Predicting Type Annotations for Python using Embeddings from Graph Neural Networks

Vladimir Ivanov, Vitaly Romanov and Giancarlo Succi
Innopolis University, Russia

Keywords: Neural Networks, Source Code, GNN, CNN, Embeddings, Type Prediction, Python.

Abstract: An intelligent tool for type annotations in Python would increase the productivity of developers. Python is a dynamic programming language, and predicting types using static analysis is difficult. Existing techniques for type prediction use deep learning models originated in the area of Natural Language Processing. These models depend on the quality of embeddings for source code tokens. We compared approaches for pre-training embeddings for source code. Specifically, we compared FastText embeddings to embeddings trained with Graph Neural Networks (GNN). Our experiments showed that GNN embeddings outperformed FastText embeddings on the task of type prediction. Moreover, they seem to encode complementary information since the prediction quality increases when both types of embeddings are used.

1 INTRODUCTION

Statically typed programming languages allow detecting some errors during the development process and before the execution. This is one reason why dynamically typed programming languages like Python and JavaScript adopt optional type hinting. However, providing type hints is labor-intensive, and it can often be neglected. A helper tool that would assist programmers with type annotations is needed.

Researchers have tried to address type prediction using the tools of machine learning and deep learning. Some results were achieved by analyzing documentation for functions (Boone et al., 2019; Malik et al., 2019). However, documentation sometimes is not available, especially for small projects and projects in the early stages of development. Other approaches viewed the source code as a sequence of tokens and the type prediction task — as sequence tagging (Pradel et al., 2019; Hellendoorn et al., 2018). A handful of studies interpreted the source code as a graph (Raychev et al., 2015; Allamanis et al., 2020; Wei et al., 2020).

This paper takes a hybrid approach to type prediction for Python. We view source code as both: a sequence of tokens and as a graph. We adopt a common Natural Language Processing (NLP) approach for sequence tagging to predict Python types. The information from a source code graph is incorporated into the prediction process by concatenating embeddings from the graph built from source code with the FastText embeddings trained on a corpus of Python repositories. The work that is closest to ours is (Allamanis et al., 2020). They predict type annotations for Python using Graph Neural Networks. However, we use a hybrid technique that combines the strength of NLP-based approaches and the strength of Graph Neural Networks for modeling long-range dependencies.

Our experiments show that introducing graph-based embeddings into the prediction process allows increasing the type prediction quality from 68% (using NLP model) to 75% when measured with the top-1 F-score. The contribution of this paper is the following:

- We prepare a dataset for predicting types for Python variables[1];
- We demonstrate a method that allows incorporating graph-based source code embeddings into a standard NLP processing pipeline;
- We demonstrate that graph-based embeddings are complementary to FastText embeddings for source code and that using them allows increasing the quality of type prediction for Python variables.

The rest of this paper is organized as follows. Section 2 overviews existing approaches for predicting type annotations for dynamic programming lan-

[1] https://github.com/VitalyRomanov/python-type-prediction

Ivanov, V., Romanov, V. and Succi, G.
Predicting Type Annotations for Python using Embeddings from Graph Neural Networks.
DOI: 10.5220/0010500305480556
In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 1*, pages 548-556
ISBN: 978-989-758-509-8

guages. Section 3 explains data collection and processing methods. Section 4 overviews the design of our type prediction model. Section 5 explains the experimental setup. Section 6 discusses limitations of the current approach. Section 7 concludes the paper.

2 RELATED WORK

Researchers have explored the possibility of predicting type annotations for dynamic languages already for some time. The earliest most relevant work on this subject is related to predicting type annotations for JavaScript programs using Conditional Random Fields (CRF) (Raychev et al., 2015).

Another approach for predicting types in JavaScript relies on extracting relevant information from JSDoc (Malik et al., 2019). The authors used a natural language description for each argument to predict the required type. Their approach has demonstrated a decent performance, but it relies on documentation that is not always available. A similar approach was used for Python in DLTPy (Boone et al., 2019). Researchers used variable names, docstring, comments, and return expressions as input features. This approach also relies on meaningful names and does not take into account the program's structure.

