrained on large multilingual dataset of *bimodal* and *unimodal* data. *Bimodal* data is parallel sequences of natural language (*docstrings*) and code, while *unimodal* data is code snippets without paired data. The model was pre-trained with Masked Language Modeling (MLM) and Replaced Token Detection (RTD) objectives.

III Deep Similarity Learning

The output of CodeBERT is a vector representation of tokens (embedding). Embedding can store various properties of a token, which can be used further by other neural networks. A common technique for sequence annotation is to pass these vector representations to some Fully Connected Neural Network (FCNN) and obtain the class label using the softmax function. However, labeling tokens in such a way is only possible when there is a known set of possible types. To overcome this issue, I treated the annotation as a DSL problem.

Under the DSL objective, a model learns to represent an input as some point in a multidimensional real space. Thus, a model can map a token with unseen type to the same space.

IV Loss

To make the model map the inputs with the same labels close to each other and as far as possible from the other inputs, I used the *Triplet Loss* [33] function. Given the anchor a, positive example with the same label p, negative example with another label n, positive scalar margin m, and distance function d the *Triplet Loss* is calculated as follows:

$$L(a,p,n) = max(0, m + d(a,p) - d(a,n)) \tag{3.1}$$

The objective function L aims to produce embeddings in a way that the anchor embedding closer by the margin value to the positive embedding than to the negative example. The illustration of the Triplet Loss objective is provided in (Fig. 1).

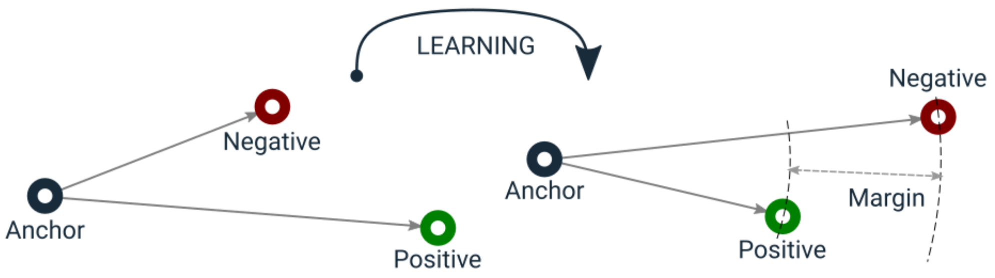

Fig. 1. Triplet Loss objective.

V Triplets mining

Given the described loss function, it is still necessary to choose triplets. There are two main ways to do it. Either mine triplets before training, the so-called offline strategy, or sample them for each mini-batch, in other words, the online strategy.

An offline strategy will produce a list of triplets *(a, p, n)*. In this case, the model needs to process the whole sequence in which each example is met to get an embedding because embeddings are contextualized and strongly depend on the context.

The offline strategy will make training inefficient, so I decided to concentrate on online strategies, which are described in the next sections.

A. Batch All strategy

An intuitive way to sample triplets is to process all possible ones. Given the embeddings *X*, the function *C* that produces the correct class of an embedding *Batch All* loss is calculated as follows:

$$L_{ba}(X) = \overbrace{\sum_{a}}^{\text{all anchors}} \; \overbrace{\sum_{p}}^{\text{all positives}} \; \overbrace{\sum_{n}}^{\text{all negatives}} L(a,p,n) \tag{3.2}$$

where

$$\begin{aligned} &a \in X \\ &p : \substack{p \in X \\ p \neq a \\ C(p)=C(a)} \\ &n : \substack{n \in X \\ C(n) \neq C(a)} \end{aligned}$$

However, it is crucial to sample triplets producing useful gradients for a model; as if a negative example is already farther from a positive example by a margin, the loss will be zero. *Batch All* strategy is likely to produce many examples with zero loss, thus vanishing losses of other triplets.

B. Batch Hard strategy

Batch Hard (BH) strategy is another way to form triplets; for each anchor, only the hardest positive and negative examples are passed to the loss function.

$$L_{bh}(X) = \overbrace{\sum_{a}}^{\text{all anchors}} L(a, \overbrace{\underset{p}{\text{argmax}}(d(a,p))}^{\text{hardest positive}}, \overbrace{\underset{n}{\text{argmin}}(d(a,n))}^{\text{hardest negative}}) \tag{3.3}$$

where

$$a \in X$$
$$p : \substack{p \in X \\ p \neq a \\ C(p)=C(a)}$$
$$n : \substack{n \in X \\ C(n) \neq C(a)}$$

It is noticeable that showing only the hardest positives and negatives to a model can make it learn only on outliers. At the same time, triplets are sampled only within mini-batches, therefore examples are selected on the small subset of overall data, making such triplets *moderate*.

C. Batch Semi-Hard strategy

Batch Semi-Hard strategy (BSH) has shown the best performance, according to [33], [34].

$$L_{bsh}(X) = \overbrace{\sum_{a}}^{\text{all anchors}} L(a, \overbrace{\underset{p}{\operatorname{argmin}}(d(a,p))}^{\text{easiest positive}}, \overbrace{\underset{n}{\operatorname{argmin}}(d(a,n))}^{\text{semihard negative}}) \tag{3.4}$$

where

$$a \in X$$
$$p : \substack{p \in X \\ p \neq a \\ C(p)=C(a)}$$
$$n : \substack{n \in X \\ C(n) \neq C(a) \\ d(a,p) < d(a,n)}$$

The key idea behind this approach is that it is not necessary for all examples of some class to converge to the same point in space, because the inference is commonly implemented using KNN lookups. Therefore, only the nearest neighbors matter; (Fig. 2) illustrates the objective.

