ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference

Kevin Jesse
krjesse@ucdavis.edu
University of California, Davis
Davis, CA, USA

Premkumar T. Devanbu
ptdevanbu@ucdavis.edu
University of California, Davis
Davis, CA, USA

ABSTRACT

In this paper, we present ManyTypes4TypeScript, a very large corpus for training and evaluating machine-learning models for sequence-based type inference in TypeScript. The dataset includes over 9 million type annotations, across 13,953 projects and 539,571 files. The dataset is approximately 10x larger than analogous type inference datasets for Python, and is the largest available for TypeScript. We also provide API access to the dataset, which can be integrated into any tokenizer and used with any state-of-the-art sequence-based model. Finally, we provide analysis and performance results for state-of-the-art code-specific models, for baselining. ManyTypes4TypeScript is available on Huggingface, Zenodo, and CodeXGLUE.

KEYWORDS

Type Inference, Machine Learning, TypeScript, Code Properties

ACM Reference Format:
Kevin Jesse and Premkumar T. Devanbu. 2022. ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528507

1 INTRODUCTION

There is considerable interest recently in the application of machine learning (ML) models to a variety of software-related tasks and datasets. ML has largely focused on improving performance, using probabilistic models of source code that exploit code's regularity and patterns [4]. The type-inference problem is one such task where probabilistic code models work well. Probabilistic type guessers can infer types for developers, helping them avoid type errors, and lowering the annotation effort [9]. TypeScript and Python have been the primary languages targeted by researchers [27, 34]. Recent ML-based methods [5, 11, 12, 23–26, 32] appear to work well, but are hard to compare, due to variability in evaluation practices.

The field of type inference varies quite a bit, in methods, data, and metrics. With the abundance of open source repositories, new methods often mine their own data or attempt to sample similar data from previous work [5, 12, 26, 32]. Despite these works often using similar metrics, performance is confounded with scoring differences and sampling bias. Scoring differences arise when various subsets of types are evaluated and not others, for example, based on frequency (top-100), location (parameter, and function level), and annotation type (user-defined). Sampling bias occurs from type inference papers sampling different projects or files at various commits where code context and the annotations themselves can differ. Though there have been some attempts at standardized comparisons for instance DeepTyper [11] and NL2Type [21], Typilus [5] and Type4Py [23], other recent publications showed quite a bit of variance in evaluation, *e.g.* some used Top 100 types [24]; some compare across different projects; others use the same projects, but at different time slices. We feel there is still a need for a comprehensive TypeScript dataset and metrics.

To help standardize training and evaluation for TypeScript type inference, we offer the ManyTypes4TypeScript dataset. This comprehensive dataset includes over 9 million type annotations, which is 10x more annotations than the next largest Python annotated dataset ManyTypes4Py [22]. The ManyTypes4TypeScript also comes with evaluation scripts, enabling models to be properly benchmarked against the test set. We make all of our collection scripts, unprocessed data (Zenodo[1]), processed API dataset (Huggingface[2]), usage examples, and evaluation script publicly accessible. The dataset was collected in mid January of 2022 for publicly available GitHub projects. Our contributions are as follows:

- A dataset containing a comprehensive set of code snippets and aligned type annotations across 13,953 TypeScript projects resulting in 9M type annotations.
- Standardized access across a range of state-of-the-art models on 🤗Huggingface.
- Standardized scoring with metrics and existing evaluation of three state-of-the-art models.
- Additional word tokenized data for flexible model input, allowing choice of sub-tokenization methodologies. We include the mining scripts so the SE community can update the dataset as needed.

All of our code is publicly available. In the next section we discuss the collection process and parsing of projects.