Three works are the closest to what we are trying to build. The first approach was presented in (Hellendoorn et al., 2018). In this work, authors treat a program as a sequence of tokens and process Python functions using a Recursive Neural Network (RNN). They point out that relevant information about a variable can be spread over a long distance in source code. For example, one can guess the type of a variable by looking at the usage patterns. However, relevant usages can be spread over long distances in a function. Their solution to this problem is to take the representations for all mentions of a target variable and average them together. This averaged representation is used for type prediction. Another approach based on RNNs is called TypeWriter (Pradel et al., 2019). This approach combines probabilistic and search-based predictions. First, the types of variables are predicted using an RNN. Then, the predicted type undergoes a refinement process to verify the validity of the predicted annotation. Because of the second step, this approach can ensure type correctness. The third approach is called Typilus (Allamanis et al., 2020) where authors represented source code as a graph and addressed an open world type prediction problem, where the set of target type classes is not predefined.

Our method is based on a text Convolutional Neural Network (CNN). Similarly to (Pradel et al., 2019; Hellendoorn et al., 2018), we process the source code as a sequence of tokens. However, we also introduce additional information in the form of graph embeddings. Graph embeddings naturally address long-range dependencies in source code because such dependencies are represented as adjacent nodes in a graph. Moreover, they were demonstrated to be useful for predicting return types for Python functions (Romanov, 2020).

3 DATASET

Our goal is to explore the effectiveness of graph embeddings for predicting types for Python variables. However, we were not able to find suitable datasets for this task. We are aware of two datasets with Python functions that also come with a parsed abstract syntax tree (AST): 150k Python Dataset (Raychev et al., 2016) and CodeSearchNet (Husain et al., 2019). Unfortunately, ASTs in these datasets are limited to the scope of a single function, and it is impossible to resolve function calls reliably. We believe that the information about function calls is important for predicting types. For this reason, we decided to collect our own dataset.

3.1 Source Code Graph

Our dataset consists of a collection of interdependent packages. We selected a set of popular Python libraries and their dependencies from GitHub. We filtered those packages that contain type annotations in their sources. The remaining were analyzed with an open-source indexing tool Sourcetrail [2], which stores the source code index in graph format where source code elements, such as modules, methods, classes, and variables are represented as nodes. Edges represent relationships between nodes, for example, definitions or function calls. The process of package indexing is time-consuming, and we plan to index more packages in the future. The example of Sourcetrail output is shown in Figure 1. The list of indexed packages can be found in Appendix.

Sourcetrail resolves the mentions of variables, functions, and classes in the source code. However, it does not store information from AST. For this reason, we implemented a tool for creating a graph representation of Python AST and merging this representation with the Sourcetrail graph index. We used a standard

[2] https://www.sourcetrail.com/

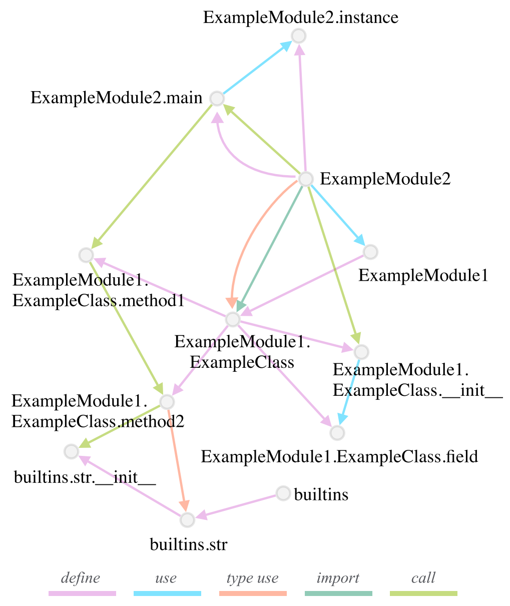


Figure 1: Example of source code graph that captures definition and usage of a simple Python class.

Python library for AST parsing.

Some nodes in the AST have identical names. This is usually the case for AST nodes that do not represent named elements in the code such as variables and functions. The examples of these other AST nodes are control flow statements such as for, if, or while, operators, assignments, slices, attributes, and others. To avoid these nodes being merged, which would result in different functions exchanging information during the message-passing stage in GNN, we create a new node for each instance of AST node that does not represent a user-assigned name. Overall, the graph construction process is similar to (Romanov et al., 2020).

