TypeSpaceBERT: A Deep Similarity Learning-based CodeBERT Model for Type Inference

Xueyuan Chen
Delft University of Technology
Delft, The Netherlands
x.chen-47@student.tudelft.nl

Călin Georgescu
Delft University of Technology
Delft, The Netherlands
c.a.georgescu@student.tudelft.nl

Àlex De Los Santos Subirats
Delft University of Technology
Delft, The Netherlands
a.delossantossubirats@student.tudelft.nl

Atanas Marinov
Delft University of Technology
Delft, The Netherlands
a.marinov-1@student.tudelft.nl

Rover van der Noort
Delft University of Technology
Delft, The Netherlands
s.r.vandernoort@student.tudelft.nl

ABSTRACT

The introduction of gradual type annotations in legacy software systems increases their understandability and maintainability, and allows developers to easily discover previously undetected bugs. Machine learning techniques for type inference have proven to be promising solutions, automatically providing high-accuracy solutions without requiring much computational time. However, the results they achieve for user-defined and rare types drop significantly due to the data sparsity and the bounded type vocabulary that many models utilize. TypeSpaceBERT uses deep similarity learning to fine-tune CodeBERT for type inference allowing the model to have an unbounded type vocabulary and synthesize potential unseen types as a result of the pre-trained CodeBERT's general language understanding. This paper reports on the construction of the model and the rationale used for its design and fine-tuning. It provides an empirical evaluation that seeks to assess TypeSpaceBERT's performance in comparison to a classification-centric baseline, in terms of efficiency and scalability. The results show an increasing trend in the performance of the model related to the data; in particular, after 2% of the dataset of 9 million type instances, it goes from 0.026 accuracy to 0.032. We find that our approach significantly outperforms the baseline both in terms of prediction performance and scalability. Additionally, the discussion about the limitations of this study explains the lower accuracy achieved as a result of the time and budget constraints. Finally, we make some recommendations for future research to better generalize the results of this model.

1 INTRODUCTION

The type system is one of the defining components of every programming language. Type systems not only determine crucial attributes of programming languages, but also play a key role in steering the ways practitioners utilize and understand them. *Dynamically typed programming languages* such as Python and JavaScript type check the variables during run time. Their advantages are that they are generally easier to use and allow for a quicker development process. However, this comes at a cost, namely, weaker maintainability and an inability to catch errors at compile time.

To fully reap the benefits of a type system, some dynamically typed languages have been implementing ways for developers to use static typing in their code. For example, the Python community introduced optional type hinting in PEP 484 [10]. The JavaScript community coped with the issue by proposing an alternative - TypeScript, a gradually typed language. Gradually typed languages perform type checking at compile time only for the optionally provided typed variables. This way of type checking partly verifies the type safety of the program and thus guarantees certain properties. Without compromising flexibility, such strategies reduce type-related bugs by restricting and informing developers about type errors and other related defects.

However, this poses a problem when it comes to a large volume of code that had been written before the typing options were introduced to the language in question. The code is untyped and therefore can not benefit from the new changes. To address this issue, we distinguish three methods for solving this problem.

The simplest solution is to add the types manually. Clearly, this is a very cumbersome and time-consuming process. Compounding that, understanding code written by another individual is hard, making the whole procedure especially error-prone.

The second option is the utilization of static type inference methods. Despite improving both the time needed and the accuracy achieved compared to manual labor, static analysis methods are still imprecise. The main reasons for this are the dynamic language features which result in overestimations and produce results that achieve type matching, which is not exact [25]. For this reason, the end results are seldom satisfactory.

Finally, the third option, machine learning, has proven to be adequate in predicting the types of code fragments and thus accelerating the automation of development tasks. Still, ML solutions are not perfect and have suffer from several weaknesses. One of the main hurdles was that the vocabulary size for older attempts was limited [12, 20, 23, 32]. Over time there were various attempts to mitigate this using different architectural basis such as RNNs [21, 27], GNNs [2, 32] and LSTMs [20, 23]. These attempts culminated in the current state-of-the-art (SOTA), namely, the use of Transformers [31] as an architectural basis in conjunction with similarity learning. Transformers constitute the most widely adopted recent advancement in the field. This is because they are currently the backbone of established SOTA solutions for natural language processing (NLP) problems. The deeply ingrained semantic regularities and similarities between NL and PL [13] make them natural candidates for code-related tasks. One of the main advantages of the Transformer architecture is that it allows for greater context to be effectively exploited in the prediction [8, 11, 16].

More recent approaches address this hurdle by using type clustering methods to allow for an unbounded number of types [2, 14, 21]. Another benefit of using type clustering is that it can also allow for unseen type prediction. In the case of our model, although not tested due to lack of time, as it would require to re-train CodeBERT and the tokenizer, it should also be able to predict such unseen types [14].

Motivated by the shortcomings previous approaches had, we present TypeSpaceBERT, a fine-tuned CodeBERT extension with an unbounded type space that applies Deep Similarity Learning (DSL) to discriminate between the different types. We utilize the output embedding of the fine-tuned CodeBERT model to generate a d-dimensional type space. To infer a type, the input example is projected to the type space, and then k-nearest neighbors heuristic is used to calculate the output distribution. The fact that CodeBERT was originally pre-trained on a large corpus of multilingual code data provides the desired flexibility in our model. The framework we propose can be generalized and applied to any encoder model. Implementing our approach would result in a decoder model with an unbounded vocabulary size that could synthesize previously unseen types. To summarize, the main contributions of the paper are:

- TypeSpaceBERT, A DSL-based CodeBERT model, fine-tuned on ManyTypes4TypeScript that uses type clusters allowing it to have an unbounded vocabulary and providing it with the ability to synthesize previously unseen types.
- A classification version of CodeBERT that is fine-tuned for type prediction and then used as a comparable baseline to evaluate.
- A rigorous comparative evaluation of the model in terms of type prediction performance and scalability, that includes all standardized metrics for the type prediction task.

The remaining of the paper is structured as follows: first, Section 2 gives an overview of the historically significant and current SOTA methods for type inference. Following that, Section 3 introduces our TypeSpaceBERT and details the underlying design choices. Section 4 explains our decisions when setting up the evaluation. The empirical results and the threats to their validity are analyzed in Section 5 and Section 6, respectively. A discussion and potential future work are then presented in Section 7. Finally, Section 8 concludes the paper.

2 RELATED WORK

In recent years, code generation and translation problems have been receiving an increasing amount of attention from researchers and practitioners alike. Machine learning solutions have played an essential role in developing accurate techniques for tasks like type prediction. Many techniques have been proposed which seek to leverage both natural language (NL) and source code of different programming languages (PL) to best extract semantic information from large open-source code corpora. Specifically for the type prediction task, most efforts focus on Python, JavaScript, and TypeScript programming languages. This section aims to provide an extensive overview of the current literature on SOTA models for the type prediction task. We compile the surveyed prior type prediction work in Table 2.1, highlighting their most distinctive characteristics.

