TypeWriter: Neural Type Prediction with Search-based Validation

Michael Pradel*  
University of Stuttgart  
michael@binaervarianz.de  

Jason Liu  
jasonliu@fb.com  
Facebook  

Georgios Gousios*  
Delft University of Technology  
g.gousios@tudelft.nl  

Satish Chandra  
schandra@acm.org  
Facebook

ABSTRACT
Maintaining large code bases written in dynamically typed languages, such as JavaScript or Python, can be challenging due to the absence of type annotations: simple data compatibility errors proliferate, IDE support is limited, and APIs are hard to comprehend. Recent work attempts to address these issues through either static type inference or probabilistic type prediction. Unfortunately, static type inference for dynamic languages is inherently limited, while probabilistic approaches suffer from imprecision. This paper presents TypeWriter, the first combination of probabilistic type prediction with search-based refinement of predicted types. TypeWriter’s predictor learns to infer the return and argument types for functions from partially annotated code bases by combining the natural language properties of code with programming language-level information. To validate predicted types, TypeWriter invokes a gradual type checker with different combinations of the predicted types, while navigating the space of possible type combinations in a feedback-directed manner.

We implement the TypeWriter approach for Python and evaluate it on two code corpora: a multi-million line code base at Facebook and a collection of 1,137 popular open-source projects. We show that TypeWriter’s type predictor achieves an F1 score of 0.64 (0.79) in the top-1 (top-5) predictions for return types, and 0.57 (0.80) for argument types, which clearly outperforms prior type prediction models. By combining predictions with search-based validation, TypeWriter can fully annotate between 14% to 44% of the files in a randomly selected corpus, while ensuring type correctness. A comparison with a static type inference tool shows that TypeWriter adds many more non-trivial types. TypeWriter currently suggests types to developers at Facebook and several thousands of types have already been accepted with minimal changes.

1 INTRODUCTION
Dynamically typed programming languages, such as Python and JavaScript, have become extremely popular, and large portions of newly written code are in one of these languages. While the lack of static type annotations enables fast prototyping, it often leads to problems when projects grow. Examples include type errors that remain unnoticed for a long time [9], suboptimal IDE support, and difficult to understand APIs [22]. To solve these problems, in recent years, many dynamic languages obtained support for type annotations, which enable programmers to specify types in a fashion similar to a statically typed language. Type annotations are usually ignored at runtime; nevertheless, they serve both as hints for developers using external APIs and as inputs to gradual type checkers that ensure that specific programming errors cannot occur. To cope with legacy code bases, type annotations can be introduced gradually; in such cases, the type checker will check only code that is annotated.

As manually annotating code is time-consuming and error-prone [26], developers must resort to automated methods. One way to address the lack of type annotations is type inference via traditional static analysis. Unfortunately, dynamic features, such as heterogeneous arrays, polymorphic variables, dynamic code evaluation, and monkey patching make static type inference a hard problem for popular dynamic languages, such as Python or JavaScript [7]. Static type inference tools typically handle these challenges by inferring a type only if it is certain or very likely (under some assumptions), which significantly limits the number of types that can be inferred.

Motivated by the inherent difficulties of giving definitive answers via static analysis, several probabilistic techniques for predicting types have been proposed. A popular direction is to exploit the existence of already annotated code as training data to train machine learning models that then predict types in not yet annotated code. Several approaches predict the type of a code entity, e.g., a variable or a function, from the code contexts in which this entity occurs [15, 28]. Other approaches exploit natural language information embedded in source code, e.g., variable names or comments, as a valuable source of informal type hints [21, 37].

While existing approaches for predicting types are effective in some scenarios, they suffer from imprecision and combinatorial explosion. Probabilistic type predictors are inherently imprecise because they suggest one or more likely types for each missing annotation, but cannot guarantee their correctness. The task of deciding which of these suggestions are correct is left to the developer. Because probabilistic predictors suggest a ranked list of likely types, choosing a type-correct combination of type annotations across multiple program elements causes combinatorial explosion. A naive approach would be to let a developer or a tool choose from all combinations of the predicted types. Unfortunately, this approach does not scale to larger code examples, because the number of type combinations to consider is exponential in the number of not yet annotated code entities.

This paper presents TypeWriter, a combination of learning-based, probabilistic type prediction and a feedback-directed, search-based validation of predicted types. The approach addresses the imprecision problem based on the insight that a gradual type checker can

*Work performed while on sabbatical at Facebook, Menlo Park.
Figure 1: Example of search for type-correct predicted types.

 pinpoint contradictory type annotations, which guides the selection of suitable types from the set of predicted types. To make the search for a consistent set of types tractable, we formulate the problem as a combinatorial search and present a search strategy that finds type-correct type annotations efficiently. TypeWriter makes use of the variety of type hints present in code through a novel neural architecture that exploits both natural language, in the form of identifier names and code comments, similar to prior work [21], and also programming context, in the form of usage sequences.

To illustrate the approach, consider the two-to-be-typed functions in Figure 1. Given this code, the neural type model of TypeWriter predicts a ranked list of likely types for each argument type and return type, as indicated by the comments. TypeWriter starts by adding the top-ranked predictions as type annotations, which introduces a type error about an incorrect return type of find_match, though. Based on this feedback, the search tries to change the return type of find_match to the second-best suggestion, Optional[set]. Unfortunately, this combination of added types leads to another type error because the return type is inconsistent with the argument key being of type int. The search again refines the type annotations by trying to use the second-best suggestion, str, for the argument key. Because the resulting set of type annotations is type-correct according to the type checker, TypeWriter adds these types to the code.

