DLTPy: DEEP LEARNING TYPE INFERENCE OF PYTHON FUNCTION SIGNATURES USING NATURAL LANGUAGE CONTEXT

Casper Boone
Delft University of Technology
c.c.boone@student.tudelft.nl

Niels de Bruin
Delft University of Technology
d.j.m.debruin@student.tudelft.nl

Arjan Langerak
Delft University of Technology
a.c.langerak@student.tudelft.nl

Fabian Stelmach
Delft University of Technology
f.p.stelmach@student.tudelft.nl

November, 2019

Abstract

Due to the rise of machine learning, Python is an increasingly popular programming language. Python, however, is dynamically typed. Dynamic typing has shown to have drawbacks when a project grows, while at the same time it improves developer productivity. To have the benefits of static typing, combined with high developer productivity, types need to be inferred. In this paper, we present DLTPy: a deep learning type inference solution for the prediction of types in function signatures based on the natural language context (identifier names, comments and return expressions) of a function. We found that DLTPy is effective and has a top-3 F1-score of 91.6%. This means that in most of the cases the correct type is within the top-3 predictions. We conclude that natural language contained in comments and return expressions are beneficial to predicting types more accurately. DLTPy does not significantly outperform or underperform the previous work NL2Type for Javascript, but does show that similar prediction is possible for Python.

Index terms— *deep learning, natural language, type inference, python*

1 Introduction

Programming languages with dynamic typing, such as Python or JavaScript, are increasingly popular. In fact, supported by the increasing use of machine learning, Python is currently the top programming language in the IEEE Spectrum rankings [1]. Dynamically typed languages do not require manual type annotations and only know the types of variables at run-time. They provide much flexibility and are therefore very suitable for beginners and for fast prototyping. There are, however, drawbacks when a project grows large enough that no single developer knows every single code element of the project. At that point, statically typed languages can check certain naming behaviors based on types automatically whereas dynamic typing requires manual intervention. While there is an ongoing debate on static vs. dynamic typing in the developer community [2], both excel in certain aspects [3]. There is scientific evidence that static typing provides certain benefits that are useful when software needs to be optimized for efficiency, modularity or safety [4]. The benefits include better auto-completion in integrated development environments (IDEs) [5], more efficient code generation [4], improved maintainability [6], better readability of undocumented source code [6], and preventing certain run-time crashes [5]. It has also been shown that statically typed languages are "less error-prone than functional dynamic languages" [2].

Weakly typed languages, such as JavaScript, PHP or Python do not provide these benefits. When static typing is needed, there are usually two solutions available. Either using optional type syntax within the language or to use a variant of the language, which is essentially a different language, that does have a type system in place. For JavaScript, there are two often used solutions: Flow [7], which uses type annotations within JavaScript, and TypeScript [8], a JavaScript variant. PHP offers support for type declarations since PHP 7.0 [1], and checks these types at run-time. A well-known

[1] https://www.php.net/manual/en/migration70.new-features.php

PHP variant that has strong typing is HackLang [9], by Facebook. Python has support for typing since Python 3.5 [2]. It does not do any run-time checking, and therefore these types do not provide any guarantees if no type checker is run. The type checker mypy [10] is the most used Python type checker.

Type inference for Python has been addressed from multiple angles [11, 12, 13, 14, 15]. However, these solutions require some manual annotations to provide accurate results. Having to provide manual annotations is one of the main arguments against static typing because this lowers developer productivity.

In an attempt to mitigate this need for manual annotations and support developers in typing their codebases, we present DLTPy: a deep learning type inference solution based on natural language for the prediction of Python function types. Our work focuses on answering the question of how effective this approach is.

DLTPy follows the ideas behind NL2Type [5], a similar learning-based approach for JavaScript function types. Our solution makes predictions based on comments, on the semantic elements of the function name and argument names, and on the semantic elements of identifiers in the return expressions. The latter is an extension of the ideas proposed in [5]. The idea to use natural language contained in the parameter names for type predictions in Python is not new, Zhaogui Xu et al. already used this idea to develop a probabilistic type inferencer [11]. Using the natural language of these different elements, we can train a classifier that predicts types. Similar to [5] we use a recurrent neural network (RNN) with a Long Short-Term Memory (LSTM) architecture [16].