The final graph contains 690676 nodes and 2971025 edges (Table 1). The inclusion of AST nodes dramatically increases the graph size. An example that extends the graph in Figure 1 is shown in Figure 2. Due to the size of the graph, only a fragment of it is shown here. The source code that was used for producing this graph is given in Appendix.

AST nodes and edges are provided by Python's *ast* package. The number of different node and relationship types reaches several dozens. However, the GNN model that we use for computing graph embeddings does not distinguish node types. The reason is discussed in Section 4. We believe that it is important to include information about node types into the final graph because it possesses high predictive power. For this reason we add AST type nodes into the graph.

When predicting types inside Python function's body, we match source code elements with graph nodes and use graph embeddings for predictions.

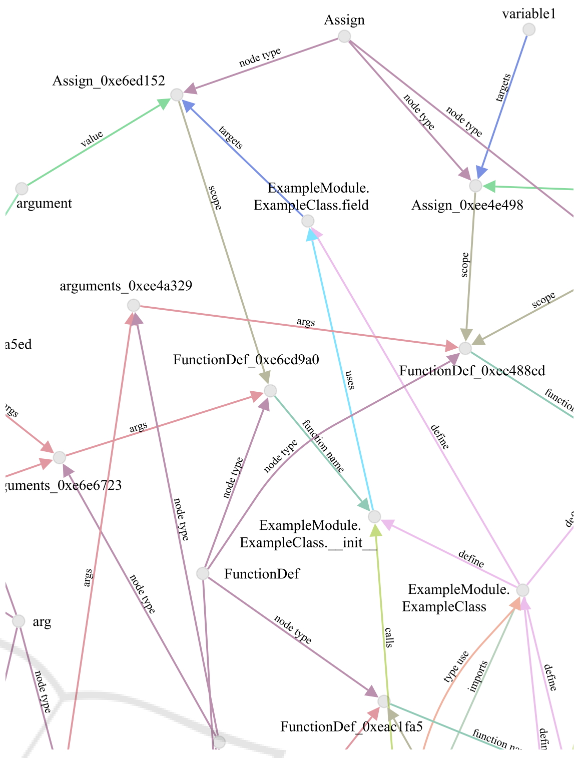


Figure 2: Example of source code graph including AST edges. Due to the size of the graph, only a fragment of the graph is shown here. AST nodes that do not represent user-defined names are assigned a unique identifier.

3.2 Python Type Annotations

For the task of type annotation, we extract labels from the source code itself. Type annotations in Python can be provided for: (i) variables defined in the function's signature, (ii) variables defined inside the function's body, and (iii) the function itself (return type). In this work, we focus on predicting type annotations for variables and leave the prediction of the function's return type out of consideration. We strip type annotations from source codes and use them as labels within Named Entity Recognition (NER) framework. Overall, our dataset contains 2166 functions with 4548 type labels.

Some types are composite, for example `List[Str]` or `Union[Str, Int]`. We decided to simplify these annotations and predict only the most outer-level type description, e.g., (`List` and `Union`). A similar approach was used in (Allamanis et al., 2020). We resort to this measure to simplify the neural network architecture in the first stages of research since we can use a simple classifier to

Table 1: Node and edge types present in Python source graph.

Node Type	Node Count	Edge Type	Edge Count
AST related	643369	AST related	2179648
Function	28863	Usage	588120
Class field	6961	Function Call	91882
Global Variable	4156	Defines	49584
Class	3857	Type Usage	42977
Module	1780	Import	16389
Class Method	1286	Inheritance	2424

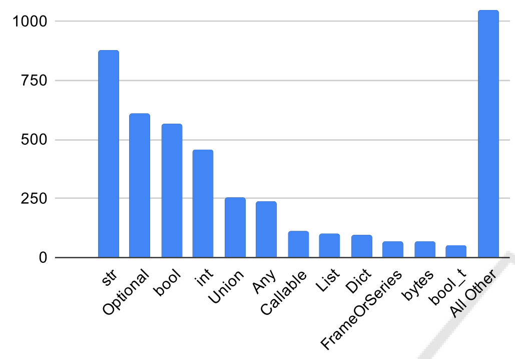


Figure 3: Distribution of type labels for the task of predicting type annotations in Python. Only the top 12 labels are demonstrated. Top 20 labels were used in this paper for experiments.

predict labels. In the future, we plan to explore more complex prediction strategies, similar to one presented in (Hellendoorn et al., 2018). Overall we have 127 unique type labels. However, the distribution of the labels in our dataset seems to follow the power law. To ensure that the learning model has enough training data for each label, we use the top 20 type annotations and replace the rest with a label *Other*. Again, similar filtration was also applied in (Allamanis et al., 2020). The distribution of the labels is given in Figure 3.