2.1 Fixed Type Vocabulary Approaches

The first work to show that vast open-source corpus can be exploited for synthesizing semantically powerful type inference algorithms is JSNice [28]. This approach established that meaningful representations can be effectively extracted from input programs using Conditional Random Fields (CRFs), an undirected graphical model which can predict properties of the source code. This work establishes several cornerstone concepts that would be utilized in most following work, including the need for capturing program structure, the value of processing NL elements of code (such as identifiers), and the possibility of casting type inference as either a Maximum a Posteriori (MAP) estimation or a clustering (similarity-based) problem. Future work would improve on the shortcomings of JSNice's dependency network.

Xu et al. [33] introduced the first probabilistic type prediction approach for Python. Previous work in the Python community had focused on static analysis and deterministic computation [9, 29]. The novelty of this approach comes from combining such techniques with a classifier that serves as the engine of a probabilistic inference engine. The framework leverages NL in source code by first fitting a classifier based on identifier names and their associated (statically inferred) types. The classifier is then used to construct a probabilistic constraint generator that establishes "requirements" that type predictions must match, with regard to data flow, sub-typing, and naming. Finally, a graphical probabilistic inference engine exploits these constraints by allowing for iterative belief propagation of type predictions of the form $p(\texttt{variable}, \texttt{type})$. This approach shows that probabilistic models and static type inference can be effectively conjoined, but lacks in its scalability potential due to the verbose graphical model, and is limited in terms of type vocabulary.

Hellendoorn et al. [12] describe DeepTyper, the first approach to mitigate the scalability shortcomings of previous work by taking advantage of powerful deep neural networks, a technique that has become the basis for SOTA models. DeepTyper is heavily inspired by NLP solutions and treats the task of type inference as a sequence-to-sequence generation task, where a sequence of tokens extracted from the source code is taken as input and a sequence of types is generated as output. DeepTyper uses a bidirectional recurrent neural network (biRNN) architecture in which two RNNs simultaneously traverse the input sequence both forward and backward. The goal is to capture large amounts of context to increase scalability. This strategy is an improvement over prior work aimed at the TypeScript language. However, it suffers from a phenomenon the authors call type drift: type predictions for the same variable may be inconsistent over multiple instances, even if the variable's type is fixed.

NL2Type [20] utilizes Long Short-Term Memory (LSTM) recurrent neural network architecture to address type inference as a classification problem for JavaScript. The standout design choice of this framework is the inclusion of extensive NL components in the source code, including variable and parameter names, as well as comments. The goal is to predict function return types. The appropriation of additional NL information is an effective improvement over previous work, which the authors attribute to the latent underlying connection between source code properties and documentation. However, this approach is still limited in practice by a fixed vocabulary of only a thousand types.

TypeWriter [27] extends the deep learning formulation introduced by prior work with a novel combinatorial search component that seeks to effectively find type-correct annotations for Python function return types and formal parameters. TypeWriter first employs both natural language (comments, identifier names) and contextual code (parameter usage statements) extracted from the Abstract Syntax Tree (AST) representation of source code to train a neural type predictor based on a hierarchical RNN (hRNN) with two LSTM cells. Once trained, this model can produce a ranked list of type predictions for a target type annotation. With a set of missing type locations and their corresponding ranked predictions, TypeWriter commences a feedback-guided search procedure to eliminate possible inconsistencies that may arise when naively assigning each location independently its highest ranked prediction. To this end, a gradual type checker is used to guide a combinatorial search space of all possible type assignments.

LambdaNet [32] proposes an alternative representation of source code, that more effectively leverages type-centric structure. The input program is transformed into a type dependency graph representation, where each variable type is encoded as a node, and relations between types are modeled as labeled edges. The authors used a graph neural network (GNN) to generate an embedding for each node, which is used for type prediction. A key element of the graph representation resides in the semantics of edges: each edge represents either a logical or a contextual relation between the connected variables. Logical edges impose hard constraints (such as asserting that certain types must be functions, booleans, or objects), while contextual edges encode helpful information extracted from adjacent code context. Finally, a pointer network computes a distribution over type assignments based on the embedding computed by the GNN.

OptTyper [23] merges logic and learning-based techniques to produce type-correct annotations for TypeScript. The usage of logic constraints in OptTyper is based on a static analysis constraint generator that calls the TypeScript compiler for a target annotation and then constraints the possible type space for that target to the disjunction of the types emitted by the compiler. This contrasts the LambdaNet approach, which is centered around learning constraints, rather than imposing them on the output. OptTyper combines the logical constraints with an LSTM-based neural prediction in a continuous relaxation composite optimization framework. This framework increasingly relaxes the logical constraints derived from static analysis and combines them with the natural constraints generated by the neural network to generate a single objective function.

CodeBERT [8] is a pre-trained bimodal model for programming languages and natural language pairs (both PL-NL and NL-PL) trained on multiple languages. GraphCodeBERT [11] is an extension of CodeBERT which considers the inherent structure of code by using the data flow graph of a program. These models are generally trained on finding the semantic relations between code and language, which they can use to infer types. Both are multi-layer bidirectional transformer models which are used as baselines for the CodeXGlue benchmark [15]. GraphCodeBERT outperforms all compared models in this benchmark.

PLBART [1] is a pre-trained sequence-to-sequence model that uses programming languages and their associated natural languages text including StackOverflow posts. The authors find that their model outperforms SOTA models for most code summarization, generation, translation, and classification tasks. They also establish PLBART's effectiveness on program understanding and show that it learns crucial program characteristics.

TypeBERT [16] is a fine-tuned model based on BERT for the type inference task with a final type classification layer. It uses simple token-sequence embedding with a large dataset and finds that TypeBERT outperforms SOTA models by inherently learning the structure of the code if the dataset trained on is large enough. The authors compose their own dataset to fulfill this data requirement.

2.2 Similarity Learning-based Approaches

Typilius [2] introduces a meta-learning strategy for type inference in Python, in which the model learns a relaxed d-dimensional continuous representation of the search domain, called the type space. The key advantage of this technique is that the type vocabulary, in this case, the positions of individual data points in the type space, can be effectively amended with novel type embeddings. This enables the model to flexibly adapt to new data, in essence allowing for an unlimited (or *open*) type vocabulary. At prediction time, a point is embedded in this type space and its nearest neighbors are used to determine the probability distribution over types, based on distance. This approach is widely known as Deep Similarity Learning. To realize this, the authors propose a gated graph neural network (GGNN) architecture, trained using a triplet loss function, which simultaneously discriminates between similar and dissimilar types. The network inputs contain rich semantic and syntactic information extracted from raw tokens of the program, nodes of the syntax tree, and unique symbols. The type space is represented as a map of a sample of embeddings to which new points are mapped before performing k-nearest neighbor (KNN) search.

