Learning To Predict User-Defined Types

Kevin Jesse, Premkumar T. Devanbu, Anand Sawant

Abstract—TypeScript is a widely adopted gradual typed language where developers can optionally type variables, functions, parameters and more. Probabilistic type inference approaches with ML (machine learning) work well especially for commonly occurring types such as `boolean`, `number`, and `string`. TypeScript permits a wide range of types including developer defined class names and type interfaces. These developer defined types, termed *user-defined types*, can be written within the realm of language naming conventions. The set of user-defined types is boundless and existing bounded type guessing approaches are an imperfect solution. Existing works either under perform in user-defined types or ignore user-defined types altogether. This work leverages a BERT-style pre-trained model, with multi-task learning objectives, to learn how to type user-defined classes and interfaces. Thus we present DIVERSETYPER, a solution that explores the diverse set of user-defined types by uniquely aligning classes and interfaces declarations to the places in which they are used. DIVERSETYPER surpasses all existing works including those that model user-defined types.

Index Terms—Transfer Learning, Multi-Task Learning, Representation Learning, Type Inference

1 INTRODUCTION

GRADUAL typing is gaining popularity particularly in dynamically typed languages like JavaScript and Python. Typing and type-checking can find common kinds of data-misuse in programs by checking that variables, expressions, functions and modules are used in a consistent fashion. *Type systems* can verify the type safety of the program in different ways: most languages verify types either statically at compilation time, dynamically at run-time, or some combination of both. Developers have come to appreciate the benefits of type checking at run-time; benefits include faster prototyping and more flexible use of variables [1], [2]. These advantages come at a cost because the resulting program has less known type associations prior to running [3]. Running a program with fewer type verified variables and functions results in an increased probability of uncaught type errors [4], [5].

To address these concerns, gradual type systems [6], [7] were proposed; they provide developers an attractive balance between static and dynamic typing. Developers can gradually add type annotations to a program, as they see fit. TypeScript is a gradually typed version of the JavaScript programming language that is gaining traction. TypeScript can be used anywhere JavaScript is used because the type checker enforces type rules prior to *transpiling* into JavaScript. Thus, code bases can still run on the highly popular frameworks JavaScript runs on and enforce some type rules.

Unfortunately this approach also has disadvantages. The *optionality* of gradually typed languages is a double-edged sword wherein the convenience of not typing variables and functions may result in type errors that can be caught prior to deployment if properly labeled [4]. Consequently, researchers have been determined to develop tools that adequately help developers label types especially when it can prevents bugs.

The abundance of typed source code (in gradually typed languages) from repository sites like GitHub[1] enables researchers to use machine learning methodologies to infer types in dynamic languages [8], [9], [10], [11], [12], [13]. Advancements in neural networks are helpful for software engineering tasks, including type inference [9], [13], [14], [15]. These approaches adapt machine learning architectures as best as possible but neglect particular aspects of programming languages that are consequential to the problem. Type-inference is traditionally framed as a bounded classification task because of the natural alignment with fixed categorical classification losses in machine learning. However, types have unbounded vocabulary, as do variable and function names; so it is desirable to accommodate an *open* type vocabulary. Thus, modeling types with a bounded classification layer is overly restrictive; the model's performance is limited by an upper bound.

We approach type inference with an unbounded vocabulary very much in mind. We further hypothesize that user-defined type *declarations* contain important information that can help probabilistic machine learning methods to infer type annotations. Our implemented model, DIVERSETYPER, leverages two principles: *large-scale pre-training* and *deep similarity learning*.

The first principal idea, pre-training, is the practice of teaching models the form of languages by enforcing auto-encoding objectives like masked-language modeling [16]. Pre-trained models are ideal for efficiently encoding programming features like user-defined classes or type interfaces. The second principal idea, deep similarity learning, is used to align or associate two encodings, for example, a class declaration and the use of the declaration as a type. Our

- *Kevin Jesse is with the Department of Computer Science, University of California Davis, Davis, CA, 95616.
  E-mail: krjesse@ucdavis.edu*
- *Premkumar T. Devanbu is with the Department of Computer Science, University of California Davis, Davis, CA, 95616.
  E-mail: ptdevanbu@ucdavis.edu*
- *Anand Sawant is with the Department of Computer Science, University of California Davis, Davis, CA, 95616.
  E-mail: asawant@ucdavis.edu*

Manuscript received November 8, 2021; revised April 12, 2022; accepted May 26, 2022.