We implement TypeWriter for Python and apply it on two large code bases: a multi-million line code base at Facebook that powers applications used by billions of people, and a corpus of popular open-source projects. We show that the neural model predicts individual types with a precision of 64% (85%, 91%) and a recall of 52% (64%, 68%) within the top-1 (top-3, top-5) predictions, which outperforms a recent, closely related approach [21] by 10% and 6% respectively. Based on this model, the feedback-directed search finds a type-correct subset of type annotations that can produce complete and type-correct annotations for 42% to 64% of all files. Comparing TypeWriter with a traditional, static type inference shows that both techniques complement each other and that TypeWriter predicts many more types than traditional type inference. In summary, this paper makes the following contributions:

- A combination of probabilistic type prediction and search-based validation of predicted types. The feedback-directed search for type-correct types can be used with any probabilistic type predictor and any gradual type checker.
- A novel neural type prediction model that exploits both code context and natural language information.
- Empirical evidence that the approach is effective for type-annotating large-scale code bases with minimal human effort. The initial experience from using TypeWriter at Facebook on a code base that powers tools used by billions of people has been positive.

2 APPROACH

Figure 2 gives a high-level overview of the TypeWriter approach. The input to TypeWriter is a corpus of code where some, but not all types are annotated. The approach consists of three main parts. First, a lightweight static analysis extracts several kinds of information from the given code (Section 2.1). The extracted information includes programming structure information, such as usages of a function’s arguments, and natural language information, such as identifier names and comments. Next, a neural type predictor learns from the already annotated types and their associated information how to predict missing types (Section 2.2). Once trained, this model can predict likely types for currently unannotated parts of the code. Finally, a feedback-directed search uses the trained model to find a type assignment that is consistent and type-correct according to a static, gradual type checker (Section 2.3). The overall output of TypeWriter is code with additional type annotations.

2.1 Static Extraction of Types and Context Information

The first part of TypeWriter is an AST-based static analysis that extracts types and context information useful to predict types. The analysis is designed to be lightweight and easy to apply to other programming languages. We currently focus on function-level types, i.e., argument types and return types. These types are particularly important for two reasons: (i) Given function-level types, gradual type checkers can type-check the function bodies by inferring the types of (some) local variables. (ii) Function-level types serve as interface documentation.

For each type, the static analysis gathers four kinds of context information, which the following describes and illustrates with the example in Figure 3. Each of the four kinds of information may provide hints about an argument type or return type, and our model learns to combine these hints into a prediction of the most likely types.

Identifier names associated with the to-be-typed program element. As shown by prior work [21, 27], natural language information embedded in source code can provide valuable hints about program properties. For example, the argument names name and do_propagate in Figure 3 suggest that the arguments may be a string and a boolean, respectively. To enable TypeWriter to benefit from such hints, the static analysis extracts the identifier name of each with function and each function argument.
TypeWriter: Neural Type Prediction with Search-based Validation

Figure 2: Overview of TypeWriter.

Figure 3: Example of a to-be-typed Python function.

from html import HtmlElement

def update_name(name, do_propagate, element):
    ""
    Update the name and (optionally)
    propagate to dependents.
    ""
    first_name = name.split(" ")[0]
    element.first = first_name
    if do_propagate:
        d.notify(NAME, first_name)

Code occurrences of the to-be-typed program element. In addition to the above natural language information, TypeWriter exploits programming language type hints. One of them is the way a to-be-typed program element is used: As a hint about argument types, the analysis considers all usages of an argument within the function body. Another kind of information is code that defines the to-be-typed program element: As a hint about return types, the analysis considers all return statements in a function. For each of these code locations, the analysis extracts the corresponding sequence of code tokens \(o_1, \ldots, o_k\). Specifically, the analysis extracts a window of tokens around each occurrence of an argument (default size of window: 7) and all tokens of a return statement. For example, the analysis extracts the token sequence \\n\text{\texttt{\textbackslash n, first_name, =, name, , , , \textbackslash \textbackslash n, split, ( around the usage of name at line 6.}

As an alternative to extracting token sequences, TypeWriter could perform a more sophisticated static analysis, e.g., by tracking data flows starting at arguments or ending in return values. We instead focus on token sequences because it provides a sufficiently strong signal, scales well to large code bases, and could be easily ported to other programming languages.

Function-level comments. Similar to identifier names, comments are another informal source of hints about types. For the example in Figure 3, a developer might infer from the function-level comment that the function has some side effects but probably does not return any value. To allow the approach to benefit from such hints, the static analysis extracts all function-level comments, i.e., docstrings in Python. For a given function, the approach uses this comment both for predicting the argument types and the return type of the function.

Available types. To annotate a type beyond the built-in types of Python, the type needs to be either imported or locally defined. Because types used in an annotation are likely to be already imported or locally defined, the analysis extracts all types available in a file. To this end, the analysis parses all import statements and all class definitions in a file. For the example in Figure 3, the analysis will extract HtmlElement as an available type, which hints at the argument element being of this type.

Based on these four kinds of type hints, the analysis extracts the following information for argument types and return types, respectively:

Definition 2.1 (Argument type information). For a function argument \(a\), the statically extracted information is a tuple

\[(n_{fct}, n_{arg}, N_{arg}, c, U, t)\]

where \(n_{fct}\) is the function name, \(n_{arg}\) is the name of the argument \(a\), \(N_{arg}\) is the sequence of names of other arguments (if any), \(c\) is the comment associated with the function, \(U\) is a set of usage sequences, each of which is a sequence \(o_1, \ldots, o_k\) of tokens, and \(t\) is the type of the argument.