HiTyper [26] seeks to bridge the gap between static inference- and deep learning-based strategies by introducing a framework that both guarantees type correctness of type predictions and mitigates the inaccurate predictions of rare types that prior work suffers from. The algorithm first infers a type dependency graph (TDG) based on the input program, which it then uses in alternating a two-phase process. HiTyper initially commences the *forward type inference* stage by filling in the partially-inferred TDG node using static analysis, which it prioritizes over learned annotations. For nodes that static analysis fails to fill, HiTyper uses a similarity-based neural recommender instead. In the second phase, the algorithm engages a *backward type rejection* solution that validates neural recommendations. This two-phase procedure repeats for nodes whose inferred types were rejected in the latter phase until all nodes are filled.

Type4Py [21] focuses on the type inference task for Python. The model is a deep similarity learning-based (DSL) hierarchical network, which learns to discriminate between similar and dissimilar types by mapping the models' output to a type cluster. This type

Table 2.1: Comparative Analysis of Prior Work

Model	Year of Publication	Architecture Basis	Meta-Approach	Type Vocabulary Size	Unseen Type Support
JSNice [28]	2015	CRF	Dependency Networks	-	✗
Xu et al. [33]	2016	Graphical Model	Belief Propagation	-	✗
DeepTyper [12]	2018	biRNN	Classification	11,830	✗
NL2Type [20]	2019	LSTM	Classification	1,000	✗
TypeWriter [27]	2020	3 × RNN	Classification	1.000	✗
LambdaNet [32]	2020	GNN	Pointer Network	100	✓
OptTyper [23]	2020	LSTM	Constraint Satisfaction	100	✗
Typilus [2]	2020	GNN	DSL	Unbounded	✓
TypeBERT [16]	2021	Transformer	Classification	40,001	✗
HiTyper [26]	2021	NN (unspecified)	DSL	-	✗
Type4Py [21]	2021	2 × RNN	DSL	Unbounded	✗
DiverseTyper [14]	2022	Transformer	DSL	Unbounded	✓
TypeSpaceBert (ours)	2022	Transformer	DSL	Unbounded	✗[*]
Baseline (ours)	2022	Transformer	Classificaiton	50,000	✗

[*] Unseen types can be amended to the search space in a similar manner to the technique used in Typilus [2]. This is not supported in this research prototype.

cluster allows an unbounded type vocabulary. However, the authors acknowledge that it is still restricted by the actual vocabulary of the training dataset. The authors created a dataset ManyTypes4Python [22], which they statically enhanced with more type annotations using Pyre. This increases the size of the annotated dataset, which increases the performance of the model. From this dataset, the authors retrieve the natural information of identifiers, function code context, and visible type hints, which builds a type dependency graph recursively based on imports. Lastly, the authors argue for the use of Mean Reciprocal Rank (MRR) as an evaluation metric, since developers tend to only use the first suggestion from an inference tool [24]. They report that Type4Py outperforms SOTA type inference models for Python for common types.

DiverseTyper [14] focuses on predicting user-defined types by fine-tuning TypeBERT. It alleviates the problem of only predicting user-defined types that exist in the training set by using the pre-training basis, which allows the model to predict a more diverse set of types. The model has a separate common types classification layer to produce a single-purpose type space from user-defined embedding. The resulting two lists of type classifications and their probabilities (common and user-defined) are arbitrated into a final type prediction. The authors show that their model outperforms SOTA models for user-defined types and is able to generate user-defined types that were unseen in the training set.

2.3 Model benchmark

CodeXGLUE [19] stands for General Language Understanding Evaluation benchmark for code. This benchmark provides multiple datasets and leaderboards that rank different models per type of code automation task. It provides three baseline models which allow for comparisons. CodeXGlue is extended with a type inference task and now contains the ManyTypes4TypeScript dataset [15]. This dataset consists of JavaScript code with an annotated type list. For this task, the baselines are compared with the measures of Top-100 and overall scores and use the precision, recall, F1-score, and accuracy metrics.

3 APPROACH

To build an effective type prediction algorithm for TypeScript code with an unbounded number of possible types, we use the following approach. First, we run the code through CodeBERT, where we fine-tune the model in a Deep Similarity Learning (DSL) setting. The model receives triplets of tokens and labels during training, which it casts in a low-dimensional type space fit for clustering analysis. After training, the type space is instantiated by recording the positions of all points in the data set, according to the model's output. Lastly, k-nearest neighbors search is performed in the populated type space at inference time to determine the type of a targeted token.

This section describes the intuition, design, and implementation details of our approach. We first explain the baseline architecture and its features. Next, we introduce our novel DSL-based extension. We further detail our fine-tuning pipeline, and the inference process carried out at prediction time. Finally, we provide a visual example of a type space to help build intuition.

3.1 Baseline architecture

We implement our approach as an extension of CodeBERT, a bimodal pre-trained model that captures semantic connections between NL and PL. CodeBERT supports both understanding and generation tasks [8]. It is pre-trained on a large, multilingual source code corpus, utilizing both bimodal (tuples of code and corresponding function-level documentation) and unimodal data points. The pre-training process follows a hybrid objective function setup that includes both Masked Language Modeling (MLM) [7] and Replaced Token Detection (RTD) [5]. MLM leverages bimodal data by randomly masking portions of both the NL and PL tokens input tuples, with the goal of recovering the hidden tokens. RTD operates on both unimodal and bimodal data points in a binary classification setup that aims to discriminate between "real" and "fake" input by generatively corrupting data at random points in the input [8].

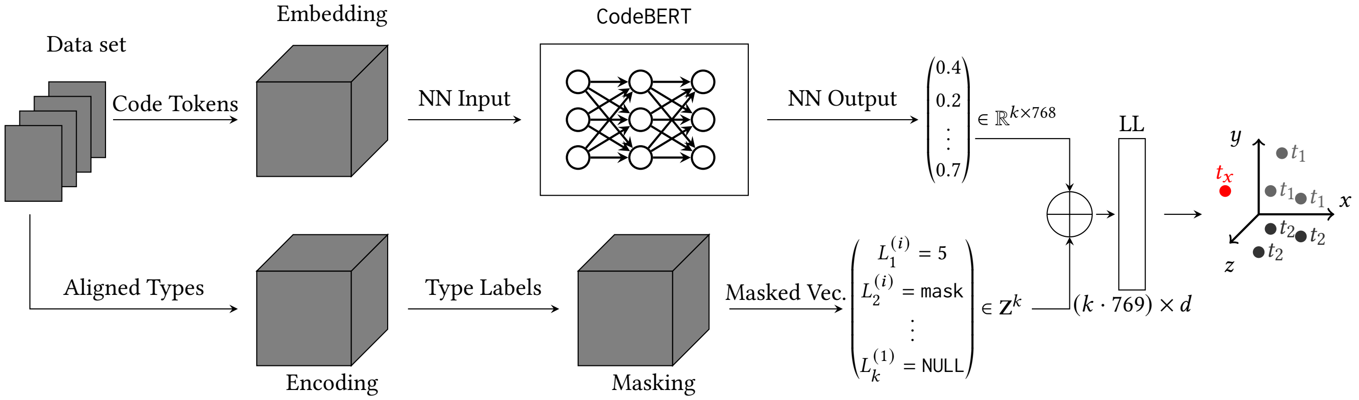

Figure 3.1: Simplified visualization of the neural architecture of our model.