1. https://github.com

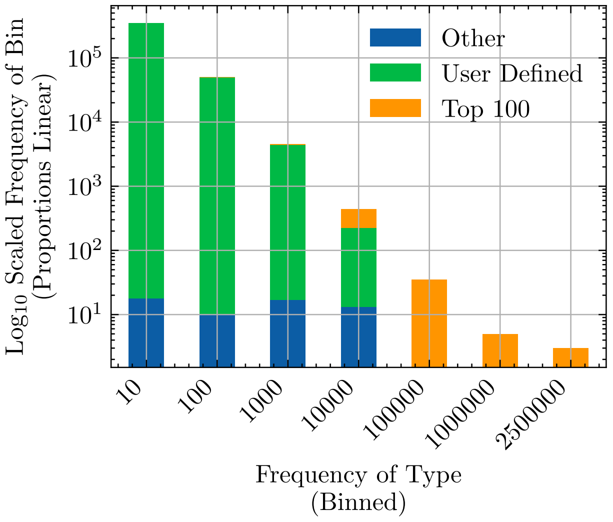


Fig. 1: A type is binned by how often it's used in code in the dataset (x-axis). The histogram of bins are scaled by Log$_{10}$ (y-axis) to see all bins. The ratios between Other, User-Defined, and Top 100 (color-coded) are linearly scaled for simplicity. For example, of the types that are used 10 times or less (first column), 77,820 are Other, and 266,882 are User-defined (22% 78%). On a log_{10} scale, the total (344,702) is between 10^5 and 10^6.

hypothesis is that such a relationship can be learned for an unbounded set of user-defined types, thus removing this artificial restriction that exists in previous methods.

In addition to user-defined types, it is common to use *native* and *library* types. Native types like `number` or `string` and library types like `ArrayBuffer` do not have declarations, and are very frequent [9], [12], [13], [15], [17], [18]. Thus the typing inference task has two seemingly orthogonal sub-tasks: learning common-types within a bounded vocabulary and aligning user-defined types to existing class and type declarations. We theorize that each task guides a neural model to learn divergent *type representations* which presents a challenge. A model learning a type representation for common-types in categorical form is equivalent to partitioning or folding embeddings across a fixed space. Additionally, a model learning a type representation that aligns declarations and annotations means clustering types into manifolds of separability. So we ask, can the model learn to selectively pick types that should be partitioned (common-types) versus clustered (user-defined types). The answer is yes! This is realized with a specially crafted training signal that balances the representation learning of common-types and user-defined types. The resulting type inference model, DIVERSETYPER, can predict common and library types, while also supporting *new* types defined using class and interface declarations. This model is also capable of predicting user-defined and rare types, even if the type definitions were not seen during training. DIVERSETYPER is effective globally across all types but especially in the most difficult user-defined types because it diverges from previous machine learning (ML) assumptions and aligns with how developers annotate custom types. This work's contributions are,

Contributions

1) A type inference model that adopts large scale pre-training to type-inference of common and user-defined types. DIVERSETYPER's adoption of pre-training helps it align new type declaration to uses of that declaration.
2) A novel multi-task uncertainty learning approach that combines type inference classification (cross entropy) and type similarity (semi-hard triplet loss) where loss scaling is learned end-to-end.
3) Improve type inference from state of the art approaches by 8.59% overall by improving user-defined type inference 30.16%. User-defined type inference is significantly harder than common-type inference due to its long-tailed distribution.

DIVERSETYPER can be found at our public GitHub[2].