2 COLLECTION PROCESS AND PARSING

Figure 1 illustrates the collection process and parsing from project to machine learning dataset. First we use GraphQL[3] to gather a list of ~29,500 public TypeScript projects on GitHub. The GraphQL query returns TypeScript projects by the number of GitHub stars

[1] https://zenodo.org/record/6387001
[2] https://huggingface.co/datasets/kevinjesse/ManyTypes4TypeScript
[3] https://graphql.org

Table 1: Statistics Across Data Splits

Split	Train	%	Test	%	Validation	%
Projects	11,413	81.8%	1,336	9.58%	1,204	8.62%
Files	486,477	90.16%	28,045	5.20%	25,049	4.64%
Examples	1,727,927	91.95%	81,627	4.34%	69,652	3.71%
Types	8,696,679	95.33%	224,415	2.46%	201,428	2.21%

The data set is split across projects.

Table 2: JSON schema in ManyTypes4TypeScript

JSON Field	Type	Description
`tokens`	list[string]	Sequence of tokens (word tokenization)
`labels`	list[string]	A list of corresponding types
`url`	string	Repository URL
`path`	string	Original file path that contains token sequence
`commit_hash`	string	Commit identifier in the original project
`file`	string	File name

to ensure the collection of quality projects. After mining the list of projects, a custom bash script attempts to install packages, types, and other requirements with Pnpm[4]. This is important for compiler inferred types as inferred types largely come from resolved package dependencies. Each file's AST (abstract syntax tree) is traversed, extracting both human annotations as well as compiler-inferred annotations. The traversal, gathers the tokens and labels types on the AST nodes. The types are removed and the tokens are pushed onto a queue. The types are aligned to the token sequence to create an aligned pair. This process is repeated recursively for each directory that contains a "`tsconfig.json`". The final output from our parser is a json for each project. We aggregate the project outputs and prepare the data for de-duplication.

De-duplication is essential, as shown by Allamanis [3], prior to training machine learning models; duplication can result in biased performance estimates. Lopes *et al.*[19] identified a large amount of near-duplicate code on GitHub; Allamanis [3] released a tool based on Jaccard similarity to help the community avoid this issue. We run the de-duplication tool[5] on the raw corpus to find & remove duplicates. Out of 1,128,744 original files, 204,358 duplicates (about 18%) were found and removed, leaving 924,386 files. After filtering files with annotations 539,571 files remained. The de-duplication is done *without* type annotations, to ensure that even differently annotated duplicates are safely removed; this is different from Mir *et al.*[22]. Mir *et al.*[22] performs lemmatization over variables for classic NLP techniques like TF-IDF. This limits input choices for model developers. With the adoption of subtokenization, subtokenizers *pretrained* on large code corpora are trained to tokenize complete token sequences. By leaving the sequences tokenized in contiguous words, it is up to the model designer to determine how to represent the input. Techiques include: words, identifier splitting [30], BPE [28], WordPiece [15], SentencePiece [16], lemmatization, etc. This is paramount as Shi *et al.*[29] recently showed that splitting identifiers when combined with BPE subtokens can improve performance.

The de-duplicated set of token sequences, type annotations, and type meta-information is split by projects ~80%/10%/10% which provides a file split of ~90%/5%/5% for train/test/validation respectively. More information on the data split can be found in Table 1. As shown in Figure 1, the JSONL unprocessed data splits are uploaded to Zenodo. Next we define a output vocabulary size of 50,000 and replace any type that exceeds rank 50,000 with an `UNK` token. In classification tasks with finite vocabulary, a special type token `UNK` represents a type guess that exceeds the classifiers prediction capabilities. This is a function of the model and can be changed for models using a larger or smaller classification layer. Additionally, the uninformative "`any`" type annotation is removed from the training and evaluation data. These are standard practices for classification tasks. The schema of files in the Huggingface dataset can be found in Table 2. Table 2 consists of tokens, labels, repository url, file path, commit hash and file name. This schema is fed into the dataloading script and can also be found on the Huggingface "Dataset card". Finally, the custom Huggingface dataloading script, named `ManyTypes4TypeScript.py`, can be used to generate and push the dataset to the Huggingface *hub*. This script is available on the Zenodo dataset page so anyone can "fork" a customized ManyTypes4TypeScript dataset.

In the next section, we discuss the design choices of our API Huggingface dataset and how the design of the Datasets Hub [17] provides easy to use, optimally compressed access to over 12GB of type inference data.