Definition 2.2 (Return type information). For the return type of a function \(f\), the statically extracted information is a tuple

\[(n_{fct}, N_{arg}, c, R, t)\]

where \(n_{fct}\) is the function name, \(N_{arg}\) is the sequence of argument names, \(c\) is the comment associated with the function, \(R\) is a set of return statements, each of which is a sequence \(o_1, \ldots, o_k\) of tokens, and \(t\) is the return type of \(f\).

If any of the above information is missing, the corresponding elements of the tuple is filled with a placeholder. In particular, the static analysis extracts the above also for unannotated types, to enable TypeWriter to predict types based on the context.

2.2 Neural Type Prediction Model

Given the extracted types and context information, the next part of TypeWriter is a neural model that predicts the former from the
latter. We formulate the type prediction problem as a classification problem, where the model predicts a probability distribution over a fixed set of types. The neural type prediction model, summarized in the middle part of Figure 2, combines the four kinds of information described in Section 2.1 into a single type prediction.

To represent identifier names, source code tokens, and words in a way suitable for learning, TypeWriter maps each into a real-valued vector using a Word2Vec [23] embedding. We train two embeddings, a code embedding $E_{code}$ for code tokens and identifier names, and a word embedding $E_{word}$ for words in comments. $E_{code}$ is trained on sequences of tokens extracted from source code files, while $E_{word}$ is trained on sequences of words extracted from comments. To mitigate the problem of large vocabularies in source code [5], TypeWriter preprocesses each identifier using a helper function $\text{norm}()$, which tokenizes, lemmatizes, and lowercases each identifier.

### 2.2.1 Learning from Identifiers

This neural submodel learns from the identifier names of functions and function arguments. The model combines all identifiers associated with a type into a sequence. Given argument type information $(n_{fct}, n_{arg}, N_{args}, c, U, t)$, the sequence is

$$\text{norm}(n_{arg}) \circ s \circ \text{norm}(n_{fct}) \circ \text{norm}(N_{args})$$

where $\circ$ flattens and concatenates sequences, and $s$ is a separator.

Given return type information $(n_{fct}, N_{args}, c, R, t)$, the sequence is

$$\text{norm}(n_{fct}) \circ s \circ \text{norm}(N_{args})$$

For example, the sequence for the return type of the function in Figure 3 is “update name s name do propagate element”.

TypeWriter learns from these sequences of words by summarizing them into a single vector using a bi-directional, recurrent neural network (RNN) based on LSTM cells. To ease parallelization, we pad sequences that are too short and truncate sequences that are too long (default length: 10). The final hidden states of the RNN serve as a condensed vector representation, $v_{ids}$, of all identifier-related hints.

### 2.2.2 Learning from Token Sequences

This neural submodel learns from source code information associated with a type. Similar to the submodel for identifiers, this submodel composes all relevant tokens into a sequence and summarizes them into a single vector $v_{code}$ using an RNN. For arguments and return types, the sequence consists of the tokens involved in the usages $U$ (Definition 2.1) and the return statements $R$ (Definition 2.2), respectively. Before feeding these sequences into an RNN, we bound the length of each token sequence (default: $k = 7$) and of the number of token sequences (default: 3).

### 2.2.3 Learning from Comments

This neural submodel learns type hints from comments associated with a function. To this end, TypeWriter splits a given comment into a sequence of words, bound the length of the sequence to a fixed value (default: 20), and summarizes the sequence via another RNN. The result is a fixed-length vector $v_{comments}$.

### 2.2.4 Learning from Available Types

The fourth kind of information that TypeWriter learns from is the set of types available in the current source code file. The approach assumes a fixed-size vocabulary $T$ of types (default size: 1000). This vocabulary covers the vast majority of all type occurrences because most type annotations either use one of the built-in primitive types, e.g., `str` or `bool`, common non-primitive types, e.g., `List` or `Dict`, or their combinations, e.g., `List[int]` or `Dict[str, bool]`. Any types beyond the type vocabulary are represented as a special “unknown” type.

To represent which types are available, we use a binary vector of size $T$, called the type mask. Each element in this vector represents one type, and an element is set to one if and only if its type is present. The resulting vector $v_{availTypes}$ of available types is passed as-is into the final part of the neural model.

#### 2.2.5 Predicting the Most Likely Type

The final submodel concatenates the four vectors $v_{ids}, v_{code}, v_{comments}$, and $v_{availTypes}$ into a single vector and passes it through a fully connected layer that predicts the most likely type. The output vector has size $|T|$ and represents a probability distribution over the set of types. For example, suppose the type vocabulary had only four types int, bool, None, and List, and that the output vector is $[0.1, 0.6, 0.2, 0.1]$. In this case, the model would predict that bool is the most likely type, following by None.

There are two ways to handle uncertainty and limited knowledge in the model. First, we interpret the predicted probability of a type as a confidence measure and only suggest types to a user that are predicted with a confidence above some configurable threshold. Second, we encode types not included in the fixed-size type vocabulary as a special “unknown” type. The model learns to predict “unknown” whenever none of the types in the vocabulary fit the given context information. During prediction, TypeWriter never suggests the “unknown” type to the user, but instead does not make any suggestion in case the model predicts “unknown”.

#### 2.2.6 Training

To train the type prediction model, TypeWriter relies on already type-annotated code. Given such code, the approach creates one pair of context information and type for each argument type and for each return type. These pairs then serve as training data to tune the parameters of the different neural submodels. We use stochastic gradient descent, the Adam optimizer, and cross-entropy as the loss function. The entire neural model is learned by using an RNN. For arguments and return types, the sequence consists of the tokens involved in the usages $U$ (Definition 2.1) and the return statements $R$ (Definition 2.2), respectively. Before feeding these sequences into an RNN, we bound the length of each token sequence (default: $k = 7$) and of the number of token sequences (default: 3).