CodeBERT uses a similar bidirectional Transformer [31] neural architecture as its predecessors BERT [7] and RoBERTa [18], specifically following the same architecture as RoBERTa$_{base}$ [8]. CodeBERT was trained on a corpus consisting of 6.4M unimodal and 2.1M bimodal datapoints across 6 languages, and is available in several language-specific fine-tuned versions [8]. CodeBERT is shown to significantly outperform its purely NL-trained counterparts in both NL code search and documentation generation tasks across all six programming languages. This highlights not only the effectiveness of the Transformer-based architecture for code-related tasks but also the powerful combination of multilingual training and language-specific fine-tuning.

CodeBERT and GraphCodeBERT [11] display SOTA results for the task of type prediction [15]. In our approach, we seek to further improve the performance of CodeBERT by enriching its output with a similarity-based clustering technique and fine-tuning the model for the type inference task. Utilizing the pre-trained weights of CodeBERT allows the amortization of the overall training time and only requires fine-tuning the extended architecture on the task of type prediction. Because CodeBERT is an encoder model, and our extension simply casts the encoding into a lower-dimensional space, extending the number of learned types is possible. [14]. Users can effectively amend the type space with additional points (i.e., from local codebases) whose coordinates in the type space are recorded alongside their labels.

3.2 Type space extension

We extend CodeBERT with a fully connected linear layer which casts the type inference problem into a similarity learning framework. More concretely, we map type prediction inputs to outputs in a multidimensional Euclidean space, where the model learns to discriminate between similar and dissimilar types based on distance. Formally, this is an instance of a DSL task [4]. Similar approaches have been explored in literature, with the output space referred to as *type clusters* [21] or *type space* [2].

The DSL framework offers several crucial advantages in comparison to a classification-based approach for type prediction. First, in a classification setting (i.e., [20], [27]), a typical architectural setup consists of one neural output head for each type in a given vocabulary. At prediction time, the user runs the model forward, and the result usually takes the form of a probability distribution constructed using the softmax function over the values of the output layer. While this strategy is effective for situations with an enforced pre-defined vocabulary, it fails to scale up to arbitrary vocabulary sizes, and predictions that do not fit any known type are assigned a reserved UNK label. To mitigate this issue, classification-centric neural architectures for type prediction usually rely on large vocabularies (in the range of thousands and tens of thousands) derived from the pre-training data.

Second, classification models are ineffective at adapting to changes in vocabulary. Since the vocabulary size dictates the architecture, changing the vocabulary requires a mirroring adaptation in the model to accommodate new types. DSL addresses both of those limitations. By building a Euclidean space that can contain an arbitrary number of clusters, DSL eliminates the scalability concerns. Further, online predictions on new data enable amending the type space with new clusters for previously unseen types, thus eliminating the pre-set size dependency problem. Note, however, that this does not inherently imply that DSL allows for unseen types to be synthesized from their mapping in the cluster space (i.e., the position of the mapping only determines the distance previously "seen" types).

The novelty of our approach comes from enriching the pre-trained architecture of CodeBERT with a fine-tuned type space mapping layer. Unlike Type4Py [21], and Typilus [2], which also use a type spaces with unbounded vocabularies, our approach is based on a Transformer model rather than RNNs and GNNs, respectively. Our approach also differs from DiverseTyper [14], which employs TypeBERT [16], a Transformer model similar to CodeBERT, and type spaces, in that our approach exploits the multilingual pre-training of CodeBERT in conjunction with language-specific fine-tuning, as opposed to a task- and language-specific pre-training procedure.

We provide a simplified visual representation of this architecture in Figure 3.1. We provide further details on the techniques used to process labels in Section 3.3 and explore an intuitive example of a fully assembled type space in Section 3.5.

3.3 Fine-tuning

We reduce the complexity of the training procedure by re-utilizing the base weights of CodeBERT. The training process consists of a single end-to-end fine-tuning step, in which we sample the training dataset (we provide more details about the dataset in Section 4.1) for aligned token-type sequence pairs to learn a spacial transformation of the input pair into the target Euclidean space. We do this by composing triplets of data points $\langle x_a, x_p, x_n \rangle$ where each member of the triplet is composed by (1) a contextual sequence of source code tokens that contains a target whose type the model should predict, and (2) a partially masked label vector of matching size, which contains the corresponding type of each token. Untyped tokens, such as syntactic elements or keywords are assigned a special NULL type, which is ignored during training.

To create data points for training, we perform a series of operations on the dataset. Starting from the raw data, in which each document is formatted as two aligned lists T and L that contain sequence-aligned tokens and labels respectively, we first break the two lists into equally sized segments. That is, for the list of tokens T, we generate i sublists such that each sublist $T^{(i)}$ is created according to Equation 3.1, ith k the fixed size of the selected segments.

$$T^{(i)} := \left\{ T_{(i-1) \cdot k : i \cdot k}, \forall i \in \left\{ 1, ..., \left\lfloor \frac{|\, T \,|}{k} \right\rfloor \right\} \right\} \tag{3.1}$$

The creation of labels lists $L^{(i)}$ is analogous and preserves alignment. If the last segments of the lists do not exactly match the size k, we simply discard them, as to not introduce additional noise to the model by altering the data or padding it. Finally, each sublist tuple $\langle T^{(i)}, L^{(i)} \rangle$ generates l additional tuples $\langle T^{(i)}, L^{(j)} \rangle$, for each $L^{(j)}$ belonging to the set described in Equation 3.2.

$$\left\{ \left[\begin{cases} L_n^{(i)}, \text{if } n \neq m, \\ \texttt{mask}, \text{if } n = m \end{cases} \right]_{n=1}^{k} \forall m \in \{0, ..., |L^{(i)}|\} \mid L_m^{(i)} \neq \texttt{NULL} \right\} \tag{3.2}$$

Prosaically, for each partitioned label list, we generate h offspring lists, with h the number of non-NULL type labels. Each of the offspring lists is identical to the original, except for the introduction of a special mask at one of the m types' positions. Each of the generated masked label lists is associated with the same original token list, which allows for an independent inference procedure to be carried out for each type in the token sequence. We repeat this procedure for all documents in the training set.