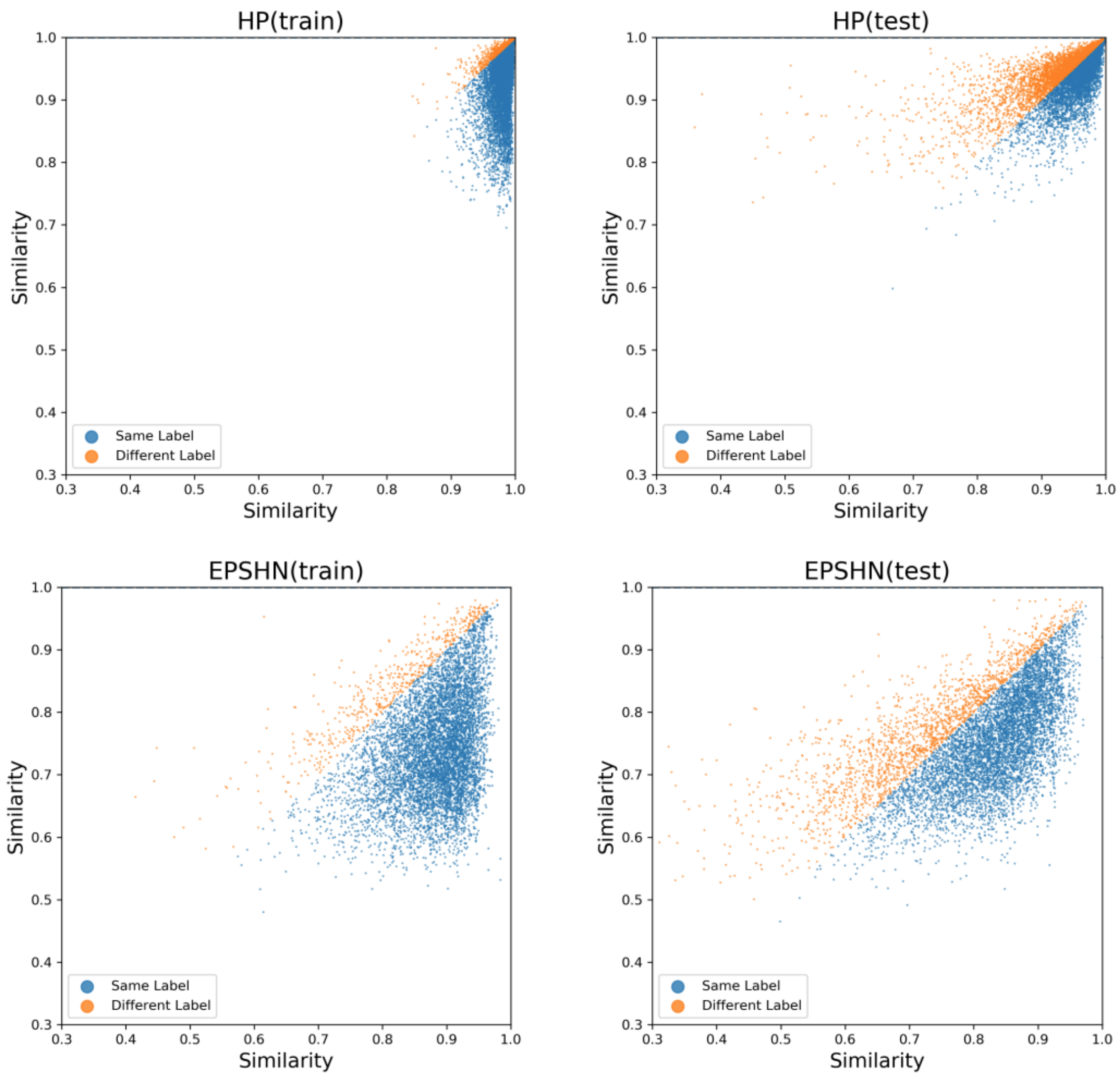

Fig. 2. Comparison of the *Batch Hard* strategy with the *Batch Semi-Hard* as illustrated in [34, Figure 5]. *HP* is the same as *Batch Hard* and *ESPHN* is the same as *Batch Semi-Hard*.

VI Training

I fine-tuned the model using *Batch Hard* and *Batch Semi-Hard* mining strategies, combining them or using each individually. The results will be reviewed in Chapter 5.

After the training model is capable to map tokens to multidimensional real space. However, it cannot infer token types. To address this issue, the Type Space is built.

A. Learning Type Space

Training data have tokens with annotated types. For each of these tokens, the model computes the projection to *Type Space*. Projections are marked with the corresponding type labels. At this point, the embeddings for all types met in the training data are stored.

B. Extending the vocabulary

The vocabulary of a model could be extended incrementally during usage. If the model meets a variable with a new type, it can predict the embedding for it and put the projection to the Type Space with the corresponding type label. There is no need to retrain or fine-tune the model itself; only *Type Space* is extended.

C. Inference

The model calculates an embedding for a variable and finds the nearest neighbor of this embedding in the *Type Space*. The type of this neighbor is the inferred type of the variable.