### 2.3 Feedback-guided Search for Consistent Types

The neural type prediction model provides a ranked list of $k$ predictions for each missing type annotation. Given a set of locations for which a type annotation is missing, called type slots, and a list of probabilistic predictions for each slot, the question is which of the suggested types to assign to the slots. A naïve approach might fill each slot with the top-ranked type. However, because the neural model may mis-predict some types, this approach may yield type
Algorithm 1 Find a correct type assignment for a file \( f \)

1. function ASSIGN_TYPES\(^1\) (\( f \))
2. \( T \leftarrow \) all type slots in \( f \)
3. \( \mathcal{P}_{i}^{1..k} \leftarrow \{ \text{predictions}(t, k) \mid t \in T \} \) \( \triangleright \) Top \( k \) predictions
4. \( a \leftarrow \{ \mathcal{P}_{i}^{1} \mid t \in T \} \) \( \triangleright \) Initial type assignment
5. \( a.\text{score} \leftarrow \text{typecheck}(a, f) \) \( \triangleright \) Feedback function
6. work_set \( \leftarrow \text{new\_states}(a, \mathcal{P}, T) \)
7. done \( \leftarrow \{ a \}
8. \text{while} \ \min \{ (x.\text{score} \mid x \in \text{done}) \} > 0 \land \text{work\_set} \neq \emptyset \) do
9. \( a \leftarrow \text{pick(work\_set)} \) \( \triangleright \) Biased random selection
10. \( a.\text{score} \leftarrow \text{typecheck}(a, f) \)
11. \( \text{if greedy} \land a.\text{score} < a.\text{parent}.\text{score} \) then
12. \( \text{work\_set} \leftarrow \text{new\_states}(a, \mathcal{P}, T) \)
13. \( \text{else if non-greedy} \) then
14. \( \text{work\_set} \leftarrow \text{work\_set} \cup (\text{new\_states}(a, \mathcal{P}, T) \setminus \text{done}) \)
15. end if
16. done \( \leftarrow \) done \( \cup \{ a \} \)
17. end while
18. return \( \text{argmin} \{ (x.\text{score} \mid x \in \text{done}) \} \)
19. end function
20. function NEW\_STATES\(^1\) (\( a, \mathcal{P}, T \))
21. children \( \leftarrow \{ \} \)
22. for all \( t \in T \) do
23. \( \text{for all} \ \mathcal{P}_{j}^{t} \) where \( j > \text{rank} \) of current \( a[t] \) do
24. \( a_{\text{child}} \leftarrow \text{modify} \ a \ \text{to use} \ \mathcal{P}_{j}^{t} \ \text{at} \ t \)
25. children \( \leftarrow \{ a_{\text{child}} \} \)
26. end for
27. \( a_{\text{child}} \leftarrow \text{modify} \ a \ \text{to not use any type at} \ t \)
28. children \( \leftarrow \{ a_{\text{child}} \} \)
29. end for
30. return children
31. end function

assignments where the added annotations are not consistent with each other or with the remaining program.

To avoid introducing type errors, TypeWriter leverages an existing gradual type checker as a filter to validate candidate type assignments. Such type checkers exist for all popular dynamically typed languages that support optional type annotations, e.g., pyre and mypy for Python, and flow for JavaScript. TypeWriter exploits feedback from the type checker to guide a search for consistent types, as presented in Algorithm 1 and explained in the sections below.

2.3.1 Search Space. Given a set \( T \) of type slots and \( k \) predicted types for each slot, we formulate the problem of finding a consistent type assignment as a combinatorial search problem. The search space consists of the set \( \mathcal{P} \) of possible type assignments. For \( |T| \) type slots and \( k \) possible types for each slot, there are \((k + 1)^{|T|}\) type assignments (the +1 is for not assigning any of the predicted types).

2.3.2 Feedback Function. Exhaustively exploring \( \mathcal{P} \) is practically infeasible for files with many missing types, because invoking the gradual type checker is relatively expensive (typically, in the order of several seconds per file). Instead, TypeWriter uses a feedback function (typecheck) to efficiently steer toward the most promising type assignments.

The feedback function is based on two values, both of which the search wants to minimize:

- \( n_{\text{missing}} \): The number of missing types.
- \( n_{\text{errors}} \): The number of type errors.

TypeWriter combines these into a weighted sum \( \text{score} = \nu \cdot n_{\text{missing}} + \omega \cdot n_{\text{errors}} \). By default, we set \( \nu \) to 1 and \( \omega \) to the number of initially missing types plus one, which is motivated by the fact that adding an incorrect type often leads to an additional error. By giving type errors a high enough weight, we ensure that the search never returns a type assignment that adds type errors to the code.

2.3.3 Exploring the Search Space. TypeWriter explores the space of type assignments through an optimistic search strategy (Algorithm 1). It assumes that most predictions are correct, and then refines type annotations to minimize the feedback function. Each exploration step explores a state \( a \), which consists of a type assignment, the score computed by the feedback function, and a link to the parent state. The initial state is generated by retrieving the top-1 predictions from \( \mathcal{P} \) for each type slot \( t \) and invoking the feedback function (lines 4 and 5). The next states to be explored are added to a work set, while the explored states are kept in the "done" set. The algorithm loops over items in the work set until either the feedback score has been minimized or the search explored all potential type assignments (line 8). The assignment with the minimal score is returned as a result (line 18).