Using 5,996 open source projects mined from GitHub and Libraries.io that are likely to have type annotations, we train the model to predict types of functions without annotations. This works because code has been shown to be repetitive and predictable [17]. We make the assumption that comments and identifiers convey the intent of a function [5].

We train and test multiple variants of DLTPy to evaluate the usefulness of certain input elements and the success of different deep learning models. We find that return expressions are improving the accuracy of the model, and that including comments has a positive influence on the results.

DLTPy predicts types with a top-3 precision of 91.4%, a top-3 recall of 91.9%, and a top-3 F1-score of 91.6%. DLTPy does not significantly outperform or underperform the previous work NL2Type [5].

This paper's contributions are three-fold:

1. A deep learning network type inference system for inferring types of Python functions
2. Evaluation of the usefulness of natural language encoded in return expressions for type predictions
3. Evaluation of different deep learning models that indicates their usefulness for this classification task

[2]https://www.python.org/dev/peps/pep-0484/

2 Method

DLTPy has two main phases: a training phase and a prediction phase. In this section, we first describe the steps involved in the training process, and then discuss how prediction works, given the trained model. The training process consists of multiple steps. First, we extract relevant training data from Python projects (section 2.1). Next, we preprocess the training data by for instance lemmatizing the textual parts of the data (section 2.2). The preprocessed training data is then filtered and only relevant functions are selected (section 2.3). Then, we generate input vectors using word embeddings and one-hot encoding (section 2.4). Finally, we train an RNN (section 2.5). After the training process has completed, the trained RNN can be used to make predictions for function types (section 2.6).

2.1 Collecting Data from ASTs

For each Python project in our data set, we want to export relevant parts of functions. Every Python file is parsed to an abstract syntax tree (AST). From this AST, we find the functions within or outside a class in the Python file. For each function, we extract the following elements:

- n_f: The name of the function
- d_f: The docstring of the function
- c_f: The comment of the function
- n_p: A list of the names of the function parameters
- t_p: A list of the types of the function parameters
- c_p: A list of the comments describing function parameters
- e_r: A list of the return expressions of the function
- t_r: The return type of the function
- c_r: The comment describing the return value

Together, these elements form the tuple $(n_f, d_f, c_f, n_p, t_p, c_p, e_r, t_r, c_r)$. Figure 1a shows a code sample, this sample is parsed and the information for the tuple is extracted as described in Figure 1b. This tuple is similar to the input data used in NL2Type [5], except for d_f and e_r.

Figure 1: Overview of the process of training from annotated source code.

d_f is the docstring of the Python function. This docstring often contains a few lines of text describing the working of the function, and sometimes also contains information about the parameters or the return value. In some cases, a more structured format is used, such as ReST, Google, or NumPy style. These formats describe parameters and the return value, separately from the function description. In these cases, we can extract this information for c_f, c_p, and c_r. We extract these comments only if the docstring is one of the structured formats mentioned before.

e_r is a list of return expressions of the function. After the preprocessing step (section 2.2), this contains a list of all the identifiers and keywords used in the return expressions. The intuition is that often variable names are returned and that these names may convey useful information.

2.2 Preprocessing

The information in the tuple is still raw natural language text. To capture only the relevant parts of the text, we first preprocess the elements in the tuple. The preprocessing pipeline consists of four steps and is based on the preprocessing stage in [5]:

1. **Remove punctuation, line breaks, and digits** We replace all non-alphabetical characters. Line breaks are also removed to create a single piece of text. We replace a full stop that is not at the end of a sentence with a space. We do this to make sure that, for instance, an object field or function access is not treated as a sentence separator (for example `object.property` becomes `object property`).

2. **Tokenize** We tokenize sentences using spaces as a separator. Before tokenization, the underscores in snake case and camel case identifiers are converted to a space-separated sequence of words.

3. **Lemmatize** We convert all inflected words to their lemma. For example, "removing" and "removed" become "remove".