As the dataset has about 1 million annotated types, it is required to use some efficient nearest-neighbor search algorithm.

VII Dataset

In this work, I use the ManyTypes4Py dataset [16]. Mir et al. collected Python projects with *mypy* dependency from GitHub. They downloaded 5,382 Python projects, deduplicated them, generated *normalized seq2seq representations* of source files using the [35] library. The characteristics of the dataset are presented in (Tab. I).

TABLE I
Characteristics of the ManyTypes4Py dataset as presented in [16, Table II]

Metrics	Dataset			
	All	Training	Validation	Test
Repositories[a]	5,382	4,913	2,789	3,796
Lines of code[b]	22M	-	-	-
Files	183,916	132,409	14,675	36,832
...with type annotations	50,838 (27.6%)	36,542	4,105	10,191
Functions	2,096,797	1,509,048	169,519	418,230
...with comment	1,129,573 (53.8%)	812,632	91,325	225,616
...with return type annotations	325,532 (15.5%)	234,319	26,104	65,109
Arguments	3,923,667	2,822,699	310,685	790,283
...with comment	220,976 (5.6%)	159,453	16,924	44,599
...with type annotations	480,793 (12.2%)	347,898	37,148	95,747
Types	869,825	347,898	89,334	192,102
...unique	67,060	53,614	13,995	23,572

[a] Note that there is an intersection among repositories in the three sets as the dataset is split by files.

[b] Comments and blank lines are ignored when counting lines of code.

The characteristics presented by Mir et al. [16] seems to be inconsistent.

Authors state that there are 243,319 functions and 347,898 arguments with return type annotation in the training dataset. The sum of these two values is greater than the reported value of 347,898 annotated types in total in the training dataset. My measurements regarding the total number of annotated types are shown in the (Tab. II).

TABLE II
Number of annotated types in ManyTypes4Py

Metrics	Dataset			
	All	Training	Validation	Test
Types	2,121,424	1,540,975	162,713	417,736
...unique	45,691	35,131	5,377	11,854

The dataset has some limitations due to the *normalized seq2seq representation*, that introduces information loss. Firstly, tabulations are missing. Tabulations are an important part of Python code, and it is impossible to fully restore the semantic patterns without them. Secondly, documentations and comments are replaced with "[comment]" token. In many cases, comments can be very informative about the type and meaning of variables.

VIII Data preprocessing

Originally, the dataset consists of LibSA4Py json outputs, one file for each project. The structure of json files is presented in (Listing 3.1). Accessing files one by one during the training is inefficient, so I merged all files into training, validation, and test csv files and excluded source files that does not have type annotations. Each line of these files represents one source file, it stores *file_path*, *untyped_seq*, and *typed_seq* fields. *untyped_seq* and *typed_seq* represent *normal-*

ized seq2seq representation of code and type annotations, respectively. *file_path* field allows for the original source code to be obtained in case it would be needed. Training itself requires only *typed_seq* and *normalized seq2seq representation*. The *untyped_seq* could be referred as input sequence and *typed_seq* as labels. An example of these sequences is shown in (Listing 3.2).

Listing 3.1: The structure of LibSA4Py output file

```
{"author/repo":
  {"src_files":
    {
      "file_path":
        {
          "untyped_seq": "",
          "typed_seq": "",
          "imports": [],
          "variables": {"var_name": "type"},
          "mod_var_occur": {"var_name": []},
          "mod_var_ln": {"var_name": [[1,0], [2, 2]]},
          "classes": [],
          "funcs": [],
          "set": null ,
          "tc": [false , 5],
          "no_types_annot": {"U":  0, "D": 0, "I": 0},
          "type_annot_cove": 0.0
        }
    },
    "type_annot_cove": 0.0
  }
}
```

Listing 3.2: An example of untyped and typed sequences

```
untyped_seq:
def __init__ ( self , bucket_type , timestamp , event ,
   ↪ is_end ) : [EOL] self . _bucket_type = bucket_type [EOL
   ↪ ] self . _is_end = is_end

typed_seq:
0 $None$ 0 0 0 $BucketType$ 0 0 0 0 0 $bool$ 0 0 0 0 0 0 0
   ↪ $BucketType$ 0 0 0 0 0 $bool$

correspondance illustration:
def-0 __init__-$None$ (-0 self-0 ,-0 bucket_type-$BucketType$
   ↪  ,-0 timestamp-0 ,-0 event-0 ,-0 is_end-$bool$ )-0 :-0
   ↪ [EOL]-0 self-0 .-0 _bucket_type-0 =-0 bucket_type-
   ↪ $BucketType$ [EOL]-0 self-0 .-0 _is_end-0 =-0 is_end-
   ↪ $bool$
```

Similarly to previous works [21], [26], [27], I performed several types preprocessing steps:

1. Functions such as __str__ and __len__ are trivial to annotate. __str__ always returns string , and __len__ always returns int . The annotations for these functions do not show the ability of a model to predict types and blur the results, and thus were excluded from the data.

2. Any and None types were filtered out because it is not valuable to predict them.

3. The parametric types with the nested level greater than two were replaced by Any. For example, Dict[str , List [Union[str , int]]] is rewritten as Dict[str , List [Any]]. Mir et al. state that "This removes very rare types or

outliers."