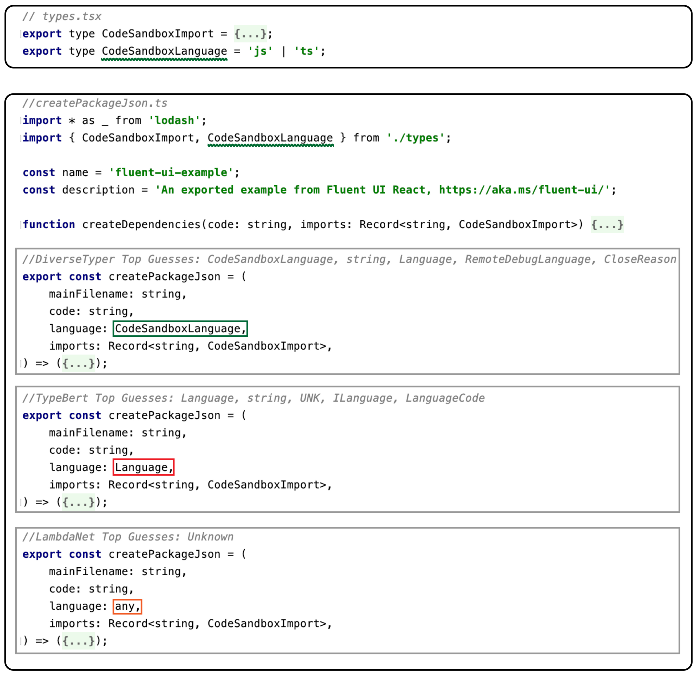
In the following sections we discuss the challenges of user defined types, the underlying intuition behind our approach, and why the approach of DIVERSETYPER is positioned better than previously developed approaches.

2 Challenges of User-Defined Types

Software programs introduce new vocabulary at a higher rate than natural language [19], due to new identifier names, functions, classes, enums, structs, *etc*. Types also feature large vocabularies, and thus (like variable, function names, *etc*) are a challenge for models with finite type vocabularies. Figure 1 shows that most unique types occur less than 10 times, typical of a long-tailed distribution. The figure shows the proportions of the top-100 types, user-defined types, and other (library) types; it's clear that most types constituting the long tail are in fact *user-defined* types. This is because user-defined types typically occur just within the scope of the project that defines them, and rarely exist elsewhere. It's evident from Figure 1 that type inference approaches that model a finite type vocabulary ignore a lot of types.

Close inspection user-defined type annotations (and their respective declarations) reveals that the declarations are often co-located in the same file or exist nearby (See Figure 2). In the example of Figure 2, representative of many user-defined type annotations, the compiler cannot deduce the corresponding type `CodeSandBoxLanguage` because of type ambiguity. This annotation ultimately resolves to a string and a variety of variables also are of string type. The task of correctly labeling types becomes challenging when types appear ambiguous to the compiler, say different type declarations with the same underlying string type. However, developers have a grounded common sense rooted in their experiences programming and familiarity with the language. A developer would observe the context around the type and gain familiarity with how the type should typically be used. The developer would likely see that the variable `createPackageJson` has an attribute `code` and `language` with a function `createDependencies` taking a code string, and record object that includes a `CodeSandbox` object. The developer would correlate that the words "dependencies" and "package" often requires certain keywords like "imports" in the context of JavaScript. Lastly,

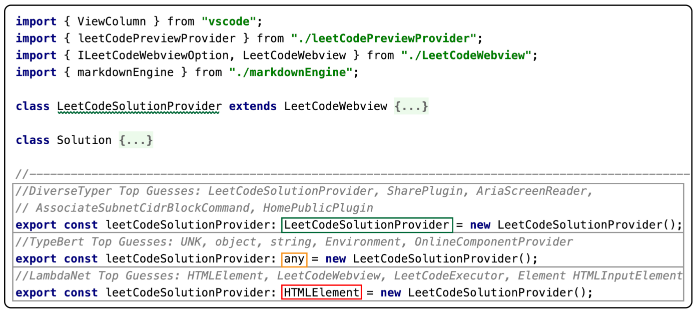
2. https://github.com/diversetyper/diversetyper



```
// types.tsx
export type CodeSandboxImport = {...};
export type CodeSandboxLanguage = 'js' | 'ts';
```

```
//createPackageJson.ts
import * as _ from 'lodash';
import { CodeSandboxImport, CodeSandboxLanguage } from './types';

const name = 'fluent-ui-example';
const description = 'An exported example from Fluent UI React, https://aka.ms/fluent-ui/';

function createDependencies(code: string, imports: Record<string, CodeSandboxImport>) {...}

//DiverseTyper Top Guesses: CodeSandboxLanguage, string, Language, RemoteDebugLanguage, CloseReason
export const createPackageJson = (
    mainFilename: string,
    code: string,
    language: CodeSandboxLanguage,
    imports: Record<string, CodeSandboxImport>,
) => ({...});

//TypeBert Top Guesses: Language, string, UNK, ILanguage, LanguageCode
export const createPackageJson = (
    mainFilename: string,
    code: string,
    language: Language,
    imports: Record<string, CodeSandboxImport>,
) => ({...});

//LambdaNet Top Guesses: Unknown
export const createPackageJson = (
    mainFilename: string,
    code: string,
    language: any,
    imports: Record<string, CodeSandboxImport>,
) => ({...});
```

