Approaches for Type Inference in Python

Xinchen Jin
ShanghaiTech University
jinxch@shanghaitech.edu.cn

Abstract

Dynamic languages like Python and JavaScript are some of the most popular programming languages today. Without static typing, coding would be more flexible, but it may cause run-time exceptions and weak IDE support as well. To alleviate this, dynamic languages often has optional type annotations. If it is possible to automatically infer types, it may be beneficial in many different fields like automatic test case generation, bug detection and grammar checking. Existing static analysis tools often have limited support for complex cases, dynamic features and duck typing. We propose a deep learning approach aiming to mitigate this problem with higher efficiency and accuracy.

1. Introduction

In recent years, dynamic typed languages such as Python and JavaScript have been widely used. According to Github Octoverse 2022, JavaScript stays the most used language and Python grows with a 22.5% increase. It is known that statically-typed languages are less error-prone[23], on contrast, dynamic language feature makes it friendly for agile development, but in the meantime introduces type checking issues. Take Python as an example, this dynamic typed language allows variables to take different types of values during execution. Inheritance, duck-typing and other features of the language makes it difficult to infer the accurate variable type. According to Chen et al. [5], the misuse of the dynamic typing produces hidden bugs and reduces software quality. Consequently, developers need to add a lot of assertions and exception handlers to mitigate the problem.

We focus our language to python as it is becoming more and more popular these days, with the development of deep learning. To mitigate the problems, the Python community also proposed PEP 484[29], aiming to add optional static typing annotations. However, most of the static tools usually run on the whole program, meaning that they are less efficient. Existing static analysis tools of python programs mainly has 3 types of methods[11]: pattern matching, AST matching and symbolic execution. Existing static analysis tools are imprecise[19], which means they lack the ability to identify dynamic types, mainly because of over-approximation for soundness.

To address the limitations of static type inference tools for Python, researchers have adopted many Machine Learning and Deep Learning techniques for type predictions in Python to mitigate it[2, 4, 6, 20, 31]. The experimental results of these studies show that ML/DL-based type prediction approaches are more precise than static type inference methods or they can also recommend for static methods[20].

Despite the progressiveness of the learning based approaches, it still faces the following limitations:

- For code analysis tools, it needs to guarantee soundness. Existing DL tools provide type annotations in a list of candidates[22]. Or they will provide some results that contradict constraint rules thus cannot pass the type checker[2]. Some existing work [22] uses search-based methods to filter out the incorrect types but they cannot correct it.
- Rare types are the types that occurs less frequently in datasets[2]. The vocabulary distribution are likely to be a long-tailed distribution[6], meaning that these models may not perform well when encountering user-defined types. Low frequency vocabularies are often the bottleneck for learning-based models. For example, Typilus's accuracy drops by more than 50% for the types with occurrence frequencies fewer than 100, compared to the accuracy of the types with occurrence frequencies more than 10,000.

Motivated by the above discussion, We hope to find novel approaches to solve the type inference problem in an efficient and accurate way. Here we propose our way of combining deep learning with program analysis. Our objective is to develop deep learning models for type inference in dynamic typed programming languages, and enhance the accuracy and efficiency through our deep learning technique. Also we would like to evaluate and compare our model performance with existing approaches. For the challenges of the above analysis, we will adjust the network model and structure to cope, for example, for the contradictory constraint rules that may appear in type discrimination, we will adjust different models to learn the contradictory constraint rules between variables, and using the attention mechanism, ensemble learning, transfer learning and so on to help solve the problem of long tail distribution of data. In addition, we will attempt to utilize the existing large language models(LLM), such as GPT to solve the type inference problem. LLM is already equipped with excellent text analysing ability meanwhile being free from training costs. We will employ preprocessing steps to extract useful information about the code snippets, design appropriate prompts to motivate the understanding potential of LLM.

2

2. Related Work

2.1. Type inference in dynamic languages

For dynamic typed languages, Python and JavaScript are most popular. There exists many learning-based tools for JavaScript, because the type annotations can be obtained from TypeScript - a typed version of JavaScript. For example, TSLint[28] is a tool specifically designed for static analysis and code quality checking of TypeScript[16] code. It is able to detect and correct potential type errors while providing a configurable set of rules for enforcing code style and best practices. ESLint[26] is a widely used static code analysis tool for JavaScript and TypeScript. ESLint can be used to perform type checking and support custom rules through plugins and configuration files. Flow[9] is a JavaScript static type checker developed by Facebook to help developers catch potential type errors in their JavaScript code. By adding type annotations, Flow is able to analyze the code and find type mismatches, undefined variables, and other potential problems.

2.2. Python type inference

Python language standards have proposed type hints and annotations: PEP 484[29] and PEP 526[24]. Based on this, a number of static type checkers have been developed, including MyPy[27], pyright[17], pyre-check[15], PyType[10] and so on. These checkers use type annotations to analyze the code and verify the consistency of types. Manual annotations are expensive, as they require human work.

The previous mentioned tools are all static analysis tools, meaning that they have the result before we actually run the program. Some other tools may be dynamic analysis, which means they return types observed at runtime. Some of the dynamic analysis tools include pysonar2[30], PyAnnotate[8]. Dynamic analysis infer type annotation during runtime, meaning that the result is input sensitive. Also, for some real-world scenario, this approach is not applicable.

2.3. Learning based type inference

Traditional type inference tools mainly use rule-based formal methods. Xu et al.[31] designed a probablistic model which for each variable output the corresponding type annotation ranking by the probability. This model combines formal methods with probabilistic model. Hellendoorn et al. [12] regard the task as a sequence of annotation tasks similar to NLP. Their work however processes each variable independently, meaning that it doesn't have constraints on each variable. Typewriter[22] is the first combination of probabilistic type prediction with search-based refinement of predicted types. They use 4 separate sequence models to infer function types in Python. Typilus[2] propose a graph neural network to map variables to a type space and perform kNN on the type space to predict types. Type4Py[18] is a deep similarity learning-based hierarchical neural network model. It learns to discriminate between similar and dissimilar types in a high-dimensional space, which results in clusters of types. For the learning-based model, it faces the rare type prediction problem. Hityper[20] alleviates this problem by integrating deep learning models in static analysis framework. The DL model is consulted when static analysis cannot predict the type.