Sometimes docstring contains information useful for type inference (Boone et al., 2019). We wanted to find out how well a model can predict types when no such documentation is available. In other words, we wanted our models to perform inference by only reading the code. For this reason, we remove all documentation from the functions that we process.

4 MODEL

We explore the usefulness of two representations of source code for predicting type annotations for Python: representation as a sequence and representation as a graph. We decided to treat type annotations the same way Named Entities are treated in NLP. We chose a model that classifies each token as a part of an entity or an outside token. Specifically, we adopted the BILUO tagging scheme.

We wanted to compare the performance of our model with and without the graph embeddings. For this reason, we wanted to adopt an architecture where we can easily add new information to the model. We chose an architecture from (Collobert et al., 2011) that received the Test of Time Award at ICML 2018. We use this architecture because of the simplicity of implementation and fast training times. In our future work, we plan to experiment with more up-to-date language modeling techniques for code (Kanade et al., 2020; Feng et al., 2020). Our current architecture is shown in Figure 4.

4.1 Source Code Embeddings

First, we tried predicting type annotations without the use of graph embeddings. Instead, we used embedding models popular in NLP. Our original goal was to use CuBERT embeddings for source code that was presented in (Kanade et al., 2020). Unfortunately, the model was not published for public access at the time we completed our work. Our second choice was to use pre-trained FastText embeddings for Python presented in (Efstathiou and Spinellis, 2019). However, we found that this model does not have embeddings for many language keywords and punctuation characters that seem to be useful for predicting type annotations. We decided to pre-train our own FastText embeddings that preserve frequent tokens. For this, we downloaded repositories listed in CodeSearchNet (Husain et al., 2019) and extracted Python sources. As a result, we had a Python corpus of 2.8 GB, 2 GBs more than the corpus used in (Efstathiou and Spinellis, 2019). We trained FastText embeddings using Gensim (Řehůřek and Sojka, 2010) for 20 epochs with standard parameters (context size 5, vector size 100).

4.2 Graph Embeddings

FastText embeddings were designed to address representation learning for natural languages. Specifically, natural language models usually try to predict a word inside a given context. However, objectives like this are arguably less useful for source code, primarily due to a greater amount of neologisms and the presence of long-range dependencies (Allamanis et al., 2018). At the same time, we can provide information about dependencies explicitly by extracting relationships between variables, functions, and classes from sources. Moreover, we can use additional source-code-specific objectives. We decided to use the following objectives:

- **Node Name Prediction:** where we predict names for classes, variables, or functions. This objective is semantically challenging. The names can be predicted by looking at usage patterns or the implementation;
- **Variable Use Prediction:** where we predict the names of variables that are used in the body of a given function. This objective captures the topical semantics of the code. For example, if we know that this function is related to networking, it is likely to have variable names such as *client* and *server*. Variable names are predicted for nodes that represent functions. These nodes aggregate information from all AST nodes in the function's body;
- **Next Call Prediction:** where we predict which function will be called next given the current function. This objective encourages the model to learn usage patterns of different functions. Once again, the objective is applied to nodes for functions to encourage the usage of AST nodes.

We train graph embeddings using a version of Graph Neural Network called Relational Graph Convolutional Network (Schlichtkrull et al., 2018). A distinctive feature of this model is that it incorporates relationship types into the modeling process. RGCN is a message-passing model. During training and inference, nodes share their embeddings with adjacent neighbors. A learned relationship-type-specific function is applied to node representation during the message passing. It transforms a node embedding before sending it to the adjacent node. Each node incorporates messages from other nodes into its own embedding. In our case, we use three message passing turns. On each turn, a different message-passing transformation is applied, which corresponds to three layers of Graph Neural Network. The final embeddings are used as an input for our three objective functions. The use of a relation-specific GNN model is motivated by the high diversity of relationship types in our source code graph.