The training algorithm samples the triplet dataset to select three distinct list tuples, which are forwarded to the model to obtain $\langle t_a, t_p, t_n \rangle$. We use these values to compute the error according to the triplet loss formula provided in Equation 3.3 (where m is a pre-defined margin):

$$\mathcal{L}(\langle t_a, t_p, t_n \rangle, m) := max\{0, m + ||t_a - t_p||^2 - ||t_a - t_n||^2\} \tag{3.3}$$

Intuitively, this function reaches its optimum when the model minimizes the difference between the anchor (or reference) point t_a and a matching (positive) example t_p, while simultaneously maximizing the difference between the anchor and a non-matching (negative) example t_n. This loss function lends itself naturally to the task of similarity learning because of its tendency to concurrently incentivize both desirable properties of clustering.

3.4 Type space inference

After the fine-tuning phase, the type space is built by running the model forward on a selected subset of inputs from the training dataset, and the output is recorded alongside each label. In practice, the records are pairs of d-dimensional type space mappings and the corresponding numeric representation of the masked type. For our implementation, we ensured that the type space is built homogeneously, meaning that the same network weights are used for each prediction. We do this to encourage consistency in the type space: as hyperparameters are learned during training, the shape of the type space and the position of data points in it is altered. To alleviate problems that might emerge from keeping data points that have been mapped with older parameters, the entire type space is built using the fully trained version of the model.

At prediction time, the model produces a d-dimensional vector x that corresponds to the position of type t in the type space. To determine the probability distribution at this point, a KNN [6] heuristic is employed. Formally, this heuristic expands a minimal hyperspherical volume in the euclidean type space centered at x until a set of size least k of other points $Y = \{y_{i \in \{1..n\}} \in \mathbb{R}^d, n \geq k\}$ is contained within. Each point y_i is the type space mapping of a corresponding type t_j. Let $T = \{t \mid \exists y_i \in Y, \texttt{model}(t) = y_i\}$ be the set of types that map onto the set of nearest neighbors. The probability distribution is defined in terms of this set T according to Equation 3.4, where N is a normalizing constant, and $d(y_t, y_i)$ measures the d-dimensional euclidean distance between the y_t and y_i.

$$p(t = t_j \in T) = \frac{1}{N} \sum_{y_i \in Y | \texttt{model}(t_j) = y_i} \frac{1}{d(y_t, y_i) + \epsilon} \tag{3.4}$$

Intuitively, this computation measures probabilities as inversely proportional to the distance between the point y_t and the selected neighbor y_i. If the same type is mapped to multiple points y_i, the probabilities are simply added up across all common mappings.

For the practical implementation of the generation of the type space and the queries using the KNN heuristic, we use the Annoy library [3].

3.5 Example

To build intuition on the mechanisms employed in the type space, we provide a visual example in Figure 3.2. Individual types are represented as colored spheres, whereas two spheres that share the same colors are representations of the same type. Types can form homogeneous clusters, as in the case of the grey spheres, or overlapping clusters, as in the case of the mixed blue and yellow sphere clusters. Types can also be cast far away from any cluster if they are dissimilar and rare, as in the case of the orange sphere. The red sphere is selected as an example of the KNN clustering procedure: in this case, the red volume encompasses 7 other spheres which all share the same type: as a result, the final prediction would be the grey type, with a probability of 1.

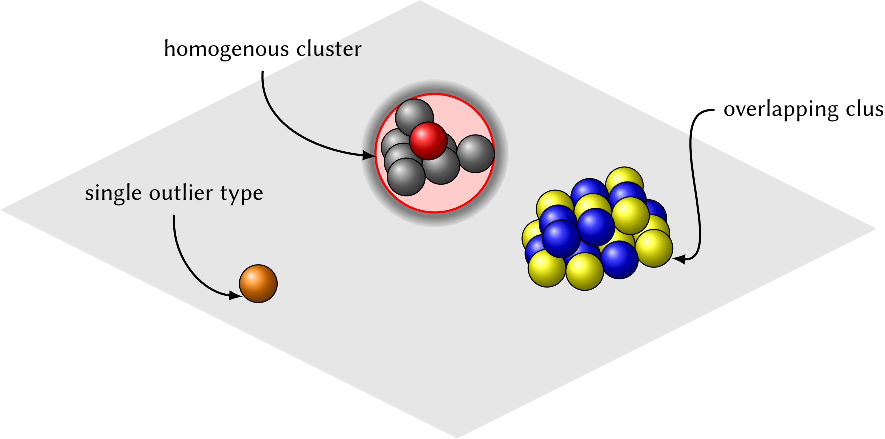

Figure 3.2: A sample type space mapping.

Table 4.1: CodeXGlue dataset properties

Type	Training	Testing	Validation
Projects	11.4K (81.8%)	1.3K (9.58%)	1.2K (8.62%)
Files	486K (90.16%)	28K (5.20%)	25K (4.64%)
Sequences	1.7M (91.95%)	81K (4.34%)	69K (3.71%)
Types	8.6M (95.33%)	224K(2.46%)	201K (2.21%)

4 EVALUATION SETUP

This section covers the core aspects of our empirical study and details the rationale behind the design choices we made.

4.1 Dataset

To align our analyses with literature standards for evaluating code intelligence tasks, we choose the `ManyTypes4TypeScript` [15] dataset distributed through the `CodeXGLUE` challenge [19]. The dataset is specifically aimed at TypeScript type prediction and was originally released in 2022. It contains over 9 million type annotations across 13,953 projects and 539,571 files. The dataset is split into three partitions for training, validation, and testing, each containing roughly 80%, 10%, and 10% of the data, respectively. We provide a detailed breakdown of the dataset composition in Table 4.1.

We pre-process the data to filter out all fields except for tokens and their labels (types). Tokens are then split into batches of a pre-defined size and provided as inputs to `CodeBERT`. Aligned types are split correspondingly and given as input to the linear layer, following the masking procedure described in Section 3.3.

4.2 Baseline

To meaningfully assess the performance of our model, we seek to choose a fair baseline algorithm for evaluation purposes. Due to the resource limitations, this work adheres to, comparing our model against SOTA solutions does not provide a fair comparison because our model has not been trained to the full extent of the dataset. Instead, we chose to design a separate model that shares the same architectural traits described in Section 3, but instead employs a classification-based approach, as opposed to DSL. This baseline allows us to accurately gauge the strengths and weaknesses of DSL within the `CodeBERT` framework.

4.3 Implementation

`TypeSpaceBERT` *implementation.* To implement our approach, we use the pre-trained `CodeBERT-base` model from huggingface[1] and augment it with linear layer implemented in `PyTorch`[2]. We use a similar triplet sampling strategy as the one employed in `Type4Py`[3], where we define a positive t_p with respect to an anchor t_a as any instance of the same type, that is, $t_a = t_p$ (see Section 3.3 for details). Negative examples are sampled at random from the remaining data points of the corpus (i.e., any type $t_n \neq t_a$).