2.4. Learning based code embeddings

In order to applying deep learning model to learn from type hints, the code snippet should be represented as real-valued vectors by embedding techniques. The vectors record semantic similarities between similar words. Conventional embeddings approach builds up a dictionary on frequently used tokens, and generates embeddings for every token, which guarantees that every token are reserved when feeding to the model. For example Word2Vec[14], retrieve the vectorization representation of each token. Being motivated by Word2Vec, Code2Vec[3] is presented, which is a neural model specialized for representing snippets of code as continuous distributed vectors. It achieves improvement of predicting method names than previous general language embedding techniques.

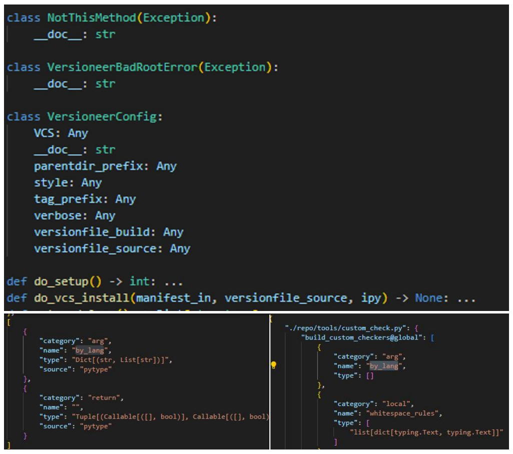
For previous type inference works, Word2Vec embeddings are employed by Type4Py[18] to train a code embedding for both code context and identifier tokens. General word embedding approaches might not be suitable for source code with plenty of user-defined variable names and function names. To avoid issue that model is unable to capture information from OOV, PyInfer[6] employed the BPE[25] algorithm to generate embeddings for source code, in order to capture semantics in variable and function names. Hityper[20] combined these two techniques: embedding two types and the variable names by Word2Vec, splitting OOV tokens into subtokens by the the BPE algorithm.

3. Our Approach

3.1. Static Approaches

We focus on 2 most advanced tools: Hityper[20] and PyType[10] from Google.

- Hityper: Hityper is a hybrid type inference approach based on both static inference and deep learning. Specifically, it records type dependencies among variables in each function and encode the dependency information in type dependency graphs (TDGs). Based on TDGs, they conduct static inference and type rejection rules to inspect the correctness of neural predictions. In short, deep learning models serve as the role of a consultant.
- PyType: PyType is a static analyzer that checks and infers types for Python code without requiring type annotations. Different from other type checkers, PyType is lenient and uses inference instead of gradual typing. It will infer types on code even when the code has no type hints on it.



```python
class NotThisMethod(Exception):
    __doc__: str

class VersioneerBadRootError(Exception):
    __doc__: str

class VersioneerConfig:
    VCS: Any
    __doc__: str
    parentdir_prefix: Any
    style: Any
    tag_prefix: Any
    verbose: Any
    versionfile_build: Any
    versionfile_source: Any

def do_setup() -> int: ...
def do_vcs_install(manifest_in, versionfile_source, ipy) -> None: ...
```

```json
[
    {
        "category": "arg",
        "name": "by_lang",
        "type": "Dict[(str, List[str])]",
        "source": "pytype"
    },
    {
        "category": "return",
        "name": "",
        "type": "Tuple[(Callable[([], bool)], Callable[([], bool)
        "source": "pytype"
    }
]
```

```json
{
    "./repo/tools/custom_check.py": {
        "build_custom_checkers@global": [
            {
                "category": "arg",
                "name": "by_lang",
                "type": []
            },
            {
                "category": "local",
                "name": "whitespace_rules",
                "type": [
                    "list[dict[typing.Text, typing.Text]]"
                ]
```

Figure 1. Example of PyType pyi results and HiTyper results

We use these 2 tools to analyze some existing codebases and found the weakness of the them. For PyType, it generates corresponding ".pyi" files. The annoations are limited - it can only infer global variables, function argument types and return types, which means it cannot infer local variables inside function body. For HiTyper, we compare the results with PyType and found that the prediction results are inaccurate. We can see from figure 1 that HiTyper is unable to predict function arguments correctly. As a result, we come up with the idea to fuse the results of 2 analyzers to improve the performance. We first use the 2 tools to conduct static analysis on the target repository independently. The next step is to fuse the result of the 2 tools. One thing worth notice is we prefer results from PyType because based on our study to some example repositories, the result of PyType is more accurate while HiTyper's analyze results are more comprehensive.

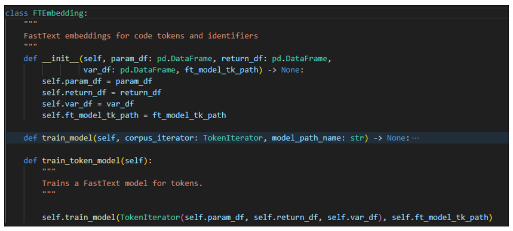
3.2. Learning Approach

3.2.1 Training Model

For this work, we mainly base on the pipeline of Type4Py[18]. Type4Py is a deep learning-based model for type inference in Python. It begins by extracting type hints from Python source code, including natural information like variable and function parameter names, as well as code context. These hints are then transformed into vector representations using Word2Vec. Type4Py employs a hierarchical neural network model, which includes two Long Short-Term Memory (LSTM) networks, to process these vectors. The model uses Deep Similarity Learning (DSL) to map types into a continuous space, creating type clusters. During the prediction phase, Type4Py employs a k-nearest neighbors approach to identify the closest types within these clusters, thus performing type inference. This process effectively enhances the model's accuracy and efficiency in type inference. We attempt to make modification to pivotal steps of the baseline model to improve the performance.