The characteristics of the dataset after applying types preprocessing steps are presented in (Tab. III). The number of type annotations and the number of unique types decreased by $\sim$42% and $\sim$11%, respectively.

TABLE III
Characteristics of the ManyTypes4Py dataset after applying types preprocessing steps

Metrics	Dataset			
	All	Training	Validation	Test
Types	1,221,711	886,306	94,545	240,860
...unique	40,074	30,803	4,736	10,372

After the previous steps, a model-specific preprocessing is performed. RoBERTa model accepts input tokenized with Byte-Pair Encoding (BPE) [36]. The major limitation of RoBERTa is that it can accept up to 512 BPE generated tokens. Therefore, each *untyped_sequence* is first tokenized and then divided into 512 token chunks.

Labels are aligned respectfully. A common approach in sequence annotation task using BERT-like models is to label only the first subtoken of each word. This step is done before *untyped_seq* chunking. After that, *typed_seq* is broken up in the same way as *untyped_seq*. As an optimization, fragments without annotations were excluded.

IX Evaluation

I followed the evaluation setup of Mir et al. Thus, the model is evaluated on *all*, *ubiquitous*, *common*, and *rare* types. *Ubiquitous* types are defined as

{ str , int , list , bool, float }. If the frequency of a type in the train set is more than 100, it is considered as *common*, and as *rare* otherwise. *Ubiquitous* types were excluded from the *common* types set. The distribution of types is shown in (Tab. IV).

TABLE IV
Distribution of the types in the ManyTypes4Py dataset

Types number	Dataset		
	Training	Validation	Test
Ubiquitous	446,872 (50.4%)	49,143 (52%)	49,143 (50.3%)
Common	276,017 (31.1%)	26,735 (28.3%)	71,650 (29.7%)
Rare	163,417 (18.4%)	18,667 (19.7%)	47,937 (19.9%)
Common unique	318	244	270
Rare unique	30,480	4,487	10,097

A. Criteria

To measure the performance of type prediction, I used two criteria proposed by [26].

1. *Exact match*: model prediction t_p and ground truth target t_t are the same.
2. *Match up to Parametric Type*[1]: type parameters are ignored, and only the base type is considered. For example, if t_p is List [int] and t_t is List [str], the criterion is satisfied.

[1] *Ubiquitous* types do not have type parameters, therefore metrics are not measured separately for this types set under *match up to Parametric Type* criterion. However, *ubiquitous* types are taken into account when measuring metrics for *all* types.

B. Metrics

Performance was evaluated with the following metrics.

1. *Hit rate top-k*: represents if there is a correct prediction among the top *k* predictions of a model.

2. *Mean reciprocal rank(MRR)*: this metric examines the position of a correct prediction. Specifically, I will use the *MRR@10* metric. Unlike the usual *MRR*, it takes into account only the first 10 predictions of a model. And if there is no relevant retrieval among the top 10 predictions, the reciprocal rank is considered zero. The motivation behind this is that the computation becomes much faster, whereas it is very unlikely that a user will consider predictions beyond the top 10.

$$MRR = \overbrace{\frac{1}{\mid Q \mid}\sum_{q}}^{\text{mean}} \overbrace{\frac{1}{rank_q}}^{\text{reciprocal rank}} \tag{3.5}$$

where

Q is the queries set

$q \in Q$

$rank_q$ represents the position of the first correct retreival for query q

Chapter 4

Implementation and Results

I Implementation details and environment setup

I fine-tuned *codebert-base* model from the Hugging Face *transformers* library using losses and triplets miners from PyTorch Metric Learning library. Training is implemented using PyTorch Lightning. KNN lookups are performed with *Faiss* [37] using the *IndexFlatL2* index placed in GPU memory.

Training and experiments were performed in the Kaggle Notebooks environment with one Nvidia Tesla P100 accelerator. The implementation is available on GitHub.[1]

II Training details

The Adam optimizer with 1e-5 learning rate was used to minimize loss. Training was performed with the batch size of 12, and it takes 1.5 hour to train one epoch.

[1] `https://github.com/gfx73/CodeBERT-DSL`

III Experiments and Results

This section describes how the choice of triplet miners and margin values affected the performance. The tables provide metrics evaluated on the validation dataset.

In all experiments, the margin value is set to one, unless otherwise stated.

A. *Training with Batch Semi-Hard strategy*

At first, I have tested the performance of the model trained using Batch Semi-Hard and Batch Hard strategies standalone. The results obtained during training with Batch Semi-Hard strategy are reported in (Tab. V).