4. **Remove stop words** We remove stopwords (such as "was", "be", "and", "while" and "the" [3]) from the sentences because these words are often less relevant and thus more importance can be given to non-stopwords. This step is not included in the pipeline for identifiers (function names, parameter names, and return expressions), considering that in the short sentences these identifiers form, stopwords are more relevant.

An example of a preprocessed tuple is shown in Figure 1c.

2.3 Function Selection

After collecting and preprocessing the function tuples, we select relevant functions. We filter the set of functions on a few criteria.

[3]See https://gist.github.com/sebleier/554280 for a full list of stopwords

Length	Features
1	Datapoint Type ([1 0 0 ... 0])
1	Separator
6	Name ($n_{p,i}$)
1	Separator
12	Comment ($c_{p,i}$)
1	Separator
10	Padding
1	Separator
10	Padding
1	Separator
10	Padding

Table 1: Vector representation of parameter datapoint.

Length	Features
1	Datapoint Type ([1 0 0 ... 0])
1	Separator
6	Function Name (n_f)
1	Separator
12	Function Comment (c_f) *if present, otherwise* Docstring (d_f)
1	Separator
10	Return Comment (c_r)
1	Separator
10	Return Expressions (e_r)
1	Separator
10	Parameter Names (n_p)

Table 2: Vector representation of return datapoint.

First, a function must have at least one type in t_p or it must have t_r, otherwise, it cannot serve as training data. A function must also have at least one return expression in r_e, since we do not want to predict the type for a function that does not return anything.

Furthermore, for functions where n_p contains the parameter `self`, we remove this parameter from n_p, t_p and c_p, since this parameter has a specific role for accessing the instance of the class in which the method is defined in. Therefore, the name of this parameter does not reflect any information about its type and is thus not relevant.

Finally, we do not predict the types `None` (can be determined statically) and `Any` (is always correct). Thus, we do not consider a function for predicting a parameter type if the parameter `Any`, and a return type if the return type is `Any` or `None`.

2.4 Vector Representation

From the selected function tuples, we create a parameter datapoint for each parameter and a return datapoint. We convert these datapoints to a vector. We explain the structure of these vectors in 2.4.1. All textual elements are converted using word embeddings (see 2.4.2), and types with one-hot encoded (see 2.4.3).

2.4.1 Datapoints and Vector Structure

The format of the input vectors is shown in Table 1 for parameter datapoints, and in Table 2 for return datapoints. All elements of the features have size 100. This results in a 55×100 input vector.

The lengths of the features are based on an analysis of the features in our dataset. The results are shown in Table 3. A full analysis is available in our GitHub repository (see section 3.1).

The datapoint type indicates whether the vector represents a parameter or a vector. A separator is a 1-vector of size 100. For parameter datapoints, padding (0-vectors) is used to ensure that the vectors for both datapoints have the same dimensions.

2.4.2 Learning Embeddings

It is important that semantically similar words result in vectors that are close to each other in the n-dimensional vector space, hence we cannot assign random vectors to words. Instead, we train an embeddings model that builds upon Word2Vec [18]. Since the meaning of certain words within the context of a (specific) programming language are different than the meaning of those words within the English language, we cannot use pre-trained embeddings.

We train embeddings separately for comments and identifiers. Comments are often long (sequences of) sentences, while identifiers can be seen as short sentences. Similarly to [5], we train two embeddings, because the identifiers "tend to contain more source code-specific jargon and abbreviations than comments".

Using the trained model, we convert all textual elements in the datapoints to sequences of vectors.

For the training itself, all words that occur 5 times or less are not considered to prevent overfitting. Since Word2Vec learns the context of a word by considering a certain amount of neighbouring words in a sequence, this amount of set to 5. The dimension of the word embedding itself is found by counting all the unique words found in the comments and identifiers and taking the 4th root of the result as suggested in [**?**]. This results in a recommended dimension of 14.

2.4.3 Representing Types

The parameter types and return type are not embedded, however, we also encode these elements as vectors. We use a one-hot encoding [19] that encodes to

Feature	Average Length	Median Length	Max Length	Chosen Length	Fully Covered Data Points
Function Name*	2.28	2	11	6	99,77%
Function Comment**	6.31	5	482	15	97,98%
Return Comment	8.40	5	533	6	65,98%
Return Expressions	6.01	3	1810	12	89,52%
Parameter names	2.84	2	72	10	97,67%
Parameter Name*	1.45	1	8	6	99,99%
Parameter Comment	6.38	4	2491	15	93,46%

Table 3: The chosen vector lengths of the features.