- **Preprocessing Step:**
  Type4Py extracts the Abstract Syntax Tree (AST) from source code files and obtains type hints that are valuable for predicting types of function arguments, variables, and return types. In order for machine learning models to learn from type hints, Word2Vec[14], as a common NLP technique, is applied to preprocess extracted identifiers and code contexts. As mentioned above, after the classical Word2Vec came out, more specialized embedding tools are proposed. Though CodedVec[3] is specially designed for code embedding, its purpose is to encode whole code snippet to one vector for functional prediction, which is not suitable for our purpose of encode single token for type prediction. Instead, through our investigation, fastText[13], as a improved text classification tool to train word vector representations, is more suitable for our task. Therefore, we employ fastText vectorization as a improvement to the preprocessing step.
- **Neural Model:**
  Type4Py employs a hierarchical neural network (HNN) consisting of two recurrent neural networks (RNNs) based on Long Short-Term Memory (LSTM) units. This setup is effective in capturing different aspects of identifiers and code context. The information captured by these RNNs is summarized into vectors, which are then concatenated with the visible type hints vector. Considering the purpose of effectively capture useful features, we attempt to employ different combinations of neural model techniques.
  The **Transformer** model is based on the self-attention mechanism, which is able to process all parts of the input data simultaneously, enables the model to capture global dependencies more effectively. In type inference

tasks, Transformers can more fully understand the context of the code and improve the accuracy of the prediction. **GRU** is another RNN that simplifies the structure of LSTM while maintaining similar performance. When dealing with code type inference, GRU can process sequence data quickly and efficiently while reducing computational complexity and training time; **Max pooling** is a common operation used to reduce the number of parameters in a neural network and reduce the risk of overfitting. When dealing with the code type inference problem, Max pooling can help the model to extract and retain the most important features and improve the generalization ability of the model.
Considering the advantages of these model, we modified the original neural models to more combinations, including adding max pooling layers. Through these attempts, we are able to compare the performance of neural models.



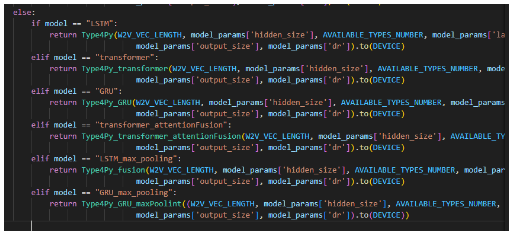
```
class FTEmbedding:
    """
    FastText embeddings for code tokens and identifiers
    """
    def __init__(self, param_df: pd.DataFrame, return_df: pd.DataFrame,
                 var_df: pd.DataFrame, ft_model_tk_path) -> None:
        self.param_df = param_df
        self.return_df = return_df
        self.var_df = var_df
        self.ft_model_tk_path = ft_model_tk_path

    def train_model(self, corpus_iterator: TokenIterator, model_path_name: str) -> None:···

    def train_token_model(self):
        """
        Trains a FastText model for tokens.
        """

        self.train_model(TokenIterator(self.param_df, self.return_df, self.var_df), self.ft_model_tk_path)
```

Figure 2. Embedding by FastText



```
else:
    if model == "LSTM":
        return Type4Py(W2V_VEC_LENGTH, model_params['hidden_size'], AVAILABLE_TYPES_NUMBER, model_params['la
                       model_params['output_size'], model_params['dr']).to(DEVICE)
    elif model == "transformer":
        return Type4Py_transformer(W2V_VEC_LENGTH, model_params['hidden_size'], AVAILABLE_TYPES_NUMBER, mode
                       model_params['output_size'], model_params['dr']).to(DEVICE)
    elif model == "GRU":
        return Type4Py_GRU(W2V_VEC_LENGTH, model_params['hidden_size'], AVAILABLE_TYPES_NUMBER, model_params
                       model_params['output_size'], model_params['dr']).to(DEVICE)
    elif model == "transformer_attentionFusion":
        return Type4Py_transformer_attentionFusion(W2V_VEC_LENGTH, model_params['hidden_size'], AVAILABLE_TY
                       model_params['output_size'], model_params['dr']).to(DEVICE)
    elif model == "LSTM_max_pooling":
        return Type4Py_fusion(W2V_VEC_LENGTH, model_params['hidden_size'], AVAILABLE_TYPES_NUMBER, model_par
                       model_params['output_size'], model_params['dr']).to(DEVICE)
    elif model == "GRU_max_pooling":
        return Type4Py_GRU_maxPoolint((W2V_VEC_LENGTH, model_params['hidden_size'], AVAILABLE_TYPES_NUMBER,
                       model_params['output_size'], model_params['dr']).to(DEVICE))
```

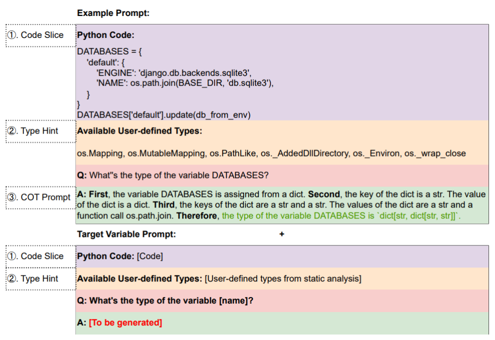
Figure 3. Combinations of Neural Models

3.2.2 Prompt + LLM

For the above model, although good prediction results can be obtained, the interpretability is not strong, so we introduce the method prompt + LLM here to try to infer the type. We use TYPEGEN[21] as a template in this project, which is a generative type inference method based on static domain knowledge from static analysis.

TYPEGEN first generates prompts with domain knowledge and then feeds them into a language model for type prediction. To achieve this goal, TYPEGEN employs the widely used contextual learning approach [7]. The method provides some example questions and answers as a demonstration of the language model, and then asks for the answer to a new question. Using this approach, TYPEGEN builds input hints by adding some example hints (example questions and answers) with domain knowledge before the target variable hints (new questions).