3 DATASET DESIGN AND USEABILITY

The ManyTypes4TypeScript dataset conforms to the Huggingface Datasets specification for several reasons. First, the compatible Huggingface transformers library incorporates state-of-the-art models including code specific models like CodeBERT [8], GraphCodeBERT [10], and CodeBERTa [33] which has been widely used across the field especially in CodeXGlue [20] for a wide set of tasks and model probing [14]. New advancements in transformers are often integrated into Huggingface, thus permitting new applications to existing tasks in addition to easily accessible models [1, 8, 10, 31]. It is our goal to make the type inference task as widely applicable to new state of the art transformers with ManyTypes4TypeScript. In later sections we discuss our application of ManyTypes4TypeScript on three SOTA models.

Second, another reason for hosting ManyTypes4TypeScript on Huggingface are the efficiency and scale capabilities. The datasets are capable of being cached completely once downloaded and mapping operations i.e subtokenization and subtoken label alignment are also cached. The datasets are stored as compressed `.parquet` files with Git-LFS (large file storage) and work seamlessly with all available tokenizers and feature-extraction tools. Massive datasets can also be streamed. Model training and evaluation can be accelerated with the Huggingface accelerate[6] library which is particularly helpful for sequence tagging efficiency.

Finally, the tokenizer, dataset and any transformer model can be instantiated in the following five lines of code (LOC).

[4] https://pnpm.io
[5] https://github.com/Microsoft/near-duplicate-code-detector
[6] https://github.com/huggingface/accelerate

Figure 1: The collection and parsing process of ManyTypes4TypeScript

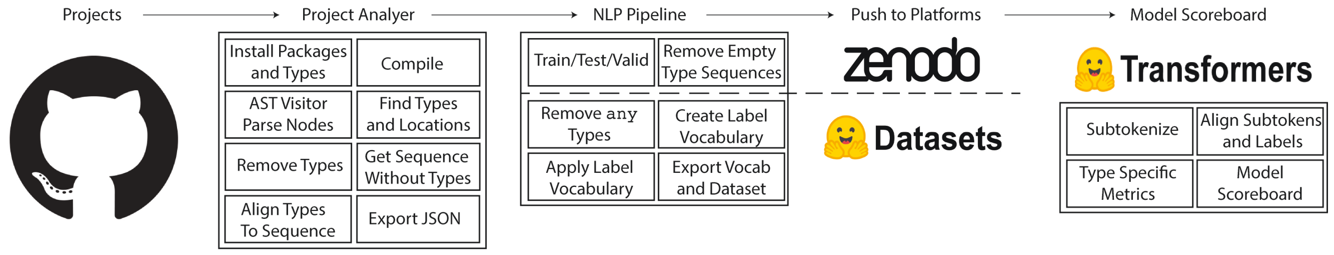


Figure 2: Frequency of annotation locations in ManyTypes4TypeScript.

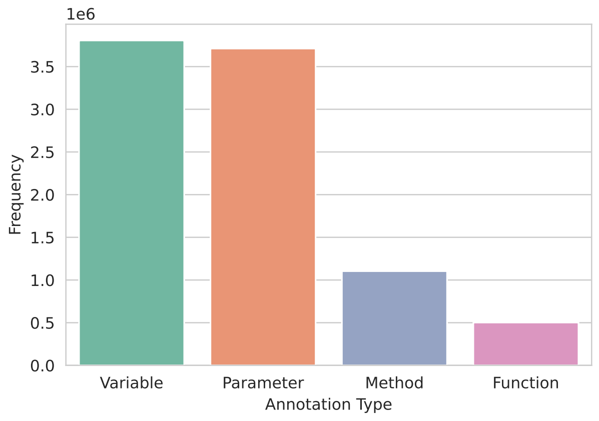
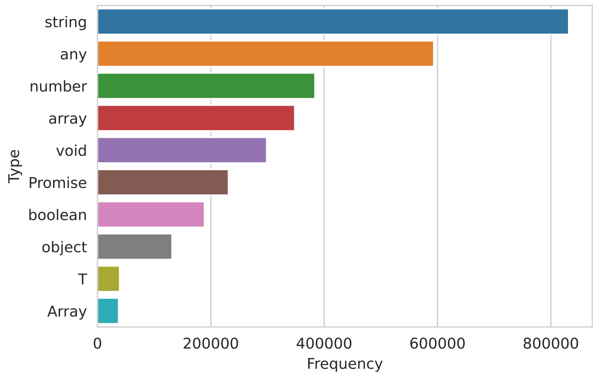