	Type	Percentage
1	`str`	28,9%
2	`int`	11,7%
3	`bool`	10,8%
4	`float`	2,9%
5	`Dict[str, Any]`	2,4%
6	`Optional[str]`	2,2%
7	`List[str]`	2,1%
8	`dict`	1,5%
9	`Type`	1,4%
10	`torch.Tensor`	1,3%

Table 4: The top 10 most frequent types in our dataset.

vectors of length $|T_{frequent}|$, where $T_{frequent}$ is the set of types that most frequently occur within the dataset. We also add the type "other" to $T_{frequent}$ to represent all types not present in the set of most frequently occurring types. We only select the most frequent types because there is not enough training data for less frequent types, resulting in a less effective learning process. The resulting vector for a type has all zeros except at the location corresponding to the type, for example, the type `str` may be encoded as $[0, 1, 0, 0, ..., 0]$.

We limit the set $T_{frequent}$ to the 1000 most frequent types, as this has shown to be an effective number in earlier work [5]. We show the top 10 of the most frequent types in Table 4.

2.5 Training the RNN

Given the vector representations described in section (2.4) we want to learn a function that would map the input vectors x of dimensionality k to one of the 1000 types T, hence that would create the mapping $\mathbb{R}^{x*k} -> \mathbb{R}^{|T|}$. To learn this mapping, we train a recurrent neural network (RNN). An RNN has feedback connections, giving it memory about previous input and therefore the ability to process (ordered) sequences of text. This makes it a good choice when working with natural language information.

We implement the RNN using LSTM units [20]. LSTM units have been successfully applied in NL2Type [5], where the choice for LSTMs is made based on the use for classification tasks similar to our problem. We describe the full details of the model in 3.4.1.

2.6 Prediction using the trained RNN

After training is done, the model can be used to predict the type for new, unseen, functions. The input to the model is similar to the input during the training phase. This means that first a function needs to be collected from an AST (section 2.1), then the function elements need to be preprocessed (section 2.2, and finally, the function must be represented as multiple vectors for the parameter types and for the return type as described in 2.4.

The model can now be queried for the input vectors to predict the corresponding types. The network outputs a set of likely types together with the individual probability of the correctness of these types.

3 Evaluation

We evaluate the performance of DLTPy by creating an implementation, collecting training data, and training the model based on this data. We judge the results using the metrics precision, recall, and F1-score. We also perform experiments using variants of DLTPy for comparison.

3.1 Implementation

We implement DLTPy in Python. We use GitPython [21] for cloning libraries, astor [22] to parse Python code to ASTs, docstring_parser [23] for extracting comment elements, and NLTK [24] for lemmatization and removing stopwords. We train two word embedding models using the Word2Vec [18] library by gensim [25]. The data is represented using Pandas dataframes [26] and NumPy vectors [27]. Finally, the prediction models are developed using PyTorch [28], a machine learning framework.

We make the source for training and evaluating DLTPy publicly available at `https://github.com/casperboone/nl2pythontype/`.

3.2 Experimental Setup

We collect training data from open-source GitHub projects. We first select projects by looking at `mypy` [10] dependents in two ways. First, we look at the dependents listed on GitHub's dependency graph [4]. Then, we complete this list with the dependents listed by `Libraries.io` [5]. The intuition is that Python projects using the type checker `mypy` are more likely to have types than projects that do not have this dependency. This results in a list of 5,996 projects, of which we can download and process (section 2.1) 5,922 projects. 74 projects fail, for instance due to the unavailability of a project on GitHub.

Together, these projects have 555,772 Python files. 5,054 files cannot be parsed. The files contain 4,977,420 functions, of which only have 585,413 functions have types (at least one parameter type or a return type).

We complement this dataset with projects for which type hints are provided in the Typeshed repository[6]. These are curated type annotations for the Python standard library and for certain Python packages. We manually find and link the source code of 35 packages, including the standard library. Using retype we insert the type annotations into the source code [29]. These projects have 29,759 functions, of which 246 have types.