The domain-aware example prompts 4 include three parts: code slice, type hint and COT prompt. In particular, the code slice extracts the statements that play a role in determining the type for the target variable, while eliminating any unrelated statements. The type hint incorporates external knowledge relevant to various code slices, encompassing user-defined types and third-party types. The COT prompt outlines the steps of static analysis for inference, with the goal of instructing language models in type inference. The target variable prompt consists solely of the code slice and the type hint associated with the target variable.



Example Prompt:

①. Code Slice

Python Code:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
DATABASES['default'].update(db_from_env)
```

②. Type Hint

Available User-defined Types:

os.Mapping, os.MutableMapping, os.PathLike, os._AddedDllDirectory, os._Environ, os._wrap_close

Q: What''s the type of the variable DATABASES?

③. COT Prompt

A: First, the variable DATABASES is assigned from a dict. **Second**, the key of the dict is a str. The value of the dict is a dict. **Third**, the keys of the dict are a str and a str. The values of the dict are a str and a function call os.path.join. **Therefore**, the type of the variable DATABASES is `dict[str, dict[str, str]]`.

Target Variable Prompt: +

①. Code Slice

Python Code: [Code]

②. Type Hint

Available User-defined Types: [User-defined types from static analysis]

Q: What's the type of the variable [name]?

A: [To be generated]

Figure 4. Domain-aware example prompts

To initiate the process, TYPEGEN utilizes a set of annotated Python source files for selecting examples and designates a target Python source file where the target variable is situated. For each target source file, TYPEGEN creates an input prompt that integrates domain-aware example prompts and the target variable prompt. Subsequently, a language model is employed to generate the COT prompt, encompassing both the predicted type and corresponding explanations. The generation of domain-aware example prompts involves three phases: (1) code slicing, (2) type hint collection, and (3) COT prompt construction to generate the code slice, type hint, and COT prompt, respectively. Finally, in the (4) Type Generation phase, TYPEGEN utilizes in-context learning to deduce the types of target variables. We provide an overview of the TYPEGEN workflow in 6.

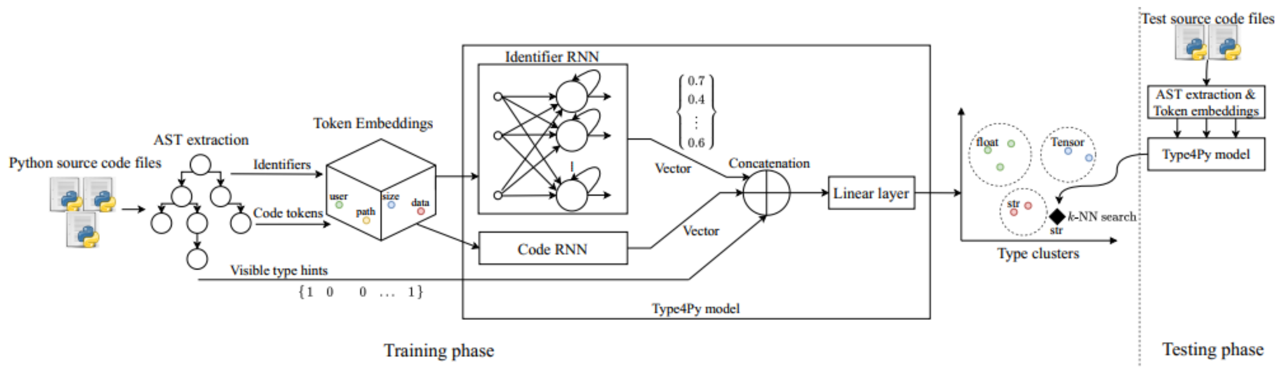


Figure 5. The overview of Type4Py

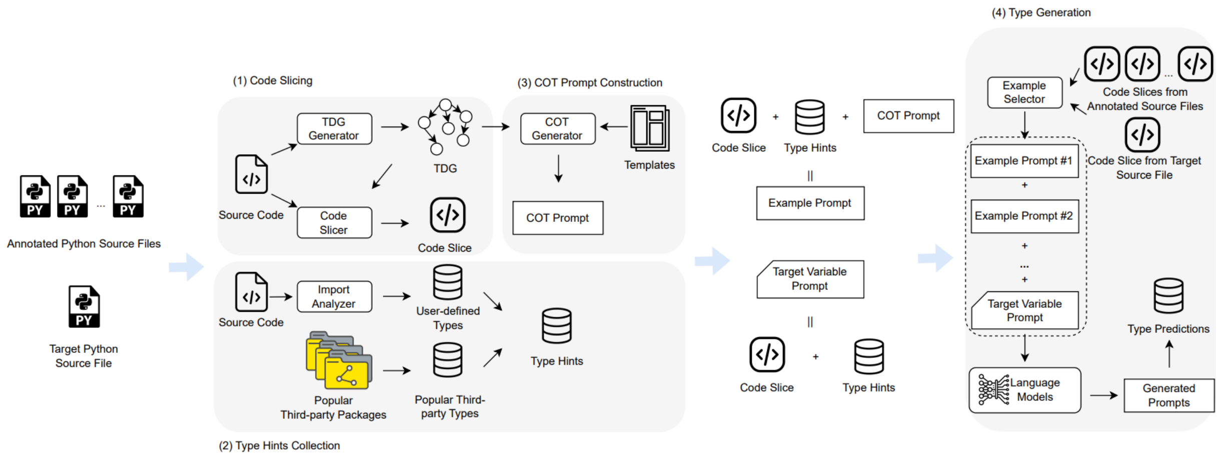


Figure 6. The overview of TYPEGEN

4. Experiment

In this paragraph, we present the first line specific details of our implementation and some main experimental results.

4.1. Dataset

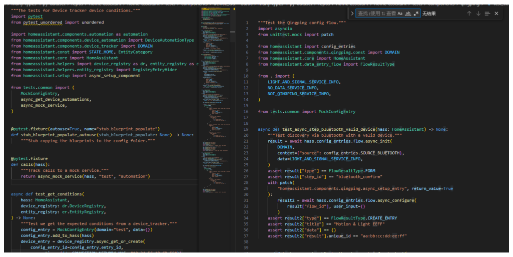
Most of the papers about type inference use the Many-Types4Py dataset[18] or its variants. The production pipeline of the dataset is the following:

1. 1. Clone popular repositories from GitHub
2. 2. Generate duplicate tokens using ‘CD4Py‘[1] tool
3. 3. Deduplicate (remove files)
4. 4. LibSA4Py: a static analysis library for python, which extracts type hints and features for training ML-based type inference models.

One of the issues is the CD4Py tool. The CD4Py code de-duplication tool uses the following procedure to identify duplicate files in a Python code corpus:

1. Tokenize all the source code files in the code corpus using tokenize module of Python standard library.
2. Preprocess tokenized source files by only selecting identifier tokens and removing language keywords.
3. Convert pre-processed tokenized files to a vector representation using the TF-IDF method.
4. Perform k-nearest neighbor search to find k candidate duplicate files for each source code file. Next, filter out candidate duplicate files by considering the threshold t.
5. Find clusters of duplicate source code files while assuming that similarity is transitive.

k-NN is a relatively traditional clustering method, and it may not be a perfect fit for code de-duplication. Besides, the CD4Py tool ignores language keywords and only focuses on identifier tokens, which may produce false positives. We reproduce its data generation steps and look into the generated datasets closely. We can observe from 7 that CD4Py mispredicts 2 test classes as duplication, while in reality they perform tests on different components. Removing instances from a dataset can lead to an imbalance in the training data. This can affect the model’s ability to learn patterns accurately and may result in biased predictions.