Baseline implementation. Our baseline implementation follows the same architecture and modules as `TypeSpaceBERT`, except for the dimension of the output linear layer. In `TypeSpaceBERT`, the user can define the output dimension of the model by setting a parameter d. The output dimension has a pre-set value of 50.000, in accordance with the type vocabulary of the dataset. At prediction time, the type distribution is determined by computing the `softmax` function over the output neurons.

Experimentation environment. To fine-tune the model, we used the Google Cloud Platform with one NVIDIA TITAN 4 GPU for 48 hours. This resulted in a utilization of around 2% of the dataset. Training on the entire dataset is estimated to take between 960 and 1270 hours. The generation of the type space based on the fine-tuned model has run for 24 hours and used around 6% of the training data. Finally, the complete evaluation using the fine-tuned model and the constructed type space would have taken around 5 hours. Because of this, we sample 10.8% of the test data, to perform multiple intermediate evaluations that could capture the progress trend of the model.

Parameter Settings. For reproducibility purposes, we provide a full list of parameter values used for experimentation in Table 4.2.

[1] https://huggingface.co/microsoft/codebert-base
[2] https://pytorch.org/
[3] https://github.com/saltudelft/type4py

Table 4.2: Experimental Parameters

Parameter	TypeSpaceBERT	Baseline
Base Model	CodeBERT	CodeBERT
Context Window Size	128	8
LL Dimension	98,432 ×8	6, 272×50,000
Type Space Dimension	8	-
KNN k	10	-
Loss Function	Triplet Loss	Cross-Entropy Loss

We chose the context window sizes empirically based on samples of training batches such that the models perform comparably in terms of speed. We also set the number of nearest neighbors k of the inference procedure of TypeSpaceBERT to 10, as this is a common value in literature [2, 21]. To train the models, we used the Adam optimizer [17] with a learning rate of 0.01 and an ϵ value of 10^{-8}.

4.4 Metrics

Prior work has established a wide range of evaluation metrics to capture the performance of type appraise algorithms. Top-X metrics measure the percentage of times the correct type was included in the X most likely candidates, as predicted by the algorithm. This metric is remarkably flexible, and variations in literature range from Top-1 (also referred to as exact-match), which might be most meaningful for developers [24] to Top-100, depending on the scope of the experiment. The Mean Reciprocal Rank (MRR) is an alternative to Top-X metrics that rewards matches between better-ranked recommendations and ground truth. For our evaluation, we compute the Top-1, Top-8, and MRR@8 measures. 8 is heuristically chosen for analysis, as we found it is the largest value for which included types are meaningful.

Standard statistical metrics such as precision, recall, F1-score, and accuracy are also commonplace in literature. We focus on exact match accuracy for our evaluation, as no clear definitions for true and false negatives have been explored in prior literature, rendering these metrics unreliable. We further note that since the type vocabulary of our dataset does not contain nested types (such as Array<number>), the Base Exact Match (BEM) metric is not of concern.

Literature distinguishes between different categories of types based on frequency and composition. Often, types are classified into ubiquitous, common, and rare types based on frequency; and base, user-defined, and nested types based on composition. Due to complexity and time constraints, this distinction is not analyzed for this research and is instead recommended for future work to give a broader evaluation of our results.

5 EVALUATION

This section presents the evaluation of the performance of our model. To assess the effectiveness of TypeSpaceBERT, we focus on evaluating the following research questions:

RQ1. *How effective is* TypeSpaceBERT *at the task of type prediction?*
RQ2. *How do the size and training time of* TypeSpaceBERT *scale?*

5.1 Type Prediction Effectiveness (RQ1)

Table 5.1 displays the Top-1, Top-8, and MRR@8 scores of our model and the classification baseline model over 41,000 training data samples. In Figure 5.1, Figure 5.2, and Figure 5.3, we show the performance of TypeSpaceBERT plotted over the number of training examples the model used.

The results in Table 5.1 indicate that TypeSpaceBERT outperforms the baseline by factors of 3.78 and 1.66 for Top-8 and MRR@8, respectively, suggesting the superiority of our model in environments in which the amount training data is limited. The Top-8 results shown in Figure 5.1 remain constant over all intermediate models compared. We hypothesise that the cause of this is the scarcely populated type space, which was generated using only 6% of the train set. This sparse type space may contain a strongly skewed topology, possibly revolving around several prominent clusters which dominate their counterparts at prediction time. We expect that increasing the amount of data used to construct the type space would mitigate this problem and, in turn, produce more meaningful results.

In contrast, the Top-1 and MRR@8 results displayed in Figure 5.2 and Figure 5.3 show a more dynamic behavior over the different models. The Top-1 score ranges between 0.024 and 0.033, while the MRR@8 varies between 0.106 and 0.113. Moreover, the trend lines reveal that for both metrics, the performance is increasing with the amount of training data, which suggests the potential viability of the model in a more general scenario. We suspect that the erratic instability which occurs between 0 and 10,000 iterations is caused by the relatively small amount of data used for the fine-tuning procedure. Due to TypeSpaceBERT's inability to predict types which it has not seen during fine-tuning, a small training set directly hinders the model's performance on the test data. As such, we would expect that the stability of the performance metrics would increase with the size of the train set.

The amount of training data leveraged in this work is severely limited by the constraints of this study. This likely causes our approach to drastically under-perform, and therefore, the results are not necessarily a reflection of the model's true capabilities but rather an indication of its potential. Despite these limitations, TypeSpaceBERT manages to predict a small subset of types without extensive training and outperforms the classification baseline on the Top-8 and MRR@8 metrics, which lightly suggests effectiveness. DiverseTyper [14], an architecturally similar approach, has a more complex network and performs better, however, our model shows potential to utilize its own multi-lingual pre-trained basis to perform comparatively when fully trained. Nevertheless, drawing an exhaustive conclusion requires further research.

RQ1: TypeSpaceBERT has low exact match accuracy due to limited training data. However, it outperforms the classification baseline by a factor of 3.78 and 1.66 for Top-8 and MRR@8, respectively. Our model shows an increasing trend in accuracy, which suggests better performance when fully trained.