We randomly split this set of datapoints into a non-overlapping training set (80%) and a test set (20%). We repeat every experiment 3 times.

All experiments are conducted on a Windows 10 machine with an AMD Ryzen 9 3900X processor with 12 cores, 32 GB of memory, and an NVIDIA GeForce GTX 1080 Ti GPU with 11 GB of memory.

3.3 Metrics

We evaluate the performance based on the accuracy of predictions. We measure the accuracy in terms of precision (the fraction of found results that are relevant), recall (the fraction of relevant results found) and F1-score (harmonic mean of precision and recall). Because in many contexts (for example IDE auto-completion) it is not necessary to restrict to giving a single good prediction, we look at the top-K predictions. Specifically, we collect the three metrics for the top-1, top-2, and top-3 predictions.

[4]https://github.com/python/mypy/network/dependents

[5]https://libraries.io/pypi/mypy/usage

[6]https://github.com/python/typeshed

We define p as the total number of predictions, p_{valid} as the number of predictions for which the prediction is not `other` (and thus the model cannot make a prediction). We define the three metrics as follows:

- $top\text{-}K\ precision = \frac{p_{valid\_correct}}{p_{valid}}$,
  where $p_{valid\_correct}$ is the number of valid predictions for which the correct prediction is in the top-K
- $top\text{-}K\ recall = \frac{p_{valid\_correct}}{p}$,
- $top\text{-}K\ F1 = 2 \times \frac{top\text{-}K\ precision \times top\text{-}K\ recall}{top\text{-}K\ precision + top\text{-}K\ recall}$.

3.4 Experiments and Models

While we mainly evaluate the performance of DLTPy as described in the previous section, we also try, evaluate and compare the results when we use different models or different input data. We train three different models and compare these to the model presented in section 2.5. Also, we evaluate the results of selecting different input elements.

3.4.1 Models

We implement and train three models to evaluate and compare their performance.

- **Model A** In the first model, we made use of two stacked LSTMs. Both LSTMs have a hidden size of 14. The first LSTM will feed its sequence fully into the second, whereas for the second LSTM we only use the output of the last unit and feed it into a fully connected linear layer. Softmax is applied to generate an approximation of the probability for each type. The model has a total of 37,288 parameters.
- **Model B** In the second model, we take an approach similar to Model A: we feed the input vector into a Gated Recurrent Unit (GRU) [30] and feed the output to a fully connected linear layer converting into the output types. The model has a total of 11,780 parameters.
- **Model C** The third model is the model architecture proposed by the authors of [5]. A single bi-directional LSTM with a hidden layer of size 256 is fed into a fully connected layer with output size 1000. Due to the size of this model and the time it took to train, the training consisted of only 25 epochs, compared to 100 epochs for the other three models. The model has a total of 404,456 parameters.

Dataset	Size	Parameter Datapoints	Return Datapoints	Dimensions
1 Complete	84,661	64,690	19,971	55 × 14
2 Optional parameter and return comment	863,936	719,581	144,355	55 × 14
3 Optional docstring	1,018,787	719,581	299,206	55 × 14
4 Without return expressions	84,661	64,690	19,971	55 × 14
5 Without return expressions, lower dimension	84,661	64,690	19,971	42 × 14

Table 5: The sizes of the datasets used to evaluate DLTPy.

3.4.2 Datasets

We try and evaluate variations in input data to measure the impact of certain elements in the datapoints. To this purpose, we create five datasets. The size of these datasets is listed in Table 5.

1. **Complete** This is the dataset as described in section 2. All datapoints in this dataset are complete. This means that the parameter datapoints have c_p, and the return datapoints have c_f, c_r, and e_r.

2. **Optional parameter and return comment** In this dataset we make the parameter and return comment optional. The presence of a docstring is still required. This means that parameter datapoints do not have to have c_p (91,01%), and the return datapoints do not have to have c_r (84,79%), but still have c_f (which is either the parsed function comment or the docstring) and e_r.