```
"""The tests for Device tracker device conditions."""
import pytest
from pytest_unordered import unordered

import homeassistant.components.automation as automation
from homeassistant.components.device_automation import DeviceAutomationType
from homeassistant.components.device_tracker import DOMAIN
from homeassistant.const import STATE_HOME, EntityCategory
from homeassistant.core import HomeAssistant
from homeassistant.helpers import device_registry as dr, entity_registry as e
from homeassistant.helpers.entity_registry import RegistryEntryHider
from homeassistant.setup import async_setup_component

from tests.common import (
    MockConfigEntry,
    async_get_device_automations,
    async_mock_service,
)


@pytest.fixture(autouse=True, name="stub_blueprint_populate")
def stub_blueprint_populate_autouse(stub_blueprint_populate: None) -> None:
    """Stub copying the blueprints to the config folder."""


@pytest.fixture
def calls(hass):
    """Track calls to a mock service."""
    return async_mock_service(hass, "test", "automation")


async def test_get_conditions(
    hass: HomeAssistant,
    device_registry: dr.DeviceRegistry,
    entity_registry: er.EntityRegistry,
) -> None:
    """Test we get the expected conditions from a device_tracker."""
    config_entry = MockConfigEntry(domain="test", data={})
    config_entry.add_to_hass(hass)
    device_entry = device_registry.async_get_or_create(
        config_entry_id=config_entry.entry_id,
```

```
"""Test the Qingping config flow."""
import asyncio
from unittest.mock import patch

from homeassistant import config_entries
from homeassistant.components.qingping.const import DOMAIN
from homeassistant.core import HomeAssistant
from homeassistant.data_entry_flow import FlowResultType

from . import (
    LIGHT_AND_SIGNAL_SERVICE_INFO,
    NO_DATA_SERVICE_INFO,
    NOT_QINGPING_SERVICE_INFO,
)

from tests.common import MockConfigEntry


async def test_async_step_bluetooth_valid_device(hass: HomeAssistant) -> None:
    """Test discovery via bluetooth with a valid device."""
    result = await hass.config_entries.flow.async_init(
        DOMAIN,
        context={"source": config_entries.SOURCE_BLUETOOTH},
        data=LIGHT_AND_SIGNAL_SERVICE_INFO,
    )
    assert result["type"] == FlowResultType.FORM
    assert result["step_id"] == "bluetooth_confirm"
    with patch(
        "homeassistant.components.qingping.async_setup_entry", return_value=True
    ):
        result2 = await hass.config_entries.flow.async_configure(
            result["flow_id"], user_input={}
        )
    assert result2["type"] == FlowResultType.CREATE_ENTRY
    assert result2["title"] == "Motion & Light EEFF"
    assert result2["data"] == {}
    assert result2["result"].unique_id == "aa:bb:cc:dd:ee:ff"
```

Figure 7. Example of CD4Py proposed incorrect duplication

4.2. Main results

4.2.1 Static analysis

For static analysis, We don't have an existing preprocessed dataset to conduct our experiment. Due to the limitations of computing resources, we extracted 10 repositories from ManyTypes4Py dataset and cloned them from GitHub. We then compared our static analysis results with the dataset's ground truth manually. We found that our fused static results are more accurate on type inference, but lacks some coverage. Moreover, we compared our analysis results with learning approach's results, and find that static results are sounder and more efficient.

4.2.2 Learning Approaches

To compare the performance with baseline, we adopt the evaluation metrics from Type4Py:

exact match: type prediction t_p and ground truth t_g are exactly the same type.

parametric match: ignores all type parameters and only matches the base types.

common & rare types: types that we have seen more than 100 times in the train set as common or rare otherwise.

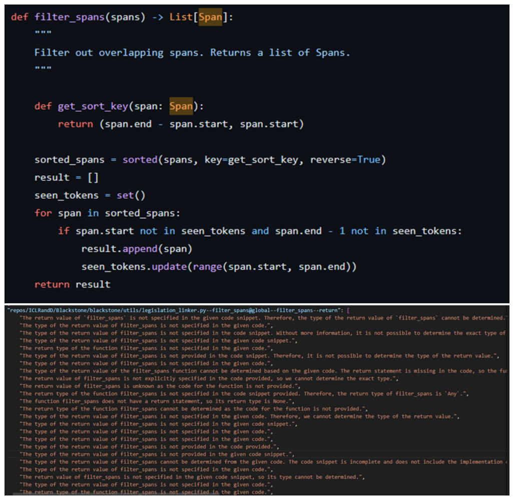
ubiquitous types: {str, int, list, bool, float}.

In addition, MRR metric is employed:

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

The MRR metric partially rewards the neural models by giving a score of $\frac{1}{r_i}$ to a prediction if the correct type annotation appears in rank r. For example, Top-1 accuracy, a score of 1 is given to a prediction for which the Top-1 suggested type is correct.

- Table 1 shows the results of our combination of different encoder models and fusion ways on the Manytype4py[18] test set. The first row of the table shows the results of training and testing our replicated type4py[] model on a Manytype4py[18] clear dataset. We can see that combining different encoders and fusion methods has a certain impact on the performance of the model.
- Table 2 shows the huge impact of different embedding methods on the performance of the model. This suggests that putting the code into a better data format gives the model more information to learn.
- Table 3 shows the excellent performance of our best model on different evaluation metrics.



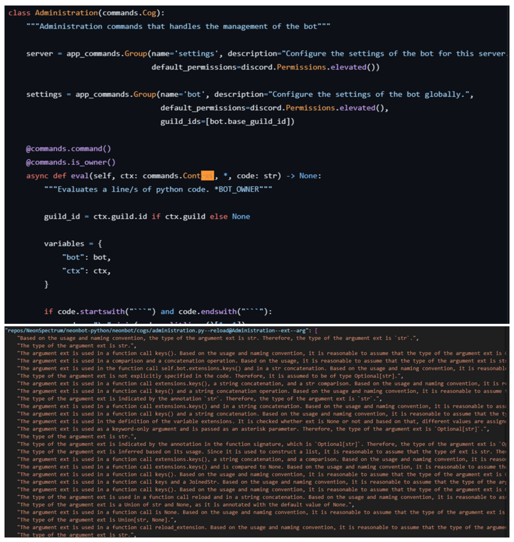
```
def filter_spans(spans) -> List[Span]:
    """
    Filter out overlapping spans. Returns a list of Spans.
    """