5.2 Size and Train Time Scalability (RQ2)

To compare the size of the models, we focus on the dimension of the appended linear layer (LL). In our implementation, the LL takes as

Table 5.1: Accuracy Results

Model	Top-1	Top-8	MRR@8
TypeSpaceBERT	0.032	0.257	0.113
baseline	-	0.0680	0.0681

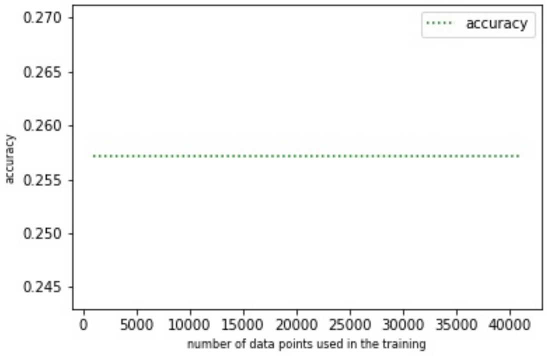

Figure 5.1: TypeSpaceBERT Accuracy Top-8

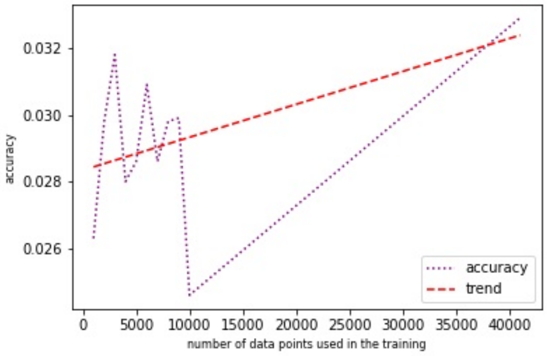

Figure 5.2: TypeSpaceBERT Accuracy Top-1 with trend line

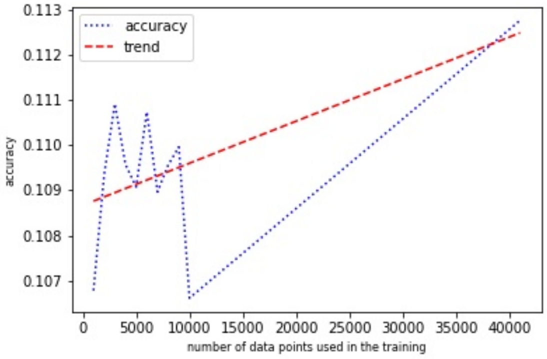

Figure 5.3: TypeSpaceBERT Accuracy MRR-8 with trend line

input the hidden state vector of CodeBERT over the embedded token input sequence. Given a window dimension k, the input of the LL is $k \cdot 768$, as per CodeBERT's architecture. This value is essential for scalability purposes because given a type space dimensionality d and a vocabulary size v, the LL of a classification model has an equal number of connections between input and output neurons compared to DSL counterpart iff $k_{\text{classification}} = \frac{k_{\text{DSL}}}{d} \cdot v$. In practice, this means the number of connections is expected to change by a factor of $\frac{v}{d}$ when changing from a DSL approach to a classification approach. The dataset used in our experiments contains a vocabulary of size 50,000, and given that DSL approaches generally employ much smaller spacial dimensions (8, in our case), we hypothesise that TypeSpaceBERT's scalability is strongly preferable terms of both size and training time in comparison to the baseline.

5.2.1 Size Scalability. We set the window size and type space dimensionality of TypeSpaceBERT to 128 and 8, respectively. To find an appropriate window size for our baseline, we repeatedly halved the window size of the classification model until we obtained comparable results on small-scale datasets. We settled on a window size of 8 as it was the largest window we could obtain without compromising either the training procedure due to memory limitations or the inference performance due to an overly narrowed context. We provide comparative measurements for both cached model sizes and estimated memory occupation during training in Table 5.2.

The results indicate that our DSL approach scales much better than the baseline. The average disc memory requirement for storing a partially trained TypeSpaceBERT model is 478.49 MB, compared to 1.64 GB for the baseline. This shows a decrease by a factor of 3.44 and is consistent with our expectations, given the much higher number of neural connections in the LL (ca. $313 \cdot 10^6$ for the baseline compared to ca. $787 \cdot 10^3$ for the DSL approach). The smaller memory requirements make our model easier to deploy in practice and provide more flexibility with regard to window size. For the baseline to integrate a context window size as large as TypeSpaceBERT, the output layer would have to be 4 orders of magnitude larger, making training cumbersome and resource intensive.

5.2.2 Train Time Scalability. We measure train time performance in terms of the mean number of training iterations per hour performed by each model. The comparison employs the same data for both models, however, loss functions are different: TypeSpaceBERT is trained via the triplet loss function, which requires three samples to be run through the model at each step, while the baseline is trained using the cross-entropy loss, which only requires one data point. We provide approximate results in Table 5.2.

On average, TypeSpaceBERT performs 3,167 iterations per hour, compared to the baseline's 200. This implies that despite requiring three times as many samples to train at each step, the whole training pipeline for TypeSpaceBERT would complete 15.8 times faster than the classification-based alternative. We attribute this impactful difference to the heavy computational constraints imposed by the introduction of the 50,000 output neurons in the baseline. We note that the training procedure for TypeSpaceBERT also requires the creation of the type space by running the model forward one more time through the dataset. This requirement does, however,

Table 5.2: Scalability Results

Metric	TypeSpaceBERT	Baseline
Mean Size when Cached	478.59MB	1649.18MB
Estimated Mean Memory Occupation	11.9GB	33.1GB
Number of Iterations Per Hour	3,167	200

not massively impact the difference between the approaches, as in this latter step (i) does not require three different samples at each step, as in the training, and (ii) does not include any gradient or loss computation.

RQ2: Despite having a context window size 16 times larger than the baseline, sampling 3 distinct points at each iteration during training, and requiring the subsequent creation of a type space, TypeSpaceBERT scales significantly better than the baseline in terms of both memory requirements and train time.

6 THREATS TO VALIDITY

Threats to internal validity concern factors that might discredit or invalidate the credibility of our conclusions. Concretely, we faced two significant setbacks in this regard: (i) choosing a fair baseline to compare our findings against, and (ii) obtaining a sufficient number of samples to draw statistically robust conclusions about the features of our approach. We addressed the former by developing a "twin" architecture (see Section 4.3 for details) that shares the same data flow, but utilizes different output layers and inference procedures. This allows us to comparatively gauge the effectiveness of the DSL extension in a fair empirical manner. We addressed the latter concern by training the models as much as we could within the constraints of this work. To address both concerns, we recommend fully fine-tuning TypeSpaceBERT on the entire training dataset and comparing the results against current SOTA models. Statistical robustness can be attained by the computation of statistical tests over multiple instances (such as the Wilcoxon signed-rank test or the Vargha-Delaney effect size test [30]).

Threats to external validity concern factors that might negatively impact the generalizability of our approach. The quality of our fine-tuned model is heavily reliant on the fine-tuning data. The ManyTypes4TypeScript dataset (see Section 4.1 for details) helps mitigate this concern in two ways. First, the sheer size of the dataset (which ca. 10 times larger than an equivalent Python dataset [15]) helps generalize the learned patterns over a large corpus of projects and files, as demonstrated in TypeBERT [16]. Second, the breadth of the type vocabulary of 50,000 types is far larger than that of datasets used in many prior works. For instance, DeepTyper's vocabulary is limited to 1000 types [12]. Utilizing this dataset to its full potential would be the most effective way to increase the robustness of our model over unseen TypeScript data. However, due to the circumstances described in the previous paragraph, this was not possible, and generalizability remains a concern for our results. Finally, an ablation study that scrutinizes the individual effects of TypeSpaceBERT components (i.e., the type space mapping, the distance heuristic, and the KNN inference procedure) on the overall performance of the algorithm would help cement the extent of our contribution. The same limitations prevented us from carrying out this study.