Fig. 2: Code snippet from `microsoft/fluentui` GitHub repository. `CodeSandboxLanguage` is type defined in `types.tsx`. The type is also imported in `createPackageJson.ts`. However, both TypeBert and LambdaNet fail to properly annotate the correct user-defined class. DIVERSETYPER references the type properly despite the type `CodeSandboxLanguage` is used only once in all repositories, *viz.*, *rare* and *infrequent*.



```
import { ViewColumn } from "vscode";
import { leetCodePreviewProvider } from "./leetCodePreviewProvider";
import { ILeetCodeWebviewOption, LeetCodeWebview } from "./LeetCodeWebview";
import { markdownEngine } from "./markdownEngine";

class LeetCodeSolutionProvider extends LeetCodeWebview {...}

class Solution {...}

//------------------------------------------------------------------------------------------
//DiverseTyper Top Guesses: LeetCodeSolutionProvider, SharePlugin, AriaScreenReader,
// AssociateSubnetCidrBlockCommand, HomePublicPlugin
export const leetCodeSolutionProvider: LeetCodeSolutionProvider = new LeetCodeSolutionProvider();
//TypeBert Top Guesses: UNK, object, string, Environment, OnlineComponentProvider
export const leetCodeSolutionProvider: any = new LeetCodeSolutionProvider();
//LambdaNet Top Guesses: HTMLElement, LeetCodeWebview, LeetCodeExecutor, Element HTMLInputElement
export const leetCodeSolutionProvider: HTMLElement = new LeetCodeSolutionProvider();
```

Fig. 3: Code snippet from `LeetCode-OpenSource/vscode-leetcode` GitHub repository. The class `LeetCodeSolutionProvider` is declared in the same file that the class annotation exists in. Both TypeBert and LambdaNet do not properly type this annotation. The user-defined class exceeds the bounded type vocabulary of TypeBert so the best annotation it can do is `any`. LambdaNet seems to reference other LeetCode classes but annotates the class instance with `HTMLElement`. DIVERSETYPER gets the annotation correct.

any word of "sandbox" would allow the developer to narrow down the correct type even if other syntactically correct and semantically similar types exist; if `ProductionLanguage = 'js'` exists as another user-defined type, this would be syntactically correct . While no model has the same abilities to reason logically as a developer would with common sense knowledge, DIVERSETYPER, is designed to follow the same clues probabilistically which differs from previous approaches. In order to demonstrate the effectiveness of this approach over previous approaches and highlight our contribution across the user-defined type space, we will first discuss how the model encodes a type declaration, and then uses these powerful encodings to type variables in the main body of the code; no other approach does this, thus falling short across this diverse domain of types.

3 WHY OTHER TYPING MODELS FALL SHORT

The aim of DIVERSETYPER is try to reach a performance level closer to that of human developers. The model's understanding parallels developers by utilizing deep pretrained embeddings to encode a user-defined type; the pretraining practice is called Masked-Language-Modeling or MLM. This pretraining approach processes large swaths of raw source code available on GitHub [20], [21], [22], [23], [24] to learn representations (neural encodings) which capture common usage patterns. The model can use its neural encoding of source code to determine commonalities with other code tokens. With pretraining, the model in example Figure 2, will be guided to the words "language" and "sandbox" when guessing `CodeSandboxLanguage`. The words "language" and "sandbox" occur frequently in the context of the type `CodeSandboxLanguage` and not other types, which make these words highly indicative of this type. DIVERSETYPER digests any class or interface declaration and stores the embedding of the class or interface declaration as a type. DiverseTyper's novelty is that it can handle class and interface *declarations* as opposed to previous approaches that rely on learning from explicit type *annotations* already present in the code.