All nodes in our graph have the same node type, and the only way to understand that we are working with a variable and not with an *if* expression is by looking at the relationships with the adjacent neighbors. The incorporation of node types into the graph leads to a much higher requirement for memory and processing time. For this reason, instead of assigning types to nodes, we introduced nodes that represent types. We use such nodes only in relationships with AST nodes. Such an approach introduces little memory overhead but still allows the inclusion of node types.

The use of shared nodes (e.g. type nodes or nodes that represent user-defined names) is a concern for ensuring that the graph that represents an implementation of a function is isolated from all other functions. Since the message passing is used for training, shared nodes can be gateways for information leakage between parts of the graph that represent completely unrelated code. To prevent this we restrict the message passing from shared nodes only in one direction.

4.3 Type Prediction Model

We use a text CNN model for predicting Python types. The model is inspired by (Collobert et al., 2011). The main reason for choosing this model was the simplicity of the implementation and training speed. The overview of the model is given in Figure 4. In the first stage of the prediction process, the tokens of a Python's function are embedded using FastText and graph embeddings. Additionally, we add embeddings for prefixes and suffixes for tokens to improve the ability to process rare short tokens. Prefixes and suffixes have a fixed length of three characters. All embeddings for each token are concatenated and then passed to a convolutional layer. The context size depends on the window size. We explore different window sizes during hyper-parameter optimization.

After passing data through two convolutional layers, embeddings for each token are passed through two fully connected layers that produce the final label on the output. We use the BILUO scheme for encoding the output classes for each token. We evaluate the performance of the model using the top-1 F1-score.

5 EXPERIMENT

For training, we used a desktop computer with Intel i5-7400 CPU and GeForce 1050Ti GPU. We trained

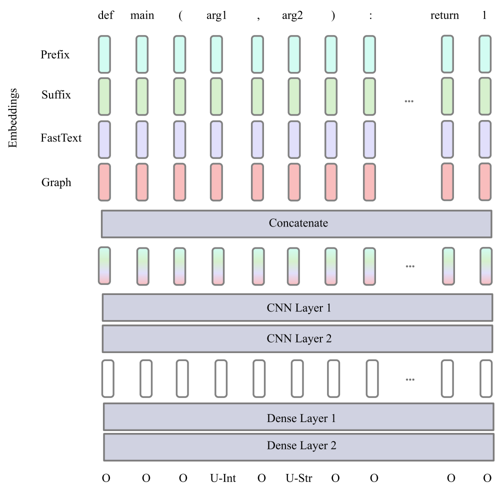


Figure 4: Architecture of our simple CNN model for type prediction.

FastText embeddings for source code (dimensionality 100) using Gensim [3] library for 20 epochs. We trained graph embeddings (dimensionality 100) using an RGCN model developed with DGL [4] library for 100 epochs. The training time was about 8 hours for both FastText and GNN embeddings. We performed a hyper-parameter search by tuning the performance of GNN embeddings on the test set. The best performance was achieved when we used three message passing turns (3 layers) of RGCN. Using more message passing turns resulted in a decreased performance. This phenomenon is well-known (Wu et al., 2020) but is not yet completely resolved.

In addition, we performed a hyper-parameter search for a text CNN model. The training time for each trial took about 2.5 hours. Overall, we explored 33 hyper-parameter configurations for CNN-type predictor. The search was performed with and without the inclusion of graph embeddings in order to evaluate their impact. Optimized parameters include the number of channels in CNN layers, the dimensionality of dense layers, the dimensionality of prefix and suffix embeddings, the size of the context window, the learning rate. The best performance was achieved when we used 40 CNN channels for both layers, 30 units for both dense layers, context size 5, prefix and suffix embedding size 50, and learning rate 0.0001.

For each hyper-parameter set, we trained a model several times to account for randomness in the initialization. We selected hyper-parameters only based on the best trial. Sometimes models would fail to converge during one of the trials but successfully converge during others.

[3]https://github.com/RaRe-Technologies/gensim
[4]https://github.com/dmlc/dgl

We tested four embedding approaches for text CNN. The first one explores type prediction for Python when embeddings only for suffixes and prefixes are used. These embeddings are trained specifically for the task of type prediction. The performance of this model likely demonstrates the dependence of type prediction performance on variable names as well as the present default values.

In the second model, we add FastText embeddings. This case serves as a valid baseline for our model. FastText embeddings are pre-trained on a large corpus of source code and can capture semantic dependencies between variable names. Again, these embeddings primarily demonstrate the importance of variable names for type prediction. The FastText embeddings are frozen during the training.