3. **Optional docstring** In this dataset we make the docstring optional. This means that parameter datapoints do not have to have c_p (91,01%), and the return datapoints do not have to have c_f and c_r (92,66%), but still have e_r (51,75%). This can be seen as a dataset without comments since only 20,52% of the datapoints in this set have comments. This means that the prediction for parameters is purely based on the parameter name, and for return datapoints, the prediction is based on the function name, the return expressions, and the parameter names.

4. **Without return expressions** To evaluate the usefulness of including return expressions in the model input, we perform the classification task also without return expressions. In this dataset all vectors representing parts of return expressions are 0-vectors.

5. **Without return expressions, lower dimension** This dataset is similar to the previous one. In this dataset, however, all vectors representing parts of return expressions are removed, resulting in lower-dimensional input vectors.

4 Results

In this section we present the results of DLTPy. First, we show and compare the results of our different experiments as described in section 3.4 using the metrics described in section 3.3. Then, we compare the results to previous work, specifically to NL2Type [5].

4.1 Models

We presented three different architectures for our prediction model (section 3.4. In Table 6 we present the results. The underlined metric scores indicate the best score for each model. The bold metric scores indicate the best overall score. We use these scores to compare the performance of the models.

Model C clearly outperforms models A and B in all metrics. The top-1 F1-score is 82.4%, and for the top-3 it is 91.6%. The top-3 recall is 91.9%, this can be interpreted as the model predicts the correct type within its first three suggestions in 91.9% of the cases it was asked to make a prediction for.

Model B performs poorly compared to model A and C. Using a GRU does not have a positive influence on the results. It is interesting to note that where model C performs best on dataset 1, model B does not benefit from the return expressions and performs better without them (dataset 4).

The performance of model A lies in between the performance of model B and C. Interestingly, this model performs best on the dataset with the lowest dimensions. Dataset 4 and 5 contain the same data, the only difference is that dataset 4 has an additional separator vector and twelve 0-vectors. The same impact can also be seen for model C when comparing dataset 4 and 5. A possible explanation could be that the learning process can converge faster because there is simply less to learn, however, we have not found a provable cause for this.

4.2 Input Elements Selection

We presented the elements of the datapoints of DLTPy and four variations (section 3.4) that we compare with.

Model	Dataset	Top-1			Top-2			Top-3		
		Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
A	1 Complete	60.3	67.7	63.8	71.7	76.7	74.1	76.8	81.2	79.0
	2 Optional parameter and return comment	51.4	62.0	56.2	65.6	69.7	67.6	72.7	75.7	74.1
	3 Optional docstring	52.9	62.8	57.4	66.1	70.1	68.0	72.9	75.6	74.2
	4 Without return expressions	60.2	68.6	64.1	72.8	77.5	75.0	78.2	82.0	80.1
	5 Without return expressions, lower dimension	63.9	70.3	66.9	74.8	78.6	76.6	80.1	83.2	81.6
B	1 Complete	41.2	54.7	47.0	55.2	63.8	59.2	61.7	70.0	65.6
	2 Optional parameter and return comment	34.3	51.0	41.0	44.3	57.2	49.9	54.4	63.6	58.6
	3 Optional docstring	36.6	52.7	43.2	45.7	57.4	50.9	56.5	64.1	60.1
	4 Without return expressions	42.2	55.9	48.1	55.8	64.5	59.8	62.2	70.5	66.1
	5 Without return expressions, lower dimension	40.4	54.7	46.5	54.7	63.8	58.9	61.1	69.8	65.2
C	1 Complete	**81.7**	**83.2**	**82.4**	**88.4**	**89.2**	**88.8**	**91.4**	**91.9**	**91.6**
	2 Optional parameter and return comment	69.3	70.0	69.6	78.5	78.3	78.4	83.5	83.1	83.3
	3 Optional docstring	70.7	71.2	71.0	79.8	79.2	79.5	84.5	83.8	84.1
	4 Without return expressions	79.1	81.3	80.2	86.6	88.0	87.3	90.0	90.9	90.4
	5 Without return expressions, lower dimension	81.0	82.8	81.9	88.1	89.0	88.5	91.2	91.6	91.4

Table 6: The evaluation results of DLTPy.