There are two popular approaches to recover types: (1) Models such as LambdaNet [17] will save the names of user-defined types and allow prediction to those names (using a pointer network). LambdaNet must determine the correct typing on very sparse occurrences of that type; as shown earlier, user-defined types are less frequent across a global set of projects. There is no pretrained embeddings involved so the parameterization of the model comes from sparsely learned co-occurrences of said infrequent types. (2) Other approaches, like Typilus and Type4Py, also do not benefit from pretraining and can only reference a user-defined type if appears as a previous *type annotation* (not *declaration*) in its training data. Our intuition is that models like Typilus and Type4Py cannot type as well as a human developer because it cannot observe new type declarations and understand nuances between such types without optimizing on them in a one-shot manner. To accommodate a new class declaration in Typilus and Type4Py, the model must rely either on previously seen type annotations of the *same exact type*, rather than directly computing an embedding from a declaration, and aligning that new embedding to valid type locations; this is what DIVERSETYPER does. When DIVERSETYPER digests class and interface declarations, it can use the pre-training basis to discriminate types by attributes and methods, thus deeming a type incompatible or compatible with the annotation location. With the type declarations at the disposal of the typing model, the model is able to reference

TABLE 1: Comparison between various learning-based type inference models

Model	Model Architecture	Type Vocabulary	User Definition Mechanism	Pre-Trained
DeepTyper [9]	biRNN	10,000	✗	✗
NL2Type [10]	LSTM	1,000	✗	✗
TypeWriter [11]	HNN	1,000	✗	✗
OptTyper [12]	LSTM	100	✗	✗
LambdaNet [17]	GNN	Unbounded	✗	✗
Typilus [18]	GNN	Unbounded	✗	✗
Type4Py [15]	HNN	Unbounded	✗	✗
TypeBert [13]	Transformer	40,000	✗	✓
DiverseTyper	Transformer	Unbounded	✓	✓

a more diverse set of types and an improved performance is expected.

We are not aware of any existing approach that achieves our levels of performance for such a diverse set of types. Earlier works like DeepTyper [9] and JSNice [8], use a limited type vocabulary and focus only on common-types. More recent Python approaches Typilus [18] and Type4Py [15], expand the type vocabulary to include all types seen in training. This expansion, to all types seen in training, is an improvement; but even these approaches face a performance ceiling on new types. A TypeScript approach called LambdaNet [17], expands the typeset to all visible project types with a scoring mechanism; this approach most in line with our proposal, and we do a careful comparison. A more recent approach, TypeBert, does not increase the type vocabulary, but demonstrates that BERT-style pre-training on JavaScript corpora boosts type prediction because the model learns token co-occurrence statistics relevant to typing. TypeBert, like other fixed type vocabulary models [8], [9], [11], [12], ignores new types. If developers were to use these tools in practice, the models will underperform on new types. Since newly defined types are common to projects per Figure 1, and are often key to good software design, our machine learning architecture is better-aligned to modern software development paradigms. Whenever a developer defines a new, project-specific class or type, DIVERSETYPER also encodes these classes and type interfaces with *pre-trained* vectors. DIVERSETYPER employs those representations in the type suggestion process. We explain in the following section how the pre-trained vectors improve a model's ability to capture the learnable and relevant features in code, and the benefits for machine learning based type-inference approaches.

4 DIVERSETYPER

In this section, we first introduce general pre-training for types, the training elements of DIVERSETYPER, followed by the inference mechanisms DIVERSETYPER.

4.1 Pre-training For Types

Pre-trained transformer models for code such as CodeBert [20], CuBert [23], PLBart [22], and TypeBert [13] achieve state-of-the-art (SOTA) results on code-related tasks by pre-training on large code corpora followed by fine-tuning weights on a specific task. Pre-training on large corpora is compute-intensive, which is often performed at large, resource-rich organizations that can afford the cost of training [25], [26]. Our work amortizes the expensive cost of pre-training by initializing DIVERSETYPER core weights with the pre-trained weights from TypeBert [13]. The pre-trained model takes in a sequence and outputs a contextual vector for each input token. The code token's context determines the vector. The complete body of a class or type declaration provides rich information such as attributes and internal functions. This rich information can guide a type inference model to link the uses of the class or interface to its declaration; this approach is novel in this work.

To take advantage of these useful representations, DIVERSETYPER is initialized with the published TypeBert weights, input width (256), and sub-tokenized input vocabulary trained with SentencePiece [27]. Tokenizing the code, with Byte Pair Encoding (BPE) [28] or unigram language modeling [27] is common to manage large input code vocabularies [19]. Tokenizing is not used in the type vocabulary because the model needs to output valid types. With the pre-trained weights and tokenized input inherited from TypeBert [13], we are ready to define training procedure for DIVERSETYPER.