Figure 3: Top 10 most frequent types in ManyTypes4TypeScript.



(1) The dataset is downloaded from Huggingface or instantiated from a local directory.

```
dataset = load_dataset('kevinjesse/
    ManyTypes4TypeScript')
```

(2) Then the tokenizer is instantiated.

```
tokenizer = AutoTokenizer.from_pretrained('
    microsoft/graphcodebert-base')
```

(3) The dataset is tokenized into subtokens and the labels are aligned with our provided `align_labels` function to map labels to the first subtoken.

```
tokenized_dataset = dataset.map(align_labels)
```

(4) The label list is extracted from the ManyTypes4TypeScript meta data.

```
label_list = tokenized_dataset["train"].features[f"
    labels"].feature.names
```

(5) The weights for GraphCodeBert [10] are instantiated with a projection layer fit to ManyTypes4TypeScript type vocabulary.

```
model = AutoModelForTokenClassification.
    from_pretrained('microsoft/graphcodebert-base'
    , num_labels=len(label_list))
```

With the above steps, one can instantiate a model with the ManyTypes4TypeScript dataset; the model developer has end-to-end control of model input and output schemes. For example, the model developer can use the `GraphCodeBERT` contextual embeddings for a kNN (k-nearest neighbor) search rather than a classification layer; this would effectively expand the closed-vocabulary output.

The closed type output of the Huggingface API dataset is fixed to 50,000 type categories; but is amenable with the dataset scripts on Zenodo. The current type vocabulary on Huggingface covers approximately 94.08% of all type occurrences as most types are "common" types. The remaining types placed in the `UNK` category cover approximately 5.92% of the 9M types. These types are local and infrequent types, where the types occur less than 10 times corpus wide. Figure 2 represents the frequency of type annotation locations where the majority are variable declarations and function parameters with 3.8 million and 3.7 million annotations respectively. Figure 3 represents the frequency of the top 10 most frequent types in the ManyTypes4TypeScript corpus. The majority of types are `string`, `any`, and `number`. With a large majority of human and compiler inferred types resolving to the uninformative "any" type, probabilistic type inference has the potential to increase type coverage; type coverage in the optional type setting reaches traditional static typing when all types are annotated or inferred. Finally in Figure 4, we examine the ratio of compiler inferred types to human annotations in ManyTypes4TypeScript. We examine that most types are mixed between compiler inferred and human annotations. Corpus wide, the ratio is approximately 57% inferred types to 43% human annotated types. Figure 4 shows that only 20% of "any" are labeled by humans and the vast majority are inferred by

Figure 4: The ratio by percentage of developer vs. inferred annotations by type in the top 50 most frequent types.