TABLE V
Performance evaluation of CodeBERT trained with Batch Semi-Hard strategy

Hit rate Top-k	Epoch	% Exact Match				% Match up to Parametric Type		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1	1	67.3	89.4	63.8	14.3	73.6	73.8	31.4
	2	68.2	90.1	66.2	13.5	74.3	75.6	30.6
	3	69.0	91.0	67.1	14.1	75.2	76.8	31.6
	4	68.2	89.1	67.7	13.6	74.5	77.7	31.3
Top-3	1	71.3	93.1	69.9	15.8	77.8	79.6	34.8
	2	72.2	93.9	72.1	15.2	78.4	80.8	34.0
	3	72.8	94.5	73.0	15.6	79.2	82.1	34.9
	4	72.0	92.9	73.4	15.0	78.5	82.6	34.6

continued

TABLE V
Performance evaluation of CodeBERT trained with Batch Semi-Hard strategy

Hit rate Top-k	Epoch	% Exact Match				% Match up to Parametric Type		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-5	1	73.4	94.7	73.8	17	80	83	37.1
	2	74.1	95.3	75.6	16.3	80.3	83.7	36.0
	3	74.6	95.6	76.3	16.5	80.9	84.8	36.6
	4	73.9	94.3	77.0	15.8	80.4	85.5	36.3
Top-10	1	76.1	96.3	79.3	18.3	82.6	87.3	39.9
	2	76.6	96.7	80.5	18.0	82.8	87.8	39.3
	3	76.9	96.9	81.2	18.1	83.3	88.7	39.5
	4	76.2	95.8	81.3	17.1	82.5	88.8	38.5
MRR@10	1	69.9	91.6	68	15.4	76.3	77.6	33.8
	2	70.7	92.3	70.2	14.8	76.8	79.1	33.0
	3	71.4	93.0	71.1	15.2	77.7	80.2	33.8
	4	70.6	91.3	71.5	14.5	77.0	80.9	33.4

B. Training with Batch Hard strategy

During this experiment, I trained CodeBERT employing Batch Hard strategy. The performance of the model trained this way is shown in (Tab. VI).

TABLE VI
Performance evaluation of CodeBERT trained with Batch Hard strategy

Hit rate Top-k	Epoch	% Exact Match				% Match up to Parametric Type		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1	1	68.5	88.7	66.3	18.2	74.5	75.6	35.7
	2	69.7	89.9	68.2	18.6	75.6	77.1	35.9
	3	71.7	92.6	70.3	18.6	77.6	78.9	36.1
	4	70.8	90.5	71.0	18.8	76.7	79.5	36.1
Top-3	1	74.2	94.2	74.7	20.8	80.5	83.3	40.7
	2	74.6	94.3	75.9	20.8	80.8	84.1	40.3
	3	75.8	95.6	77.8	20.8	81.8	85.3	40.5
	4	75.4	94.4	78.1	21.3	81.2	85.3	40.6
Top-5	1	76.3	95.8	78.2	22.1	82.7	86.5	42.8
	2	76.6	95.7	79.4	22.1	82.7	86.9	42.6
	3	77.5	96.5	81.2	22.0	83.5	88.0	42.5
	4	77.2	95.8	81.3	22.5	83.0	87.8	42.7
Top-10	1	78.7	97.3	82.9	23.5	85.1	90.2	45.3
	2	78.8	97.1	83.8	23.6	84.9	90.3	45.1
	3	79.5	97.5	85.4	23.6	85.4	91.4	45.0
	4	79.3	97.0	85.1	24.2	85.1	90.8	45.6
MRR@10	1	71.8	91.7	71.4	19.8	78.0	80.2	38.7
	2	72.6	92.4	72.9	20.1	78.7	81.3	38.7
	3	74.1	94.3	74.9	20.0	80.1	82.8	38.9
	4	73.5	92.7	75.3	20.4	79.4	83.0	39.0

C. Batch Semi-Hard strategy followed by Batch Hard

After previous experiments, I trained CodeBERT with the Batch Semi-Hard strategy for three epochs and then continued training with the Batch Hard strategy also for three epochs. Results obtained during this experiment are reported in (Tab. VII).

TABLE VII
Performance evaluation of CodeBERT trained with BSH strategy followed by BH

Hit rate Top-k	% Exact Match				% Match up to Parametric Type		
	All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1	72.9	92.9	73.7	19.3	78.6	81.3	37.1
Top-3	76.6	95.5	80.0	22.1	82.3	86.6	41.6
Top-5	78.2	96.3	83.3	23.4	83.9	89.3	43.7
Top-10	80.2	97.3	87.1	25.2	85.8	92.2	46.5
MRR@10	75.2	94.4	77.7	21.0	80.9	84.6	39.9

D. Adjusting margin value

Lastly, I tested the performance of the model, trained with different margin values. The results of this experiment are shown in (Tab. VIII).

TABLE VIII
Performance evaluation under different margin values and BH strategy

Hit rate Top-k	Margin	% Exact Match				% Match up to Parametric Type		
		All	Ubiquitous	Common	Rare	All	Common	Rare
	0.5	**71.8**	**92.7**	**70.9**	18.3	**77.8**	**79.5**	36.0
Top-1	1	71.7	**92.6**	70.3	**18.6**	77.6	78.9	**36.1**
	2	**71.8**	92.2	**71.1**	**19.1**	**77.7**	**79.6**	**37.0**
	0.5	**76.0**	**95.8**	**78.1**	**20.9**	**82.0**	**85.8**	40.3
Top-3	1	75.8	**95.6**	77.8	20.8	81.8	85.3	**40.5**
	2	**75.9**	95.2	**78.4**	**21.5**	**81.9**	**85.7**	**41.5**

continued

TABLE VIII
Performance evaluation under different margin values and BH strategy

Hit rate Top-k	Margin	% Exact Match				% Match up to Parametric Type		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-5	0.5	**77.7**	**96.8**	**81.4**	**22.0**	**83.7**	**88.5**	42.4
	1	**77.5**	**96.5**	81.2	**22.0**	**83.5**	88.0	**42.5**
	2	**77.5**	96.1	**81.5**	**22.6**	**83.5**	**88.3**	**43.2**
Top-10	0.5	**79.6**	**97.7**	**85.4**	23.4	**85.6**	**91.5**	**45.1**
	1	**79.5**	**97.5**	**85.4**	**23.6**	**85.4**	**91.4**	45.0
	2	79.4	97.2	**85.5**	**24.0**	85.3	91.3	**45.5**
MRR@10	0.5	**74.3**	**94.4**	**75.4**	19.9	**80.3**	**83.3**	38.7
	1	74.1	**94.3**	74.9	**20.0**	80.1	82.8	**38.9**
	2	**74.2**	93.9	**75.5**	**20.6**	**80.2**	**83.3**	**39.7**