    def get_sort_key(span: Span):
        return (span.end - span.start, span.start)

    sorted_spans = sorted(spans, key=get_sort_key, reverse=True)
    result = []
    seen_tokens = set()
    for span in sorted_spans:
        if span.start not in seen_tokens and span.end - 1 not in seen_tokens:
            result.append(span)
            seen_tokens.update(range(span.start, span.end))
    return result
```

```
"repos/ICLRandD/Blackstone/blackstone/utils/legislation_linker.py--filter_spans@global--filter_spans--return": [
    "The return value of `filter_spans` is not specified in the given code snippet. Therefore, the type of the return value of `filter_spans` cannot be determined.
    "The type of the return value of filter_spans is not specified in the given code.",
    "The type of the return value of filter_spans is not specified in the code snippet. Without more information, it is not possible to determine the exact type of
    "The type of the return value of filter_spans is not specified in the given code snippet.",
    "The return type of the function filter_spans is not specified in the given code.",
    "The type of the return value of filter_spans is not provided in the code snippet. Therefore, it is not possible to determine the type of the return value.",
    "The type of the return value of filter_spans is not specified in the given code.",
    "The type of the return value of the filter_spans function cannot be determined based on the given code. The return statement is missing in the code, so the fu
    "The return value of filter_spans is not explicitly specified in the code provided, so we cannot determine the exact type.",
    "The return value of filter_spans is unknown as the code for the function is not provided.",
    "The return type of the function filter_spans is not specified in the code snippet provided. Therefore, the return type of filter_spans is `Any`.",
    "The function filter_spans does not have a return statement, so its return type is None.",
    "The return type of the function filter_spans cannot be determined as the code for the function is not provided.",
    "The type of the return value of filter_spans is not specified in the given code. Therefore, we cannot determine the type of the return value.",
    "The type of the return value of filter_spans is not specified in the given code snippet.",
    "The type of the return value of filter_spans is not specified in the given code.",
    "The type of the return value of filter_spans is not specified in the given code.",
    "The type of the return value of filter_spans is not provided in the code provided.",
    "The type of the return value of filter_spans is not provided in the given code snippet.",
    "The type of the return value of filter_spans cannot be determined from the given code. The code snippet is incomplete and does not include the implementation
    "The type of the return value of filter_spans is not specified in the given code.",
    "The return value of filter_spans is not specified in the given code snippet, so its type cannot be determined.",
    "The type of the return value of filter_spans is not specified in the given code.",
    "The return type of the function filter_spans is not specified in the given code.",
```

Figure 8. Badcase: Annotation is omitted



```
class Administration(commands.Cog):
    """Administration commands that handles the management of the bot"""

    server = app_commands.Group(name='settings', description="Configure the settings of the bot for this server
                                default_permissions=discord.Permissions.elevated())

    settings = app_commands.Group(name='bot', description="Configure the settings of the bot globally.",
                                  default_permissions=discord.Permissions.elevated(),
                                  guild_ids=[bot.base_guild_id])

    @commands.command()
    @commands.is_owner()
    async def eval(self, ctx: commands.Context, *, code: str) -> None:
        """Evaluates a line/s of python code. *BOT_OWNER"""