	Top-1			Top-3		
	Prec.	Rec.	F1	Prec.	Rec.	F1
DLTPy	81.7	**83.2**	**82.4**	91.4	**91.9**	91.6
DLTPy Dataset 4	79.1	81.3	80.2	90.0	90.9	90.4
NL2Type	**84.1**	78.9	81.4	**95.5**	89.6	**92.5**

Table 7: A comparison of DLTPy to NL2Type [5].

In Table 6 we present the results. For this comparison, we look at the results of model C, the best performing model.

The results of dataset 2 and 3 are significantly worse than of dataset 1. This shows that the natural language contained in comments positively influences the type classification task. Interestingly, the performance of dataset 3 is slightly less good than of dataset 2, while dataset 2 contains more comments.

Another observation from the results is that the predictions positively influence from including return expressions in the dataset. We see that the performance is less good when return expressions are not included (datasets 4 and 5) than when they are included (dataset 1).

4.3 Comparison to previous work

In this section, we compare our results against NL2Type [5]. DLTPy operates in a similar way on similar data as NL2Type. However, NL2Type predicts types for JavaScript files and DLTPy predicts types for Python files. Also, DLTPy uses a different input representation (different feature lengths and return expressions as an addition) and a different word embedding size (14 instead of 100). This makes it interesting to compare our results against this work.

From the results, we cannot observe that DLTPy significantly outperforms or underperforms NL2type (see Table 7). It is, however, interesting to note that without return expressions (dataset 4), DLTPy would underperform. Another interesting observation is that the results did not have a negative impact on the significantly smaller word embedding size.

5 Conclusion

Our work set out to study the applicability of using natural language to infer types of Python function parameters and return values. We present DLTPy as a deep learning type inference solution based on natural language for the prediction of these types. It uses information from function names, comments, parameter names, and return expressions.

The results show that DLTPy is effective at predicting types using natural language information. The top-1 F1-score is 82.4%, and the top-3 F1-score is 91.6%. This shows that in most of the cases the correct answer is in the top 3 of predictions. The results show that using natural language information from the context of a function and using return expressions have a positive impact on the results of the type prediction task.

We do not significantly outperform or underperform NL2Type. Without our additions to the ideas behind NL2Type, however, DLTPy would underperform. This shows that the main idea behind NL2Type, namely using natural language information for predicting types, is generalizable from JavaScript to Python, but additional information, such as return expressions, is needed to get comparable results.

We identify two threats to the validity of our results. The first threat is that there is no separation of functions between the training and test set on project level. Because functions within the same project are more likely to be similar, this might influence the validity of our results. Also, since the best performing model, model C, has 404,456 parameters and the best performing dataset, dataset 1, has just 84,661 datapoints, which increases the risk of overfitting.

DLTPy has limitations that can be improved upon in future work. Dataset 1, the complete dataset, is relatively small. A better data retrieval strategy that goes beyond looking at mypy dependents might result in more data points and thus allows for better training resulting in more accurate results. Furthermore, the predictions of DLTPy are currently restricted to the 1000 most frequently used types in Python. Open type predictions would improve the practical use of DLTPy, given that there is enough training data available for the types that are less frequent.

6 Reflection

This section is added for IN4334 - Machine Learning for Software Engineering and should not be considered as a part of this paper.

During the project we faced the following challenges:

- Because we are not using a ready-made dataset, we had to gather the data ourselves. This turned out to be quite complicated. The data fetching process as described in the paper takes quite a long time (to run), and because we, of course, did not get the pre-processing steps completely right the first time (or even the first three times), we had to redo the process a couple of times. Even after paralleling this process and running it on a cloud server instance, it still cost us a lot of time.

- It is not very easy to find typed Python projects. NL2Type uses Typescript projects which means they can easily filter on language, but we did not have that option. Therefore we looked at mypy dependents and hoped that they would have functions with type annotations. This was not always the case, only 11.8% of the functions had types, and only 0.4% made it into our "complete" dataset (after filtering on comments for instance). Initially we were only planning to select the top-*x* projects, sorted by stars, but in the end, we needed the data of *all* projects we found to come to a reasonably sized dataset.