To retrieve the next type assignments to possibly explore from the current state, TypeWriter invokes the new\_states helper function. It adds all type assignments that can be obtained from the current state by modifying exactly one type slot, either by using a lower-ranked type suggestion or by not adding any type for this slot (lines 22 to 29).

The main loop of the algorithm (lines 8 to 17) picks a next state to evaluate from the working set (line 9), queries the feedback function (line 10) and updates the done set with the explored state (line 16). The pick function is a biased random selection that prefers states based on two criteria. First, it prefers states that add more type annotations over states that add fewer annotations. Second, it prefers states that modify a type annotation at a line close to a line with a type error. Intuitively, such states are more likely to fix the cause of a type error than a randomly selected state.

The working set is updated with all new states that have not been currently explored.

TypeWriter implements two variants of the search, a greedy and a non-greedy one. The greedy strategy aggressively explores children of type assignment that decrease the feedback score and prunes children of states that increase it (line 12). The non-greedy performs no pruning, i.e., it can explore a larger part of the search space at the expense of time (line 14).

As an optimization of Algorithm 1, TypeWriter invokes the assign\_types function twice. The first call considers only type slots for return types, whereas the second call considers all type slots for argument types. The reason for this two-phase approach is that

\(^{1}\)The reason for relying on line numbers as the interface between the type checker and TypeWriter is to enable plugging any type checker into our search.
many gradual type checkers, including pyre, the one used in our evaluation, type-check a function only if its return type is annotated. If TypeWriter would add argument type annotations before adding return type annotations, the feedback function might not include all type errors triggered by an incorrectly added argument annotation.

3 IMPLEMENTATION
The implementation of TypeWriter builds upon a variety of tools in the Python ecosystem. For the static analysis phase, we apply a data extraction pipeline consisting of Python’s own ast library to parse the code into an AST format, and NLTK and its WordNetLemmatizer module to perform standard NLP tasks (lemmatization, stop word removal). The pipeline is parallelized so that it handles multiple files concurrently. The neural network model is implemented in PyTorch. For obtaining embeddings for words and tokens, we pre-train a Word2Vec model using the gensim library. The search phase of TypeWriter builds upon the LibCST\textsuperscript{2} to add types to existing Python files. We use pyre for static type checking. Our LSTM models all use 200-dimensional hidden layers, and we train for 10 epochs with a learning rate of 0.005 using the Adam Optimizer.

4 EVALUATION
We structure our evaluation along four research questions.
RQ 1: How effective is TypeWriter’s model at predicting argument and return types, and how does it compare to existing work?
RQ 2: How much do the different kinds of context information contribute to the model’s prediction abilities?
RQ 3: How effective is TypeWriter’s search?
RQ 4: How does TypeWriter compare to traditional static type inference?

4.1 Datasets
TypeWriter is developed and evaluated within Facebook. As the internal code base is not publicly available and to ensure that the presented results are replicable, we use two datasets:

Internal code base We collect Python from a large internal code repository.

OSS corpus We search GitHub for all projects tagged as python\textsuperscript{3}. We also search Libraries.io for all Python projects that include mypy as a dependency. We then remove all projects that have less than 50 stars on GitHub, to ensure that the included projects are of substantial public interest. To ease future work to compare with TypeWriter, all results for the OSS corpus are available for download.\textsuperscript{3}

The resulting dataset statistics can be found in Table 1. The internal dataset is much larger in size, but both datasets are comparable in terms of the percentage of annotated code. By restricting the type vocabulary to a fixed size, we exclude around 10% of all type occurrences for both datasets. This percentage is similar for both datasets, despite their different sizes, because types follow a long-tail distribution, i.e., relatively few types account for the majority of all type occurrences. We ignore some types because they are trivial.

Table 1: Internal and open-source datasets.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Internal</th>
<th>OSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td>1</td>
<td>1,137</td>
</tr>
<tr>
<td>Files</td>
<td>*</td>
<td>11,993</td>
</tr>
<tr>
<td>Lines of code</td>
<td>*</td>
<td>2.7M</td>
</tr>
<tr>
<td>Functions</td>
<td>*</td>
<td>146,106</td>
</tr>
<tr>
<td>...with return type annotation</td>
<td>66%</td>
<td>80,341 (35%)</td>
</tr>
<tr>
<td>...with comment</td>
<td>21.8%</td>
<td>53,500 (39.3%)</td>
</tr>
<tr>
<td>...with both</td>
<td>16%</td>
<td>32,409 (22.2%)</td>
</tr>
<tr>
<td>...ignored because trivial</td>
<td>7.4%</td>
<td>12,436 (8.5%)</td>
</tr>
<tr>
<td>Arguments</td>
<td>*</td>
<td>274,425</td>
</tr>
<tr>
<td>...with type annotation</td>
<td>50%</td>
<td>112,409 (41%)</td>
</tr>
<tr>
<td>...ignored because trivial</td>
<td>33%</td>
<td>96,036 (35%)</td>
</tr>
<tr>
<td>Types - Return</td>
<td>*</td>
<td>7,383</td>
</tr>
<tr>
<td>...occurrences ignored (out of vocab.)</td>
<td>20.2%</td>
<td>11.3%</td>
</tr>
<tr>
<td>Types - Argument</td>
<td>*</td>
<td>8,215</td>
</tr>
<tr>
<td>...occurrences ignored (out of vocab.)</td>
<td>21.3%</td>
<td>13.7%</td>
</tr>
<tr>
<td>Training time (min:sec)</td>
<td>several minutes</td>
<td>1:45</td>
</tr>
<tr>
<td>...parsing</td>
<td>several minutes</td>
<td>2:29</td>
</tr>
<tr>
<td>...training embeddings</td>
<td>several minutes</td>
<td>2:29</td>
</tr>
<tr>
<td>...training neural model</td>
<td>several minutes</td>
<td>2:20</td>
</tr>
</tbody>
</table>

\* = not available for disclosure

4.2 Examples
Figure 4 shows examples of successful and unsuccessful type predictions in the OSS dataset. Example 1 presents a case where TypeWriter correctly predicts a type annotation. Here, the code context is expected. We hypothesize that this case of mis-prediction is due to the fact that TypeWriter tries to associate associations between natural language and types, or in this case, the word “path” and the type Path.