        guild_id = ctx.guild.id if ctx.guild else None

        variables = {
            "bot": bot,
            "ctx": ctx,
        }

        if code.startswith("```") and code.endswith("```"):
```

```
"repos/NeonSpectrum/neonbot-python/neonbot/cogs/administration.py--reload@Administration--ext--arg": [
    "Based on the usage and naming convention, the type of the argument ext is str. Therefore, the type of the argument ext is `str`.",
    "The type of the argument ext is str.",
    "The argument ext is used in a function call keys(). Based on the usage and naming convention, it is reasonable to assume that the type of the argument ext is
    "The argument ext is used in a comparison and a concatenation operation. Based on the usage, it is reasonable to assume that the type of the argument ext is st
    "The argument ext is used in the function call self.bot.extensions.keys() and in a str concatenation. Based on the usage and naming convention, it is reasonabl
    "The type of the argument ext is not explicitly specified in the code. Therefore, it is assumed to be of type Optional[str].",
    "The argument ext is used in a function call extensions.keys(), a string concatenation, and a str comparison. Based on the usage and naming convention, it is r
    "The argument ext is used in a function call keys() and a string concatenation operation. Based on the usage and naming convention, it is reasonable to assume
    "The type of the argument ext is indicated by the annotation `str`. Therefore, the type of the argument ext is `str`.",
    "The argument ext is used in a function call extensions.keys() and in a string concatenation. Based on the usage and naming convention, it is reasonable to ass
    "The argument ext is used in a function call keys() and a string concatenation. Based on the usage and naming convention, it is reasonable to assume that the t
    "The argument ext is used in the definition of the variable extensions. It is checked whether ext is None or not and based on that, different values are assign
    "The argument ext is used as a keyword-only argument and is passed as an asterisk parameter. Therefore, the type of the argument ext is `Optional[str]`.",
    "The type of the argument ext is str.",
    "The type of the argument ext is indicated by the annotation in the function signature, which is `Optional[str]`. Therefore, the type of the argument ext is `O
    "The type of the argument ext is inferred based on its usage. Since it is used to construct a list, it is reasonable to assume that the type of ext is str. The
    "The argument ext is used in a function call extensions.keys(), a string concatenation, and a comparison. Based on the usage and naming convention, it is reaso
    "The argument ext is used in a function call extensions.keys() and is compared to None. Based on the usage and naming convention, it is reasonable to assume th
    "The argument ext is used in a function call keys(). Based on the usage and naming convention, it is reasonable to assume that the type of the argument ext is
    "The argument ext is used in a function call keys and a JoinedStr. Based on the usage and naming convention, it is reasonable to assume that the type of the ar
    "The argument ext is used in a function call keys(). Based on the usage and naming convention, it is reasonable to assume that the type of the argument ext is
    "The type of the argument ext is used in a function call reload and in a string concatenation. Based on the usage and naming convention, it is reasonable to as
    "The type of the argument ext is a Union of str and None, as it is annotated with the default value of None.",
    "The argument ext is used in a function call is None. Based on the usage and naming convention, it is reasonable to assume that the type of the argument ext is
    "The type of the argument ext is Union[str, None].",
    "The argument ext is used in a function call reload_extension. Based on the usage and naming convention, it is reasonable to assume that the type of the argume
    "The type of the argument ext is str.",
```

Figure 9. Badcase: Missing information when child class inherits from parent class

For prompt + LLM, since gpt's API is charged per token, we only perform type inference on a small scale. We performed this on chat-gpt3.5 and chat-gpt4 and obtained the results in table 4.

We also analyzed the bad cases and found that the common misinference cases fell into the following categories:

model + fusion way	**all**	**ubiquitous**	**common**	**rare**
LSTM + select last token (type4py)	71.4	100.0	71.9	12.3
transformer + select last token	69.5	100.0	66.7	9.8
LSTM + max pooling	72.3 (+0.9)	100.0	74.8 (+2.9)	12.7 (+0.4)
transformer + max pooling	70.1	100.0	68.3	10.5

Table 1. Top 1 accuracy and exact matching for different models and fusion methods.

model + fusion way + embedding way	**all**	**ubiquitous**	**common**	**rare**
LSTM + select last token + word2vec (type4py)	71.4	100.0	71.9	12.3
LSTM + select last token + fastText	75.3 (+3.9)	100.0	81.4 (+9.3)	17.8 (+5.5)

Table 2. Top 1 accuracy and exact matching for different models and embedding methods.

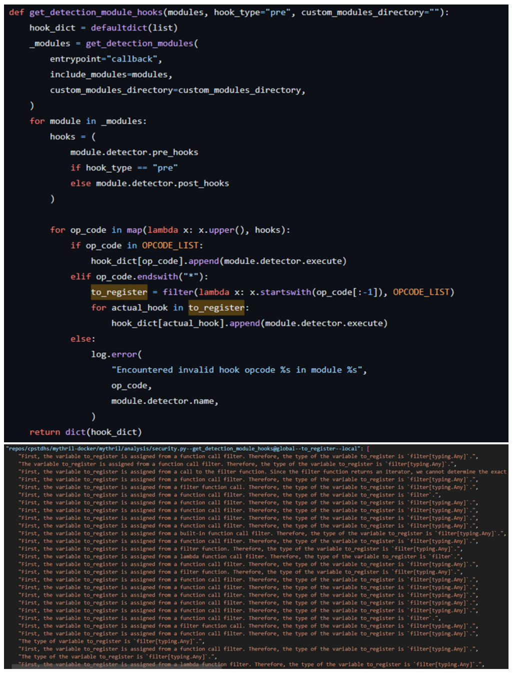
our best model	**accuracy (%)**			**MRR**		
	top 1	**top 3**	**top 5**	**top 1**	**top 3**	**top 5**
Exact match - all	76.0 (+4.6)	78.6	79.2	76.0	77.2	77.3
Exact match - ubiquitous	100.0	100.0	100.0	100.0	100.0	100.0
Exact match - common	82.6 (+10.7)	88.0	89.3	82.6	85.1	85.4
Exact match - rare	19.6 (+7.3)	24.4	25.6	19.6	21.7	22.0
Parametric match - all	81.1	84.5	85.4	72.0	74.1	74.5
Parametric match - common	85.9	91.4	92.8	74.8	79.2	80.1
Parametric match - rare	37.1	45.2	47.3	27.2	30.1	30.4

Table 3. Evaluation metrics for our best model, showing accuracy and mean reciprocal rank (MRR).

gpt + data	**Top-1**	**Top-2**	**Top-3**	**Top-4**	**Top-5**
gpt3.5_25	0.84	0.84	0.88	0.88	0.88
gpt4_25	0.76	0.84	0.88	0.88	0.88
gpt3.5_20	0.761	0.81	0.81	0.81	0.81
gpt4_20	0.85	0.85	0.85	0.85	0.85
gtp3.5_100	0.78	0.83	0.85	0.86	0.87

Table 4. Accuracy of prompt + LLM on small-scale data.

- Annotation is omitted 8: We observe that there are annotations for function return values in the original code, but the inferred return information is considered insufficient for type inference, which may be caused by the loss of annotation information due to errors in the process of generating hints from static analysis.
- Missing information occurs when a child class inherits from its parent class 9: We observe that what is needed for type inference in the original code is the return value of the function that inherits from the superclass, and then the type inference error is caused by the missing information from the superclass.
- External library function information is missing 10: We observe that what needs to be inferred in the source code is the return value of the function of the external library, and then the type inference error is caused by the missing information from the external library, but the typehint part of TYPEGEN has been supplemented with the information of the external library and the user-defined type.