4.2 Training

The DIVERSETYPER approach leverages several key components. The first component is the **context vector** provided by the BERT-style model pre-trained on code (yellow circle in Figure 4. For instance, for a particular sequence of code s, each t^{th} source token s_t, has a context vector h_t existing in $\mathbb{R}^d$ where d is a dimension determined by the neural model architecture. This context vector, h_t, is the vector we propose is capable of representing common-types and user-defined types, when fine-tuned under the proper loss function and training procedure.

The second component is the **loss function** which is required in order to transform the aforementioned context vector. The loss function determines what the model learns and how efficiently it is learned; typically this is comprised with (sub) losses focused on the optimization of a particular objective. To learn common-types, the model defines a categorical learning signal where it learns to associate each token with a common type. For simplicity, we term this signal as **Task 1** (Section 4.2.1). To learn user-defined types, the model uses a deep similarity learning signal we call **Task 2** (Section 4.2.2). **Task 2** transforms the context vector into a new user-defined type vector (orange circle Figure 4), termed h_s, which can be used to compare new declarations with individual uses of these declarations. The aforementioned

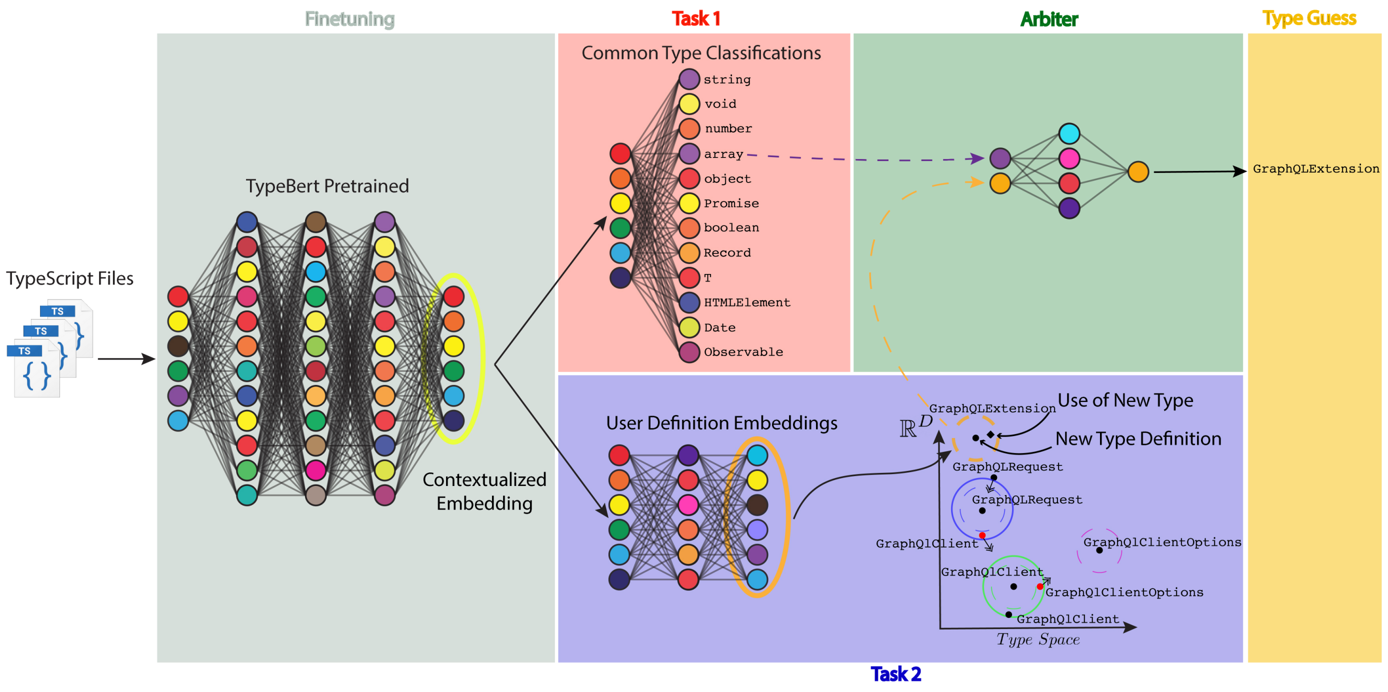


Fig. 4: Overview of DIVERSETYPER. *Training*: DiverseTyper is trained end-to-end with two tasks Task1 and Task 2. Task 1: a classification layer is trained with a cross-entropy loss on the target types. Task 2: An alignment of user-defined types and the use of types with a triplet loss. A red dot indicates a type annotation that is incorrectly positioned closest to a different type. The model learns to correct this and incrementally shift the embedding to the nearest same labeled type. Type declarations, coexist in the same type space with normal type uses. *Inference*: DiverseTyper: A deep transformer, outputs a context embedding (yellow circle) which is projected into a common type guess (Task 1) and a user-defined type embedding (orange circle) corresponding to a user defined type-space (Task 2). To convert the user-defined type embedding to a type, a k nearest neighbor (kNN) search returns the nearest neighboring types. Arbiter: An independently trained multi-layered perceptron (MLP) decides which type prediction is better between the common type and the user-defined type.

transformation into a user-defined type vector, h_s, requires DIVERSETYPER to use additional hidden layers shown in Task 2 of Figure 4. We add additional layers to allow the context vector, h_t, to contort into a new representation that might differ greatly from **Task 1**, but be more suitable for **Task 2**; these layers introduce additional degrees of freedom that can be trained using the above mentioned loss function. As previously mentioned, a representation might become more or less suitable for one task over the other during the training procedure which means the model should turn up or down the amount of feedback from each task. With two losses of varying degrees of importance at a particular moment in training, the model must judiciously combine the losses in a manner that reflects the model's confidence in them. Another term for this is *uncertainty*.