In the third model, we used embeddings for prefixes and suffixes as well as graph embeddings. This model allows assessing how useful graph embeddings are when compared with FastText embeddings. Graph embeddings should encode more structural information about source code. The graph embeddings are frozen during the training.

The fourth model involves training a model that utilizes prefix, suffix, FastText, and graph embeddings. This experiment would help us identify whether FastText and graph-based embeddings are complementary to each other. The FastText and graph embeddings are frozen during the training. The results of these experiments are shown in Table 2.

From the results, we see that the FastText baseline allows predicting a significant portion of the type annotations correctly — F-score 68.2. When using graph embeddings, the performance achieves 71.7. This indicates that the information learned by graph embeddings appears to be beneficial for type prediction. Finally, when using both kinds of embeddings, the F-score jumps 75.19. This suggests that FastText and graph-based embeddings appear to contain complementary information that is useful for predicting variable types. The demonstrated results are similar to ones reported in (Allamanis et al., 2020). For the comparison of our results with their work, we refer the reader to the next section.

Examples of predicted types can be found in Appendix.

6 DISCUSSION OF RESULTS

Our work is the most similar to (Allamanis et al., 2020): structured representation of code and GNNs are used for type prediction. However, we started our experiments before this paper was published and

Table 2: Best Python type prediction performance for models with different embeddings.

Embeddings	Top-1 F-score
Prefix, Suffix	61.7
Prefix, Suffix, FastText	68.2
Prefix, Suffix, Graph	71.7
Prefix, Suffix, FastText, Graph	75.19

therefore we have a different dataset and different architectures. Further, we will perform a comparison with respect to the dataset, architecture, and performance.

The dataset used by (Allamanis et al., 2020) is an order of magnitude larger. Their dataset consists of 592 packages (118 440 files), and ours only from 132 (67 486 files). Moreover, their dataset contains 252 470 annotated symbols (possibly with annotated return types) whereas ours has only 4548 annotated variables (no return type annotations). The huge difference comes from the fact that they perform additional type inference using *pytype* to make the dataset larger. The ways the datasets were collected are drastically different. They build their graph from AST and the binary code produced by Python. Our approach involves indexing with an open-source tool called Sourcetrail. Packages in our graph are interdependent and connected to each other whereas (Allamanis et al., 2020) analyze disassociated repositories. Their graph has less than dozen relationship types while we preserve all AST types.

(Allamanis et al., 2020) perform a comparison of their GNN approach with seq2seq and path-based models. Out of those models that they have implemented, we can compare ours with Seq2Class and Graph2Class. Their Seq2Class model is based on BiLSTM and processes the functions as sequences. It achieves the performance of 64% which is similar to our model that used FastText embeddings. The Graph2Class model uses Gated Graph Neural Network for processing graph representations. The performance of this model is 75% for common Python types, which, again, is similar to our results. However, their GNN uses 8 layers, whereas ours uses only three. The validity of this comparison is under question because the sizes of the datasets and the number of unique types in datasets are dramatically different.

It is very challenging to perform a fair performance comparison. First, the dataset sizes are different. The performance of their approach is provided per epoch, which will be different when the datasets of different sizes are used. Second, they train their models from scratch for the sole task of type prediction. We use pre-training of FastText embeddings and GNN embeddings, both of which can be later reused for other tasks. Third, the hardware is also very different. They use a K80 GPU whereas we had to pre-train FastText and GNN embeddings using CPU and only used GTX 1050i for training text CNN. They report that their model takes 86 seconds of GPU time for one epoch. Our GNN model takes on the order of 200-300 seconds of CPU time per epoch. We have to account for the fact that our GNN model performs pre-training on three objectives and computationally more expensive. However, this results in graph embeddings that later can be reused. Our text CNN model takes 18s per epoch to train on GPU. The proper comparison of performance is challenging because the graph size, function lengths, efficiency of the implementation - all have to be taken into account. If we assume that the GNN embeddings are pre-trained and shared between different tasks, then the CNN model alone is far less complex and less computationally intensive than a Gated Graph Neural Network used by (Allamanis et al., 2020). Moreover, our model uses 3 message passing turns for pre-training instead of 8 message passing turns as in the other work. When dealing with graphs, increasing the number of message-passings results in an exponential increase of complexity.