```
def get_detection_module_hooks(modules, hook_type="pre", custom_modules_directory=""):
    hook_dict = defaultdict(list)
    _modules = get_detection_modules(
        entrypoint="callback",
        include_modules=modules,
        custom_modules_directory=custom_modules_directory,
    )
    for module in _modules:
        hooks = (
            module.detector.pre_hooks
            if hook_type == "pre"
            else module.detector.post_hooks
        )

        for op_code in map(lambda x: x.upper(), hooks):
            if op_code in OPCODE_LIST:
                hook_dict[op_code].append(module.detector.execute)
            elif op_code.endswith("*"):
                to_register = filter(lambda x: x.startswith(op_code[:-1]), OPCODE_LIST)
                for actual_hook in to_register:
                    hook_dict[actual_hook].append(module.detector.execute)
            else:
                log.error(
                    "Encountered invalid hook opcode %s in module %s",
                    op_code,
                    module.detector.name,
                )
    return dict(hook_dict)
```

```
"repos/cpstdhs/mythril-docker/mythril/analysis/security.py--get_detection_module_hooks@global--to_register--local": [
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "The variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a call to the filter function. Since the filter function returns an iterator, we cannot determine the exact
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a filter function call. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a built-in function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a filter function. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a lambda function call filter. Therefore, the type of the variable to_register is `filter`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a filter function. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter`.",
   "First, the variable to_register is assigned from a filter function call. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "The type of variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a function call filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
   "The type of the variable to_register is `filter[typing.Any]`.",
   "First, the variable to_register is assigned from a lambda function filter. Therefore, the type of the variable to_register is `filter[typing.Any]`.",
```

Figure 10. Badcase: External library function information is missing

It may be that an error has been generated during the generation of the typehint.

5. Discussion and future work

Based on our previous studies and experiments, we provide the following remarks:

- We study the data distribution of the existing dataset ManyTypes4Py[18] and find that the data distribution may be biased. For future work, it can devise a more accurate way to identify code duplication and generate a more balanced dataset.
- We fuse PyType[10] and HiTyper's[20] static analysis result to improve the prediction coverageand improve the prediction accuracy.
- In the context of our future research endeavors, it is evident from our experimental results that the choice of text embedding methods has a significant impact on model performance. Therefore, as part of our upcoming research plans, we intend to focus on the development and training of an embedding method specifically designed for text related to code. This specialized approach is poised to optimize the utilization of code-related information, potentially leading to substantial enhancements in model performance.
- For prompt + LLM approach, we observe that the model has a limited throughput. We could try to save the number of tokens while preserving model performance in the future. Based on the analysis of badcase, we will try to optimize the generation process of prompt, such as the generation process of typehint, COT, etc. to achieve the goal of improving the accuracy. We also observe that there are some redundant and repeated information in the generated prompt at present. In the future we will consider filtering it to see if there is a positive impact on the effect.

References

[1] Cd4py - a code de-duplication tool for python programming language, 2020. 5

[2] 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. 1, 2

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, 2019. 2, 3

[4] Casper Boone, Niels de Bruin, Arjan Langerak, and Fabian Stelmach. Dltpy: Deep learning type inference of python function signatures using natural language context. *arXiv preprint arXiv:1912.00680*, 2019. 1

[5] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. An empirical study on dynamic typing related practices in python systems. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 83–93, 2020. 1

[6] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. Pyinfer: Deep learning semantic type inference for python variables. *arXiv preprint arXiv:2106.14316*, 2021. 1, 2

[7] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. A Survey on In-context Learning. *arXiv e-prints*, art. arXiv:2301.00234, 2022. 4

[8] Dropbox. Pyannotate - auto-generate pep-484 annotations, 2021. 2

[9] Facebook. Flow - a static type checker for javascript, 2023. 2

[10] Google. Pytype - a static type analyzer for python code, 2024. 2, 3, 8

[11] Hristina Gulabovska and Zoltán Porkoláb. Survey on static analysis tools of python programs. In *SQAMIA*, 2019. 1

[12] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis 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*, pages 152–162, 2018. 2

[13] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification, 2016. 3

[14] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents, 2014. 2, 3

[15] Meta. pyre-check - performant type-checker for python 3, 2024. 2

[16] Microsoft. Typescript - javascript that scales, 2024. 2

[17] Microsoft. pyright - static type checker for python, 2024. 2

[18] 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. 2, 3, 5, 6, 8

[19] Zvonimir Pavlinovic. *Leveraging Program Analysis for Type Inference*. PhD thesis, New York University, 2019. 1

[20] 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. 1, 2, 3, 8

[21] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. Generative Type Inference for Python. *arXiv e-prints*, art. arXiv:2307.09163, 2023. 4

[22] Michael Pradel, Georgios Gousios, Jason Liu, and Satish 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*, pages 209–220, 2020. 1, 2

[23] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165, 2014. 1

[24] Ivan Levkivskyi Lisa Roach Guido van Rossum Ryan Gonzalez, Philip House. Pep 526 – syntax for variable annotations. Online, 2016. 2

[25] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016. 2

[26] ESLint Team. Eslint - pluggable javascript linter, 2023. 2

[27] Mypy Team. Mypy: Optional static typing for python, 2023. 2

[28] Palantir Technologies. Tslint - an extensible linter for the typescript language, 2019. 2

[29] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Pep 484 – type hints. Online, 2014. 1, 2

[30] Yin Wang. pysonar2 - a semantic indexer for python with interprocedural type inference, 2022. 2

[31] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 607–618, 2016. 1, 2