Threats to reproducibility regard factors that might call into question the reliability of our study and analysis methods. To enable reproducibility, we (i) provide an open-source implementation of our model and baseline as a supplement to this work, (ii) provide extensive mathematical and intuitive explanations of our methods and the rationale behind our design choices that enable practitioners to understand and modify our algorithm (see Section 3), and (iii) we provide the values for all hyperparameters used in experimentation in Table 4.2 to enable convenient replication of our experiments.

7 DISCUSSION & FUTURE WORK

The main limitation of this study is the use of a small dataset, lack of training time. Due to a complex and lengthy development period, there was not enough time to fully fine-tune our extension. This means that the type space mapping is not fully optimized and might be skewed resulting in lower performance. In addition, having only trained the model on a small subset of available data means that there are likely many types on which the model has never been trained. This means that predicting those labels in practice is impossible without further training. Our recommendation, therefore, is to optimize the model with accelerators to reduce training time and fine-tune the model with the complete dataset for more accurate results. Better performance can also be attained by tuning the hyperparameters listed in Table 4.2 to better fit the data.

After inspection of the tokenized input sequence, we observed that the tokenizer was not fully trained on the type prediction dataset. This resulted in some variables that were parsed into multiple tokens. To alleviate this problem we would need to retrain the tokenizer and respectively the CodeBERT model, which time did not allow. However, the type label would still exist at the first token of the typed identifier, while the rest would be part of the code context. Therefore this should not have a significant impact on performance.

An idea for future work is based on an implementation of the model for developers to use as some sort of tool that could gather more specific examples and learn online to improve project-specific and general performance due to increased data. This could alleviate some of the problems of low-frequency occurring types and increase performance on them. Although our model should already be capable of synthesizing unseen types better due to the use of CodeBERT's general language understanding, the model still falls short in rare types. The flexible DSL architecture allows for the type space to be partially amended at any time, which could significantly enhance the accuracy of the model on unseen types.

7.1 Possible Extensions

The flexibility of our approach and implementation facilitates the extension of TypeSpaceBERT with several components that could better capture semantic information about the target program. This section explores 2 such extensions and provides guidelines and recommendations for implementation.

More precise code context. In our work, the tokens fed to the CodeBERT module consist of a fixed-size window that contains the target of the type inference task. The position of the target during training is not fixed, rather, the target can be at any position in the token sequence. We do this to amortize training and dataset processing time. However, this partly comes at the cost of accuracy. To improve accuracy, one can leverage the AST representation of the target program and extract only expressions that are relevant to the specific type prediction instance. Ablation studies from previous work have shown that enriching the model with AST information can be beneficial to prediction accuracy [2, 21]. An interesting research question in this regard is how such additional information can be used to lower the window size parameter of the model, while still retaining comparable performance (i.e., how much more valuable AST information is in comparison to syntactically close context). Note, however, that such an approach would require additional training for AST-related components.

Alternative inference heuristics. Our model uses KNN search at prediction time to determine the probability of a mapping belonging to each type in the Euclidean neighborhood. To the best of our knowledge, all previous work regarding DSL type prediction follows this heuristic. The main advantage of KNN is its ability to effectively extract information from the local topology of the space, however, it neglects possibly informative structural hierarchies that may emerge in the type space. To this end, we hypothesise that a hybrid approach, which combines KNN search and dendrograms (branching diagrams that capture relationships of similarity within data) might be an effective alternative. Such an inference algorithm would commence in two phases. First, KNN search selects the points within a hyperspherical subvolume around the predicted type mapping to express part of the hierarchy of the entire space. Second, a dendrogram is constructed using hierarchical agglomerative clustering (HAC) from those points until the insertion of the target mapping. The probability distribution over types is then determined by traversing the dendrogram structure downwards from the insertion of the target. Such an approach presents many design choices. How HAC proceeds, whether to continue the construction of the dendrogram past the insertion point to capture more detail and how to compute the probability distribution based on the aggregated points are all interesting design opportunities for such a method. We believe that this technique could provide an appealing trade-off between the exploitation of the local structure that KNN offers and the detailed distribution that aggregating all points in the search space allows for.

8 CONCLUSION

We introduce TypeSpaceBERT, a fine-tuned neural type prediction model that utilizes deep similarity learning to map aligned inputs of tokenized source code and corresponding types to create a Euclidean space populated by type mappings. TypeSpaceBERT uses CodeBERT's pre-trained weights and architecture for initialization and extends the base model with a linear layer that is specifically fine-tuned for the type prediction task. We fine-tune our model on a portion of the ManyTypes4TypeScript dataset and empirically compare its accuracy against a classification-based extension of CodeBERT, comparing their type prediction performance and scalability capacities. We find that with limited training, our approach shows a promising learning curve concerning the MRR@8 and Top-8 metrics, also significantly outperforming the baseline in these circumstances. In addition, we find that TypeSpaceBERT scales significantly better than the classification-based counterpart both in terms of model size and training time.

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] 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.

[3] Erik Bernhardsson. 2017. Annoy Approximate Nearest Neighbors Oh Yeah. https://github.com/spotify/annoy. Accessed on 25-10-2022.

[4] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 1. IEEE, 539–546.

[5] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).

[6] T. Cover and P. Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13, 1 (1967), 21–27. https://doi.org/10.1109/TIT.1967.1053964

[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, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Michael Gorbovitski, Yanhong A Liu, Scott D Stoller, Tom Rothamel, and Tuncay K Tekle. 2010. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*. 27–42.

[10] Łukasz Langa Guido van Rossum, Jukka Lehtosalo. 2014. PEP 484 – Type Hints. *Index of Python Enhancement Proposals* (2014).

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

[12] 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.

[13] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[14] Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. 2022. Learning To Predict User-Defined Types. *IEEE Transactions on Software Engineering* (2022).

[15] Kevin Jesse and Premkumar T Devanbu. 2022. ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference. (2022).

[16] 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.

[17] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[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] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[20] 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.

[21] Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*. 2241–2252.

[22] Amir M. Mir, Evaldas Latoškinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 585–589. https://doi.org/10.1109/MSR52588.2021.00079

[23] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. [n.d.]. Probabilistic Type Inference by Optimizing Logical and Natural Constraints. ([n. d.]).

[24] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[25] Zvonimir Pavlinovic. 2019. Leveraging Program Analysis for Type Inference. *Ph.D. Dissertation. New York University* (2019).

[26] 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).

[27] 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.

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

[29] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 944–953.

[30] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[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] 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.