4.3 RQ 1: Effectiveness of the Neural Model

Prediction tasks. To evaluate the neural type prediction, we define two prediction tasks: (i) ReturnPrediction, where the model predicts the return types of functions, and (ii) ArgumentPrediction, where the model predicts the types of function arguments, and

Metrics. We evaluate the effectiveness of TypeWriter’s neural type predictor by splitting the already annotated types in a given dataset into training (80%) and validation (20%) data. The split is by file, to avoid mixing up types within a single file. Once trained on the training data, we compare the model’s predictions against the validation data, using the already annotated types as the ground truth to predict, such as the return type of \_\_\_str\_\_, which always is str, or the type of the \_\_\_self \_\_\_ argument of a method, which always is the surrounding class. TypeWriter could easily predict many of these trivial types, but a simple syntactic analysis would also be sufficient. We ignore trivial types for the evaluation to avoid skewing the results in favor of TypeWriter.
Table 2: Effectiveness of neural type prediction.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>Task</th>
<th>Model</th>
<th>Precision</th>
<th>Recall</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Top-1</td>
<td>Top-3</td>
<td>Top-5</td>
</tr>
<tr>
<td>Internal</td>
<td>ReturnPrediction</td>
<td>TypeWriter</td>
<td>0.73</td>
<td>0.88</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NLZType</td>
<td>0.60</td>
<td>0.82</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DeepTyper</td>
<td>0.70</td>
<td>0.87</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Naive baseline</td>
<td>0.15</td>
<td>0.22</td>
<td>0.25</td>
</tr>
<tr>
<td>ArgumentPrediction</td>
<td>TypeWriter</td>
<td>0.64</td>
<td>0.86</td>
<td>0.92</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NLZType</td>
<td>0.53</td>
<td>0.80</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DeepTyper</td>
<td>0.54</td>
<td>0.80</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Naive baseline</td>
<td>0.08</td>
<td>0.15</td>
<td>0.18</td>
</tr>
</tbody>
</table>

Baseline models. We compare TypeWriter’s top-k predictions against three baseline models. The naïve baseline model considers the ten most frequent types in the dataset and samples its prediction from the distribution of these ten types, independently of the given context. For example, it predicts None as a return type more often than List[str] because None is used more often as a return type than List[str]. The DeepTyper baseline is a Python re-implementation of the DeepTyper [15] model. DeepTyper learns to translate a sequence of source code tokens to a sequence of types (and zeros for tokens without a type). To make it directly compatible with TypeWriter, we do not consider predictions for variable annotations in function bodies, even though we do perform basic name-based type propagation in case an annotated argument is used in a function body. Finally, the NLZType baseline is a re-implementation of the NLZType model [21] for Python, which also learns from natural language information associated with a type, but does not consider code context or available types.

Results. Table 2 presents the results for RQ 1. Our neural model achieves moderate to high precision scores, e.g., 73% in the top-1 and 92% in the top-5 for on the internal dataset for the ReturnPrediction task. The recall results are good but less high than precision, indicating that TypeWriter is fairly confident when it makes a prediction, but abstains from doing so when it is not. All models have slightly worse performance on the OSS dataset, which we attribute to the smaller size of that dataset. The fact that the top-3 and top-5 scores are significantly higher than top-1 in all cases motivates our work on combinatorial search (Section 4.5).

Compared to the baselines, TypeWriter outperforms both the state of the art and the naive baseline across all metrics for both datasets and all three prediction tasks. The differences between

Figure 4: Examples of successful and unsuccessful type predictions (GitHub: PrefectHQ/ct, awslabs/sockeye).

truth. We compute precision, recall, and F1-score, weighted by the number of type occurrences in the dataset. Similarly to previous work [21], if the prediction model cannot predict a type for a type slot (i.e., returns "unknown"), we remove this type slot from the calculation of precision. Specifically, we calculate precision as $\text{prec} = \frac{n_{corr}}{n_{all}}$, where $n_{corr}$ is the number of correct predictions and $n_{all}$ is the number of type slots for which the model does not return "unknown". We calculate recall as $\text{rec} = \frac{n_{corr}}{|D|}$, where $|D|$ is total number of type slots in the examined dataset. We report the top-k scores, for $k \in \{1, 3, 5\}$.
TypeWriter and NL2Type are higher in the case of the ReturnPrediction task. The context information, as obtained by analyzing token sequences, is helping the TypeWriter prediction model more in the ReturnPrediction task. Compared to DeepTyper, both NL2Type and TypeWriter are better, the latter by a significant margin, in top-1 but not in top-3 or top-5 predictions. Given that all models learn primarily from identifier names, the relatively close upper bound performance seems to indicate that performance improvements may only be achieved by introducing different (e.g., structural) information to the model.

4.4 RQ 2: Comparison with Simpler Variants of the Neural Model

The main novelty of TypeWriter’s prediction component is the inclusion of code context information and a local type mask in the prediction model. To explore the influence of the different type hints considered by TypeWriter, we perform an ablation study. Specifically, we turn off parts of the model, both in training and in testing, and then measure top-1 precision and recall at various prediction threshold levels. We start with the full model (typewriter) and then we remove, in order, the type mask, the token sequences, the method and argument names and the documentation. As a baseline, we also include nl2type, a configuration functionally equivalent with NL2Type [21], which corresponds to TypeWriter without token sequences and without a type mask. The results of the ablation study can be seen in Figure 5.