Chapter 5

Analysis

I Analysis of training progress

According to (Tab. V) and (Tab. VI) the model achieved the best metric scores in the third epoch for both Batch Semi-Hard and Batch Hard strategies. Fourth epoch did not clearly enhance the performance; hit rate top-1 increased for *common* types set, however it decreased for *all* types. The same holds for MRR@10 metrics.

As a result of these observations, I decided not to train the model further.

II Analysis of triplet mining strategies effect on performance

Tab. IX provides the performance evaluation of CodeBERT trained with different mining strategies. After analysis of this table, it can be said that CodeBERT trained with Batch Hard strategy has better performance in comparison to training with Batch Semi-Hard strategy considering all evaluation objectives. For example, hit rate Top-1 is 3.7% and 3.1% better under exact match and match up to Parametric Type criteria, respectively.

TABLE IX
Performance evaluation under different mining strategies

Hit rate Top-k	Strategy	% Exact Match				% Match up to Parametric Type		
		All	Ubiquitous	Common	Rare	All	Common	Rare
Top-1	BSH	69.0	91.0	67.1	14.1	75.2	76.8	31.6
	BH	**71.7**	**92.6**	**70.3**	**18.6**	**77.6**	**78.9**	**36.1**
	BSH+BH	**72.9**	**92.9**	**73.7**	**19.3**	**78.6**	**81.3**	**37.1**
Top-3	BSH	72.8	94.5	73.0	15.6	79.2	82.1	34.9
	BH	**75.8**	**95.6**	**77.8**	**20.8**	**81.8**	**85.3**	**40.5**
	BSH+BH	**76.6**	**95.5**	**80.0**	**22.1**	**82.3**	**86.6**	**41.6**
Top-5	BSH	74.6	95.6	76.3	16.5	80.9	84.8	36.6
	BH	**77.5**	**96.5**	**81.2**	**22.0**	**83.5**	**88.0**	**42.5**
	BSH+BH	**78.2**	**96.3**	**83.3**	**23.4**	**83.9**	**89.3**	**43.7**
Top-10	BSH	76.9	96.9	81.2	18.1	83.3	88.7	39.5
	BH	**79.5**	**97.5**	**85.4**	**23.6**	**85.4**	**91.4**	**45.0**
	BSH+BH	**80.2**	**97.3**	**87.1**	**25.2**	**85.8**	**92.2**	**46.5**
MRR@10	BSH	71.4	93.0	71.1	15.2	77.7	80.2	33.8
	BH	**74.1**	**94.3**	**74.9**	**20.0**	**80.1**	**82.8**	**38.9**
	BSH+BH	**75.2**	**94.4**	**77.7**	**21.0**	**80.9**	**84.6**	**39.9**

These results contradict the observations of Schroff et al.[33]. The authors did not provide a side-by-side comparison of these strategies, but concluded that choosing the hardest examples may lead to early convergence to bad local minima. A possible reason why this was not the case in my experiments may be the number of examples per batch. Schroff et al. performed experiments with batch size around 1,800, where each example in a batch corresponds to one class. In my case, each batch has about 108 type annotations. Additionally, this observation is supported by research by Xuan et al. [34]. They also noticed that the high number of examples per class in a batch increases the likelihood of anchor and the hardest positive example will be very dissimilar. Consequently, in the case of a lower number of examples per batch, this issue tends not to arise. Furthermore, [38] also achieved promising results by applying the Batch Hard strategy.

Another experiment carried out by me is applying Batch Hard strategy after Batch Semi-Hard strategy. This approach leads to better performance compared to using Batch Hard strategy standalone.

III Analysis of margin value effect on performance

Intuitively, making the model map dissimilar classes further in the *Type Space* may lead to better performance. To test this hypothesis, I retrained the model with the margin value of two and Batch Hard strategy. Tab. VIII shows the performance evaluation under different margin values and Batch Hard strategy.

Based on the table provided, the impact of training with different margin values on overall performance is minimal. Specifically, both the hit rate top-1 and MRR@10 exhibit changes of 0.2% across *all* types. Nevertheless, it can be observed that as the margin value increases, the performance on *ubiquitous* types tends to decrease, while the performance on *rare* types tends to improve.

IV Evaluation of the final model performance

During previous experiments, I found that the model trained with the Batch Hard strategy after the Batch Semi-Hard strategy has the best performance on the validation dataset. After that, I evaluated the performance of this model on the test dataset. I followed the experimental setup of Mir et al. Thus I compared the performance of CodeBERT measured by me with the metrics of other models reported by Mir et al. [27, Table 5]. The results of the comparison are presented in (Tab. X).