Nevertheless, both works demonstrate that GNNs are useful for the task of type annotation. Moreover, we perform pre-training of graph embeddings that can be also applied for other tasks such as variable misuse detection or name suggestion (since these tasks rely on names, and our pre-training objectives optimize for correct names).

It is worth noting that in the field of Natural Language Processing, text CNN and BiLSTM models were demonstrated to have an inferior performance compared to transformers. A comparison of more modern GNN models (Busbridge et al., 2019) versus transformers (Kanade et al., 2020; Feng et al., 2020) should be performed.

7 CONCLUSION

We explored different types of embeddings for predicting types for Python variables. Our experiments showed that type prediction depends a lot on the current variable names because the prefix and suffix embeddings alone are great predictors for the target type label. Models that rely on lexical embeddings (such as FastText) show decent performance even with a simple CNN model. Moreover, embeddings that were designed to model source code structure allow achieving a greater performance than lexical embeddings. Using both types of embeddings resulted in the best performance in our experiments. We demonstrated that the achieved performance is similar to recent work by other researchers while being less computationally intensive.

REFERENCES

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. (2020). Typilus: neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105.

Boone, C., de Bruin, N., Langerak, A., and Stelmach, F. (2019). Dltpy: Deep learning type inference of python function signatures using natural language context. *arXiv preprint arXiv:1912.00680*.

Busbridge, D., Sherburn, D., Cavallo, P., and Hammerla, N. Y. (2019). Relational graph attention networks. *arXiv preprint arXiv:1904.05811*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537.

Efstathiou, V. and Spinellis, D. (2019). Semantic source code models using identifier embeddings. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 29–33. IEEE.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. (2018). Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 152–162.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR.

Malik, R. S., Patra, J., and Pradel, M. (2019). Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315. IEEE.

Pradel, M., Gousios, G., Liu, J., and Chandra, S. (2019). Typewriter: Neural type prediction with search-based validation. *arXiv preprint arXiv:1912.03768*.

Raychev, V., Bielik, P., and Vechev, M. (2016). Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Raychev, V., Vechev, M., and Krause, A. (2015). Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124.

Řehůřek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta. ELRA.

Romanov, V. (2020). Evaluating importance of edge types when using graph neural network for predicting return types of python functions. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 25–27.

Romanov, V., Ivanov, V., and Succi, G. (2020). Representing programs with dependency and function call graphs for learning hierarchical embeddings. In *ICEIS (2)*, pages 360–366.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. (2018). Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer.

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

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*.

APPENDIX

Source Code for Illustrations

The examples on the Figures 1 and 2 were generated using a small example presented in the Listing 1.

Prediction Examples

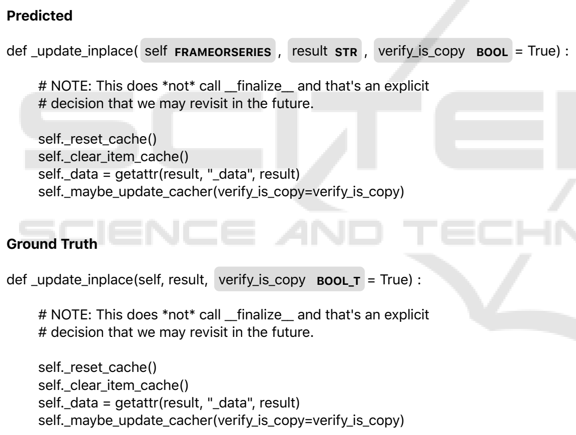
Figure 5 demonstrates a prediction example where a model misjudged the type of arguments *self* and *results*. In Figures 6 and 7 the results of successful type prediction are demonstrated.

List of Indexed Packages

absl-py, boto, cymem, h5py, jsonpatch, mkl-fft, openstacksdk, pbr, pycrypto, pytest, Scrapy, termcolor, Werkzeug, ansible, catalogue, debtcollector, httplib2, jsonpointer, mkl-random, opt-einsum, Pillow, PyDispatcher, pytest-runner, service-identity, testtools, wrapt, appdirs, certifi, decorator, hyperlink, jsonschema, mkl-service, osc-lib, plac, PyHamcrest, python-dateutil, shade, thinc, zipp, asn1crypto,

```
# ExampleModule2.py
from ExampleModule1 import ExampleClass

instance = ExampleClass(None)

def main():
    print(instance.method1())

main()

# ExampleModule1.py

class ExampleClass:
    def __init__(self, argument):
        self.field = argument

    def method1(self):
        return self.method2()

    def method2(self):
        variable1 = 2
        variable2 = str(2)
        return variable2
```

Listing 1: A simple class definition and use in Python. Source code graph for this code is given in Figure 1.