Overall, the combined information of natural language, token sequences, and type masks helps TypeWriter to perform better than previous models, such as NL2Type. The main contributor to this improvement is the token sequences component. Moreover, the results seem to reinforce the main thesis of NL2Type, i.e., that natural language information and types are strongly related: If we remove the argument and function naming information from TypeWriter, its performance drops significantly.

Contrary to our initial expectations, the type mask component is not contributing significantly in the ReturnPrediction task, while only slightly improving the ArgumentPrediction results. We attribute this to the current implementation of the type mask data extraction process: the extractor currently neither performs an in-depth dependency resolution to retrieve the full set of types available in the processed file’s name space, nor does it track type renamings (e.g., import pandas as pd). The low predictive capability of comments can be explained by the fact that only a small number of the methods in both datasets have documentation at the method level.

4.5 RQ 3: Effectiveness of Search

To evaluate the search, we collect a ground truth of 50 fully annotated files that are randomly sampled from the industrial codebase at Facebook. We ensure that they type-check correctly. The files we select originate from different products and vary in size and complexity, the files average 7 (median: 6, 95%: 13) annotations. The total number of annotations is 346. For each file in the ground truth, we strip its existing annotations and then apply TypeWriter to predict and evaluate the missing types. We configure both the greedy and the non-greedy search strategies to stop when the number of states explored is seven times the number of type slots. This threshold empirically strikes a reasonable balance between investigating time and detecting correct types. We use the same prediction model trained on the Facebook dataset as in Section 4.3.

Table 3 shows the results on two levels: individual type annotations and files. On the annotation-level, column type-correct shows how many type slots the type assignment returned by the search fills (recall that the search ensures each added type to be type-correct). Column ground truth match shows how many of all added annotations match the original, developer-produced type annotations. On the file-level, a complete and type-correct solution is a file that TypeWriter fully annotates without type errors. This metric does not include files where TypeWriter discovers a type-correct, but partially annotated solution. The ground truth match is the subset of the complete and type-correct solutions, where the solution is identical to the ground truth for all types in the file. It is possible to find a type-correct annotation that does not match the ground truth. For example, TypeWriter may correctly annotate the return type of a function as a List, but a human expert might choose a more precise type List[str]: both are type-correct, but the human annotation provides more guarantees.

Both search strategies successfully annotate a significant fraction of all types. On the annotation-level, they add between 40% and 63% of all types in a type-correct way, out of which 28% to 47% match the ground truth, depending on the search strategy. On the file-level, TypeWriter completely annotates 14% to 44% of all files, and 10% to 22% of all files perfectly match the developer annotations. Comparing the two search strategies, we find that, at the annotation-level, greedy search discovers more type-correct annotations with
We compare TypeWriter with a state-of-the-art, static type inference tool pyre infer. The type inference is part of the pyre type checker and is representative of conservative static analysis-based type inference that adds only types guaranteed to be type-correct. We run pyre infer on the same set of randomly chosen, fully annotated files as in Section 4.5 and then compare the added annotations with TypeWriter’s top-5 search results.

Tables 3 (bottom) and 4 show the results. In a head to head comparison, TypeWriter is able to provide type-correct predictions for about seven times the number of files that pyre infer can. It also discovers significantly more types, adding a total of 188 types, whereas pyre infer adds only 100. Additionally, of the 82 type slots for which both tools suggest a type, the suggestions are the same in 63 cases. Effectively, the types that TypeWriter suggests are a superset of those inferred by pyre infer, as pyre infer does not uniquely find many types.

To further illustrate the differences, we plot the distribution of the top-10 correctly predicted types in Figure 6. We see that pyre infer can infer more precise types, but the majority of its inferences are on methods with no return types. Moreover, some of the inferred types are of dubious usefulness (e.g., Optional[Optional[Context]]) indicating the difficulty of applying static type inference on dynamically-typed languages and reinforcing our thesis on the value of prediction-based type inference.

5 DISCUSSION

Effectiveness of neural type prediction. TypeWriter implements the first neural type prediction model for Python. As all existing
Type prediction models [15, 21, 28] target JavaScript code, it is difficult to draw conclusions as to whether the TypeWriter architecture is the best for the task. Two facts seem to suggest so: i) TypeWriter is better by a comfortable margin than a re-implementation of the two best-in-class JavaScript models, and ii) TypeWriter’s performance is stable across two very different datasets.

**Type-correctness vs. soundness.** Due to the way current Python type checkers work, the types that TypeWriter produces are guaranteed to be type-correct within the context of a given module. Type correctness is different from type soundness, as the later can only be verified using human intuition. This means that if a module is used within another context, the type checker might invalidate an initially correct prediction. In turn, this makes TypeWriter a soundy [20], rather than a sound approach.

**Limited type vocabulary.** TypeWriter only predicts types that are part of its type vocabulary. When the vocabulary size is configured at 1000 types, it can account for 90% of the available types in both our datasets. However, as software evolves, developers create new types or change the names of existing ones. This may lead to situations where the model would predict a wrong type because its name changed or because it simply does not know that the type exists. The out-of-vocabulary problem is well known in software engineering research [16]. Recent work for by Karampatsis et al. [18] uses sub-word information to account for neologisms with very good results. We believe that TypeWriter would benefit significantly from such an approach for embedding identifier names, as it would enable it to learn semantically similar name variants (e.g., AbstractClass and Class or List and List[str]).