- We expected to able to reuse more of NL2Type but found that not always all necessary details were mentioned or that the implementation was missing. Because of this, we had to spend more time on this than expected, giving us less time for new innovations.

- Python functions turn out to be much less consistently commented than functions in TypeScript. If they even have specific comments for parameters or return values, there are still multiple formats in which these can be reported. We spent some time on writing a parser for these different formats, or at least the most common ones.

- During the training phase, we had issues with PyTorch reporting CUDA issues. The issue had to do with memory allocation, which is definitely much more low level than the level we were working on and could control. Because of this issue, we were not able to get results for one of our model architectures and have left this architecture out.

References

[1] "Interactive: The Top Programming Languages," 2019. [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019

[2] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," *Communications of the ACM*, vol. 60, no. 10, pp. 91–100, 9 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3144574.3126905

[3] E. Meijer and P. Drayton, "Static Typing Where Possible , Dynamic Typing When Needed : The End of the Cold War Between Programming Languages," in *OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004*, New York, NY, USA, 2004.

[4] M. M. Vitousek, A. M. Kent, J. G. Siek, J. Baker, M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker, "Design and evaluation of gradual typing for python," in *Proceedings of the 10th ACM Symposium on Dynamic languages - DLS '14*, vol. 50, no. 2. New York (New York, USA): ACM Press, 2014, pp. 45–56. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2661088.2661101

[5] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript Function Types from Natural Language Information," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 5 2019, pp. 304–315. [Online]. Available: https://ieeexplore.ieee.org/document/8811893/

[6] S. Hanenberg, S. Kleinschmager, R. Robbes, E. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19,

no. 5, pp. 1335–1382, 10 2014. [Online]. Available: http://link.springer.com/10.1007/s10664-013-9289-1

[7] "Flow," 2014. [Online]. Available: https://flow.org/

[8] "TypeScript," 2012. [Online]. Available: https://www.typescriptlang.org/

[9] "HackLang," 2014. [Online]. Available: https://hacklang.org/

[10] "mypy," 2012. [Online]. Available: http://mypy-lang.org/

[11] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. New York, New York, USA: ACM Press, 2016, pp. 607–618. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2950290.2950343

[12] M. Salib, "Faster than C: Static type inference with Starkiller," *PyCon Proceedings, Washington DC*, vol. 3, 2004.

[13] R. A. MacLachlan, "The python compiler for cmu common lisp," *ACM SIGPLAN Lisp Pointers*, no. 1, pp. 235–246, 1992.

[14] M. Hassan, C. Urban, M. Eilers, and P. Müller, "MaxSMT-Based Type Inference for Python 3," in *CAV (2), volume 10982 of Lecture Notes in Computer Science*. Springer, Cham, 7 2018, pp. 12–19. [Online]. Available: http://link.springer.com/10.1007/978-3-319-96142-2_2

[15] E. Maia, N. Moreira, and R. Reis, "A static type inference for python," *Proc. of DYLA*, 2012.

[16] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 11 1997.

[17] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 6 2012, pp. 837–847. [Online]. Available: http://ieeexplore.ieee.org/document/6227135/

[18] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," 1 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[19] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman, *Applied linear statistical models*. Irwin Chicago, 1996, vol. 4.

[20] F. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: continual prediction with LSTM," in *9th International Conference on Artificial Neural Networks: ICANN '99*, vol. 1999. IEE, 1999, pp. 850–855. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/cp_19991218

[21] "GitPython," 2008. [Online]. Available: https://github.com/gitpython-developers/GitPython

[22] "astor," 2012. [Online]. Available: https://github.com/berkerpeksag/astor

[23] "docstring_parser," 2018. [Online]. Available: https://github.com/rr-/docstring_parser

[24] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[25] R. Řeh uřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, 5 2010, pp. 45–50.

[26] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56.

[27] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 3 2011. [Online]. Available: http://ieeexplore.ieee.org/document/5725236/

[28] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," in *NIPS Autodiff Workshop*, Long Beach, CA, USA, 2017.

[29] "retype," 2017. [Online]. Available: https://github.com/ambv/retype

[30] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*. Association for Computational Linguistics (ACL), 2014, pp. 1724–1734.