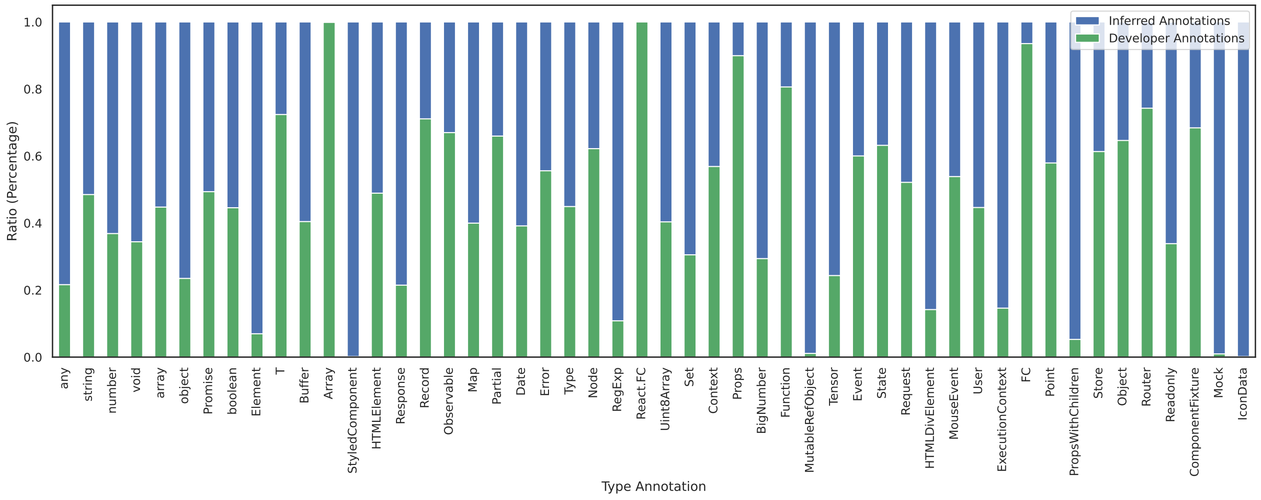


Table 3: Accuracy Comparisons On ManyTypes4TypeScript.

Model	**Top 100**				**Overall**			
	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy
CodeBERT [8]	84.58	85.98	85.27	87.94	59.34	59.80	59.57	61.72
GraphCodeBERT [10]	**84.67**	**86.41**	**85.53**	**88.08**	**60.06**	**61.08**	**60.57**	**62.51**
CodeBERTa [33]	81.31	82.72	82.01	85.94	56.57	56.85	56.71	59.81
PolyGot [2]	84.45	85.45	84.95	87.72	58.81	58.91	58.86	61.29
GraphPolyGot [2]	83.80	85.23	84.51	87.40	58.36	58.91	58.63	61.00
RoBERTa [18]	82.03	83.81	82.91	86.25	57.45	57.62	57.54	59.84
BERT [7]	80.04	81.50	80.76	84.97	54.18	54.02	54.10	57.52

Top 100 types are the most frequent 100 types. Overall is scored with all type locations. UNK is considered incorrect.

the compiler. The compiler resolves the type to be any when the compiler cannot determine the type from existing type constraints. Quantifying a model's ability to resolve the "any" type is a possible derivative work from our dataset as "any" type annotations are available in the Zenodo data. Lastly, in Figure 4, some types are all or nearly all human annotations. This is a unique opportunity for type inference models to assist compilers, alert developers to must have annotations, and resolve types accordingly.

In the next section, we discuss tracking models' performances with a public scoreboard and pushing models trained on the ManyTypes4TypeScript dataset to the Huggingface model hub.

4 TRACKING PERFORMANCE AND REPRODUCIBILITY

The ManyTypes4TypeScript dataset on Huggingface is integrated with "Papers With Code"[7] which tracks new papers with consistent metrics. The ManyTypes4TypeScript dataset on Huggingface keeps a list of all models trained or "fine-tuned" on ManyTypes4TypeScript. The models that are trained and evaluated on ManyTypes4TypeScript and pushed to the model hub are linked to the ManyTypes4TypeScript datacard *viz.* homepage. These models can be downloaded and verified in section 3. The ManyTypes4TypeScript is currently being integrated into the CodeXGLUE[8] [20] set of tasks. CodeXGLUE is a benchmark dataset and open challenge for code intelligence managed by Microsoft Research. With ManyTypes4TypeScript, there is a community driven approach to adding datasets, metrics, models, and documentation to institute a standardization across the type inference task for TypeScript. Next we discuss our supplied metrics.