TABLE X
Performance evaluation of different approaches

Hit rate Top-k	Approach	% Exact Match				% Match up to Parametric		
		All	Ubiquitous	Common	Rare	All	Common	Rare
	CodeBERT+DSL	**71.7**	91.4	73.1	**19.5**	**78.0**	**82.1**	**37.8**
Top-1	Type4Py	**75.8**	**100.0**	**82.3**	19.2	**80.6**	**85.2**	36.0
	Typilus	66.1	**92.5**	**73.4**	**21.6**	74.2	81.6	**41.7**
	CodeBERT+DSL	**75.7**	94.8	79.4	21.8	**82.0**	87.1	41.8
Top-3	Type4Py	**78.1**	**100.0**	**87.3**	**23.4**	**83.8**	**90.6**	**43.2**
	Typilus	71.6	**96.2**	**83.0**	**26.8**	79.8	**88.7**	**49.2**
	CodeBERT+DSL	**77.4**	96.0	82.3	22.9	**83.6**	89.4	43.7
Top-5	Type4Py	**78.7**	**100.0**	**88.6**	**24.5**	**84.7**	**92.1**	**45.5**
	Typilus	72.7	**96.7**	**85.1**	**28.2**	80.9	**90.1**	**51.0**
	CodeBERT+DSL	**79.4**	**97.2**	85.9	24.6	**85.5**	**91.9**	46.2
Top-10	Type4Py	**79.2**	**100.0**	**89.7**	**25.2**	**85.4**	**93.3**	**46.9**
	Typilus	73.3	97.04	**86.4**	**28.9**	81.5	90.9	**51.9**
MRR@10	CodeBERT+DSL	**74.1**	93.4	77.0	21.0	**80.4**	**85.2**	**40.3**
	Type4Py	**77.1**	**100.0**	**85.1**	**21.4**	**74.1**	**79.9**	29.4
	Typilus	69.0	**94.4**	**78.5**	**24.4**	67.4	75.8	**32.8**

It could be said that Type4Py has the highest overall performance. CodeBERT trained with DSL objective did not achieve SoTA performance in type inference task for Python, unlike TypeBert in the context of TypeScript. However, it presented a comparable performance with with the general drop in hit rate top-1 of 5.4% compared to Type4Py. A possible reason for that might be that existing models for Python are more sophisticated in comparison to those created for TypeScript.

Additionally, the performance of CodeBERT could be improved. In ManyTypes4Py dataset comments and tabulations are removed from code. Comments could be very informative about the meaning and type of variables. Furthermore, it is not possible to fully restore all the semantic patters presented in the Python code without tabulations. Thus, using a dataset that does not have these limitations is likely to further enhance the model performance.

V Impact of token position on performance

The position of a variable in an input sequence may affect the performance of the model. For example, if the variable is at the beginning of the sequence, then the model is unable to see what was in the code before this variable. To observe the dependence of inference performance on the variable position, I measured the MRR@10 metric across all types for every token positions from two to 510 inclusively (other positions are always reserved by special tokens). The results are illustrated in (Fig. 3).

It can be said that performance drops at positions very close to the end of a sequence. Additionally, MRR@10 score is higher when a variable is in the beginning of a sequence.