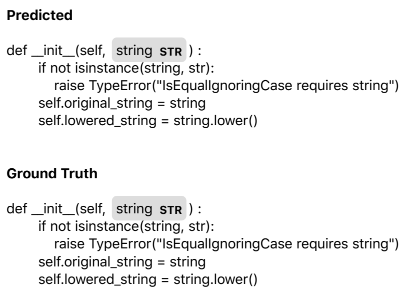
Predicted

```
def _update_inplace( self FRAMEORSERIES , result STR , verify_is_copy BOOL = True) :

    # NOTE: This does *not* call __finalize__ and that's an explicit
    # decision that we may revisit in the future.

    self._reset_cache()
    self._clear_item_cache()
    self._data = getattr(result, "_data", result)
    self._maybe_update_cacher(verify_is_copy=verify_is_copy)
```

Ground Truth

```
def _update_inplace(self, result, verify_is_copy BOOL_T = True) :

    # NOTE: This does *not* call __finalize__ and that's an explicit
    # decision that we may revisit in the future.

    self._reset_cache()
    self._clear_item_cache()
    self._data = getattr(result, "_data", result)
    self._maybe_update_cacher(verify_is_copy=verify_is_copy)
```

Figure 5: Example of incorrect type prediction. Model predicts incorrect types for variables **self** and **result**.



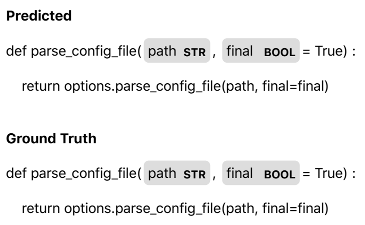
Predicted

```
def __init__(self, string STR ) :
    if not isinstance(string, str):
        raise TypeError("IsEqualIgnoringCase requires string")
    self.original_string = string
    self.lowered_string = string.lower()
```

Ground Truth

```
def __init__(self, string STR ) :
    if not isinstance(string, str):
        raise TypeError("IsEqualIgnoringCase requires string")
    self.original_string = string
    self.lowered_string = string.lower()
```

Figure 6: Model correctly predicts type for function argument.



Predicted

```
def parse_config_file( path STR , final BOOL = True) :
    return options.parse_config_file(path, final=final)
```

Ground Truth

```
def parse_config_file( path STR , final BOOL = True) :
    return options.parse_config_file(path, final=final)
```

Figure 7: Model correctly predicts type for function argument.

cffi, Django, idna, Keras-Applications, monotonic, os-client-config, pluggy, PyNaCl, python-ironicclient, simplejson, tornado, zope.interface, astor, chardet, dogpile.cache, importlib-metadata, Keras-Preprocessing, more-itertools, oslo.i18n, preshed, pyOpenSSL, python-mimeparse, six, tqdm, atomicwrites, Click, extras, incremental, keystoneauth1, msgpack, oslo.serialization, prettytable, pyparsing, pytz, spacy, traceback2, attrs, cliff, fabric, invoke, kiwisolver, munch, oslo.utils, Protego, pyperclip, PyYAML, sqlparse, Twisted, Automat, cmd2, fixtures, iso8601, linecache2, murmurhash, os-service-types, protobuf, PyQt5, queuelib, srsly, unittest2, Babel, constantly, Flask, itsdangerous, lxml, netaddr, packaging, py, PyQt5-sip, requests, stevedore, urllib3, bcrypt, cryptography, gast, Jinja2, Markdown, netifaces, pandas, pyasn1, PyQtWebEngine, requestsexceptions, tensorboard, w3lib, blis, cssselect, google-pasta, jmespath, MarkupSafe, numpy, paramiko, pyasn1-modules, pyrsistent, scikit-learn, tensorflow, wasabi, bokeh, cycler, grpcio, joblib, matplotlib, olefile, parsel, pycparser, PySocks, scipy, tensorflow-estimator, wcwidth