5 TASK SPECIFIC METRICS AND SCORES

In the dataset on Zenodo, standard sequence evaluation scripts `seqeval`[9] are available to evaluate the sequence predictions. We modify the ground truth and predictions such that scoring subsets of types can be done easily. We permit classic tagging scoring, considering UNK predictions as incorrect, and top-100 type scoring. The community can add various subsets to the existing metrics

[7] https://paperswithcode.com/dataset/manytypes4typescript

[8] https://microsoft.github.io/CodeXGLUE

[9] https://github.com/chakki-works/seqeval

such as user-definition and location specific scoring. Our scoring metrics also permit per type evaluation. The dataset in CodeXGLUE will have detailed instructions and scripts to evaluate models, and these scripts will be used to track and verify the task leader-board.

Table 3 contains a list of state-of-the-art models scored with the aforementioned metrics. The performance of the models are similar in overall top 100 accuracy to Jesse *et al.*[12] which is completely pre-trained on JavaScript. The performance between the models is in line with previous comparisons [2, 14]. The models provided by us serve as baselines for future contributions. We intend to increase the number of models evaluated across ManyTypes4TypeScript including but not limited to: C-BERT [6], CuBERT [13], PLBart [1], and CodeT5 [31]. Additionally, we plan to increase the granularity of the metrics so specific outcomes can be evaluated *viz.* user-defined types.

6 CONCLUSION

In this paper, we present the ManyTypes4TypeScript dataset of over 9 million type annotations across 13,953 projects and 539,571 files. ManyTypes4TypeScript aims to facilitate the application of new advances in ML-based type inference, with easy to use APIs. ManyTypes4TypeScript standardizes evaluation with the provided test set, metrics, and baselines. By providing the tools used to extract ManyTypes4TypeScript and evaluate state-of-the-art models, we believe that the dataset itself can be a useful resource for the community to maintain and contribute to the type inference task for TypeScript.

ACKNOWLEDGMENTS

We gratefully acknowledge support from NSF CISE (SHF) Grant No. 1414172.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] Toufique Ahmed and Premkumar Devanbu. 2021. Multilingual training for Software Engineering. *arXiv preprint arXiv:2112.02043* (2021).

[3] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 143–153.

[4] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[5] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation.* 91–105.

[6] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, et al. 2020. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641* (2020).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 758–769.

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[11] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 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.* 152–162.

[12] Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1483–1486.

[13] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained contextual embedding of source code. (2019).

[14] Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1332–1336.

[15] Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959* (2018).

[16] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[17] Quentin Lhoest, Albert Villanova del Moral, Yacine Jernite, Abhishek Thakur, Patrick von Platen, Suraj Patil, Julien Chaumond, Mariama Drame, Julien Plu, Lewis Tunstall, et al. 2021. Datasets: A Community Library for Natural Language Processing. *arXiv preprint arXiv:2109.02846* (2021).

[18] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[19] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–28.

[20] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).

[21] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 304–315.

[22] Amir M Mir, Evaldas Latoskinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. *arXiv preprint arXiv:2104.04706* (2021).

[23] Amir M Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2021. Type4py: Deep similarity learning-based type inference for python. *arXiv preprint arXiv:2101.04470* (2021).

[24] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. 2020. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. *arXiv preprint arXiv:2004.00348* (2020).

[25] Yun Peng, Zongjie Li, Cuiyun Gao, Bowei Gao, David Lo, and Michael Lyu. 2021. HiTyper: A Hybrid Static Type Inference Framework with Neural Prediction. *arXiv preprint arXiv:2105.03595* (2021).

[26] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 209–220.

[27] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[28] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[29] Jieke Shi, Zhou Yang, Junda He, Bowen Xu, and David Lo. 2022. Can Identifier Splitting Improve Open-Vocabulary Language Model of Code? *arXiv preprint arXiv:2201.01988* (2022).

[30] Xiaobing Sun, Xiangyue Liu, Jiajun Hu, and Junwu Zhu. 2014. Empirical studies on the nlp techniques for source code data preprocessing. In *Proceedings of the 2014 3rd international workshop on evidential assessment of software technologies.* 32–39.

[31] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *arXiv preprint arXiv:2109.00859* (2021).

[32] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).

[33] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

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