**Further improvements.** TypeWriter is a prototype stemming from a general effort within Facebook to make their Python code base more robust. TypeWriter can be improved in several dimensions, some of which are presented below:

**Better data:** The ablation study results suggest that type masks and documentation components of the TypeWriter model are only marginally contributing to its prediction capabilities. This goes against both intuition and published work: in [21], the authors show that code documentation is an important signal. We could, however, exploit the fact that highly used libraries, such as Flask or the Python standard library itself feature both type annotations (in the typeshed repository) and excellent documentation. Moreover, we can obtain better type masks using lightweight dependency analysis, such as importlib, to identify all types that are in context.

**Faster search feedback:** TypeWriter’s execution speed is currently constrained by the type checker used to obtain feedback. One natural way to improve this would be to integrate the TypeWriter type predictor into a static type inference loop: when the type inference cannot predict a type for a location, it can ask the neural model for a suggestion. While the theoretical cost of searching for types is similar, in practice the type inference will be able to quickly examine suggested types given that all required data is loaded in memory.

**Reinforced learning:** As with most neural models, TypeWriter can benefit from more data. One idea worth exploring is to apply TypeWriter in batches, consisting of application of an initial set of neural predictions, reviewing proposed types through the normal code review process at Facebook and then retrain the model on the new data. At the scale of the Facebook code base, we expect that the feedback obtained (accepted, modified, and rejected suggestions) could be used to improve the learning process.

6 RELATED WORK

**Type inference for dynamic languages.** Static type inference [4, 8, 14, 17] computes types using, e.g., abstract interpretation or type constraint propagation. These approaches are sound by design, but due to the dynamic nature of some languages, they often infer only simple or very generic types [8, 17]. They also require a significant amount of context, usually a full program and its dependencies. Dynamic type inference [3, 29] tracks data flows between functions, e.g., while executing a program’s test suite. These approaches capture precise types, but they are constrained by coverage. TypeWriter differs from those approaches in two key aspects: i) it only requires limited context information, i.e., a single a source code file, ii) it does not require the program to be executed and hence can predict types in the absence of a test suite or other input data.

**Probabilistic type inference.** The difficulty of accurately inferring types for dynamic programming languages has led to research on probabilistic type inference. JSNice [28] models source code as a dependency network of known (e.g., constants, API methods) and unknown facts (e.g., types); it then mines information from large code bases to quantify the probability of two items being linked together. Xu et al. [37] predict variable types based on a probabilistic combination of multiple uncertain type hints, e.g., data flows and attribute accesses. They also consider natural language information, but based on lexical similarities of names, and focus on variable types, whereas TypeWriter focuses on function types. DeepTyper [15] uses a sequence-to-sequence neural model to predict types based on a bi-lingual corpus of TypeScript and JavaScript code. NLZType [21] uses natural language information. Our evaluation directly compares with Python re-implementations of both DeepTyper and NLZType. Besides advances in the probabilistic type prediction model itself, the more important contribution of our work is to address the imprecision and combinatorial explosion problems of probabilistic type inference. In principle, any of the above techniques can be combined with TypeWriter’s search-based validations to obtain type-correct types in reasonable time.

**Type checking and inference for Python.** The Python community introduced a type annotation syntax along with a type checker (mypy) as part of Python 3.5 version in 2015 [32]. The combination of the two enables gradual typing of existing code, where the type checker checks only the annotated parts of the code. Similar approaches have also been explored by the research community [34]. Since 2015, type annotations have seen adoption in several large-scale Python code bases, with products such as Dropbox6 and Instagram,7 reportedly having annotated large parts of their multi-million line code bases. TypeWriter helps reduce the manual effort required for such a step.

---

6[GitHub: python/typeshed](https://github.com/python/typeshed)
7[GitHub: google/importlab](https://github.com/google/importlab)
**TypeWriter: Neural Type Prediction with Search-based Validation**

Machine learning on code. Our neural type prediction model is motivated by a stream of work on machine learning-based program analyses [2]. Beyond type prediction, others have proposed learning-based techniques to find programming errors [25, 27], predict variable and method names [1, 28, 33], suggest how to improve names [19], search code [10, 30], detect clones [36, 40], classify code [24, 39], predict code edits [31, 38, 41], predict assertions [35], and automatically fix bugs [6, 11, 12]. TypeWriter contributes a novel model for predicting types and a search-based combination of predictive models with traditional type checking.

Search-based software engineering. Our search-based validation of types fits the search-based software engineering theme [13], which proposes to balance competing constraints in developer tools through metaheuristic search techniques. In our case, the search balances the need to validate an exponential number of combinations of type suggestions with the need to efficiently annotate types.

**7 CONCLUSIONS**

We present TypeWriter, a learning-based approach to the problem of inferring types for code written in Python. TypeWriter exploits the availability of partially annotated source code to learn a type prediction model and the availability of type checkers to refine and validate the predicted types. TypeWriter’s learned model can readily predict correct type annotations for half of the type slots on first try, whereas its search component can help prevent annotating code with wrong types. Combined, the neural prediction and the search-based refinement helps annotate large code bases with minimal human intervention, making TypeWriter the first practically applicable learning-based tool for type annotation.

We are currently in the process of making TypeWriter available to developers at Facebook. We have tested the automation of the tool in the code review domain. Developers at Facebook received type suggestions as comments on pull requests they had authored. They would also receive pull requests containing type suggestions for their project. The initial experience from applying the approach on a code base that powers tools used by billions of people has been positive: several thousands of suggested types have already been accepted with minimal changes.

**REFERENCES**