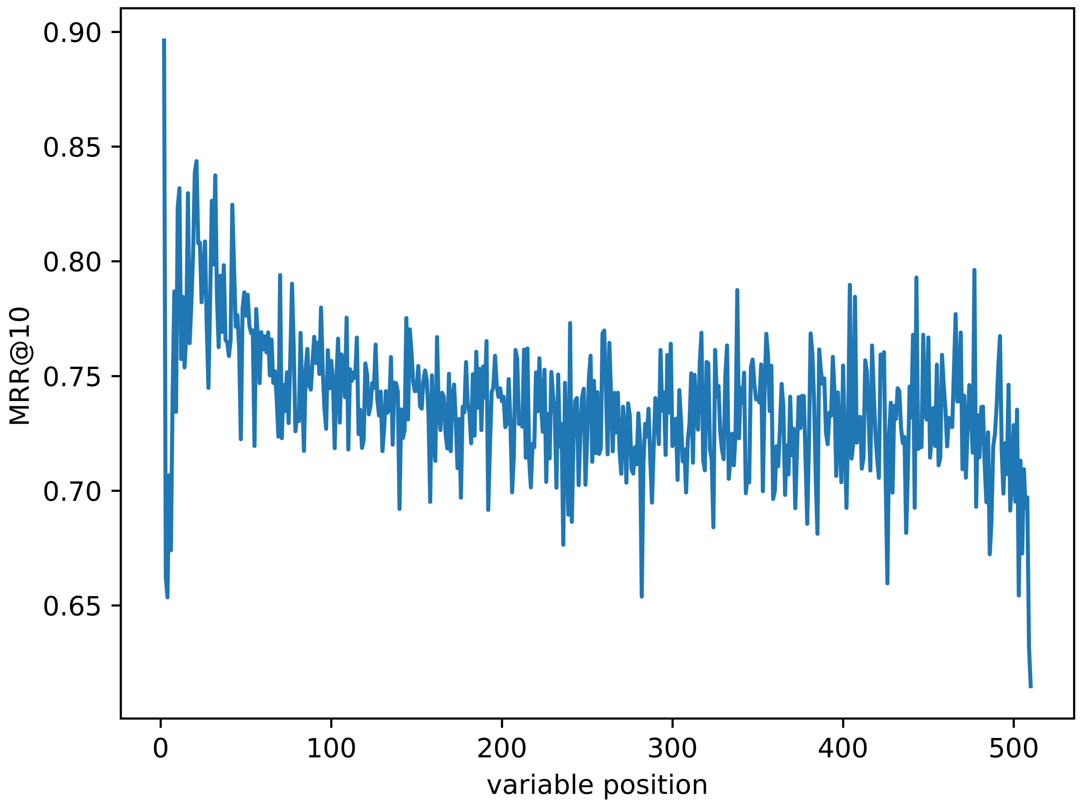

Fig. 3. MRR@10 on variable position.

Chapter 6

Conclusion

In conclusion, this study has explored the performance of the CodeBERT model for type inference in Python programming language. Fine-tuning of the CodeBERT model on the ManyTypes4Py dataset, utilizing the DSL objective, has addressed the challenge of a closed vocabulary. However, the results show that the BERT-like model applied to Python did not achieve the SoTA performance in the field of type inference, unlike TypeBert in the context of TypeScript.

I discussed a possible reason for this difference in performance, which is the more sophisticated existing models for Python compared to those created for TypeScript. Additionally, I have suggested that CodeBERT performance could be further improved by incorporating the dataset that contains comments and tabulations. As tabulations are an essential part of Python code, and comments may provide valuable information about the type of variables.

Additionally, I observed how well DSL could be applied to CodeBERT model and how the choice of mining strategy and margin affects the performance.

In summary, this study highlights the potential of BERT-based models for type inference in Python programming language, but also emphasizes the need for further research in this field to achieve better performance.

Bibliography cited

[1] S. Cass, *Top programming languages 2022*, Nov. 2022. [Online]. Available: `https://spectrum.ieee.org/top-programming-languages-2022`.

[2] A. Stuchlik and S. Hanenberg, "Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time," in *Proceedings of the 7th symposium on Dynamic languages*, 2011, pp. 97–106.

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

[4] G. V. Rossum, J. Lehtosalo, and L. Langa, "Pep 484 – type hints," PEP 484, 2014. [Online]. Available: `https://peps.python.org/pep-0484/`.

[5] Google, *Pytype*, `https://github.com/google/pytype`, 2022.

[6] Facebook, *Pyre*, `https://github.com/facebook/pyre-check`, 2022.

[7] yinwang0, *Pysonar2*, `https://github.com/yinwang0/pysonar2`, 2022.

[8] Microsoft, *Pyright*, `https://github.com/microsoft/pyright`, 2022.

[9] Scala, *Scala*, version 2.13.10, Oct. 13, 2022. [Online]. Available: `https://www.scala.com/`.

[10] JetBrains, *Kotlin programming language*, version 1.7.21, Nov. 8, 2022. [Online]. Available: `https://kotlinlang.org/`.

[11] M. Madsen, "Static analysis of dynamic languages," *Available: http://pure.au.dk/ws/files/85299449/Thesis.pdf*, 2015.

[12] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 607–618.

[13] Y. Peng, C. Gao, Z. Li, *et al.*, "Static inference meets deep learning: A hybrid type inference approach for python," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2019–2030.

[14] K. Jesse, P. T. Devanbu, and T. 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*, 2021, pp. 1483–1486.

[15] Z. Feng, D. Guo, D. Tang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[16] A. M. Mir, E. Latoškinas, and G. Gousios, "Manytypes4py: A benchmark python dataset for machine learning-based type inference," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, pp. 585–589.

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

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

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

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

[21] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "Typewriter: Neural type prediction with search-based validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 209–220.

[22] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "Lambdanet: Probabilistic type inference using graph neural networks," *arXiv preprint arXiv:2005.02161*, 2020.

[23] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, 2015.

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

[25] Microsoft, *Typescript*, 2022. [Online]. Available: `https://github.com/microsoft/TypeScript`.

[26] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[27] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Deep similarity learning-based type inference for python," *arXiv preprint arXiv:2101.04470*, 2021.

[28] J. Wei, G. Durrett, and I. Dillig, "Typet5: Seq2seq type inference using static analysis," *arXiv preprint arXiv:2303.09564*, 2023.

[29] OpenAI, *Chatgpt: Conversational ai model*, `https://openai.com/research/chatgpt`, Accessed: Month Day, Year, 2021.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[31] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[32] Y. Liu, M. Ott, N. Goyal, *et al.*, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[33] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[34] H. Xuan, A. Stylianou, and R. Pless, “Improved embeddings with easy positive triplet mining,” in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020, pp. 2474–2482.

[35] A. Mir, *Libsa4py*, version 0.3.0, Apr. 8, 2023. [Online]. Available: `https://github.com/saltudelft/libsa4py`.

[36] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *arXiv preprint arXiv:1508.07909*, 2015.

[37] Meta’s Fundamental AI Research group, *Faiss*, version 1.7.2, Jan. 11, 2022. [Online]. Available: `https://faiss.ai/`.

[38] A. Hermans, L. Beyer, and B. Leibe, “In defense of the triplet loss for person re-identification,” *arXiv preprint arXiv:1703.07737*, 2017.