While there exists many weighing strategies [29], we find using the inverse variance of a loss is a suitable weighing term, i.e., $\frac{1}{\text{variance}}$ *viz.* $\frac{1}{\text{uncertainty}}$. When the uncertainty is high, the model will weigh the signal less and vice versa when the uncertainty is low. The next sections will describe these three components in detail. We first describe the two losses and consequentially how we weigh them with uncertainty.

4.2.1 Task 1: Classifying common-types

The first task is based on the model's ability to classify commonly occurring types; these types are often self-evident through simple expressions containing string manipulation or mathematical operations. At a high level, the machine learning model is given a sequence s and returns embeddings for each token s_t. The embedding is used as input to the classifier to produce types for each token, given the token can assume a type. With a type associated to each variable, parameter, and function code token, the model can check the predicted type with the ground truth and learn a distribution that matches the ground truth distribution. This is a popular, yet effective way to classify a large bulk of type annotations but is poor in predicting a large breadth of types. In the next paragraphs we discuss the details of the common type classifier and motivate an alternative for user-defined types.

In order to learn the ground truth type distribution, machine learning models, like DIVERSETYPER, require a loss or feedback from the various tasks they are trying to solve. To get a loss value for **Task 1**, DIVERSETYPER must calculate the following for each type annotation. Per source token s_t and a corresponding type annotation τ, DIVERSETYPER passes the hidden state h_t (yellow circle in Figure 4) to a classification layer (Task 1 in Figure 4). This classification layer is defined as a probability distribution formed from the linear combination of the hidden state h_t and a learned type representation r_τ. The linear combination produces *logits* or log-odds that can be mapped to a probability distribution with the softmax equation (seen in Figure 5). This leads to a probability associated with each type. In machine learning terminology, this is defined below,

$$P_{s_t}(\tau) = \frac{\exp(h_t^T r_\tau + b_\tau)}{\sum_{\tau'}^{\mathcal{T}} \exp(h_t^T r_{\tau'} + b_{\tau'})} \tag{1}$$

where $P_{s_t}(\tau) \in (0,1)$ and τ is $\in \mathcal{T}$, which is the finite set of known types. Equation 1 is a classic equation in machine learning for predicting probabilities from a finite set of classes. During training, these probabilities are often incorrect and must be adjusted to the underlying true probability. This is accomplished with the types' true labels.

With the probability across common types, we seek to maximize the expected probability $P_{s_t}(\tau)$ over the training set by minimizing the corresponding loss $\mathcal{L}_{\text{CLASS}}$. We use a standard classification (cross-entropy) loss:

$$\mathcal{L}_{\text{CLASS}}(s_t, \tau) = -\sum_{\tau'}^{\mathcal{T}} y_{s_t} log(P_{s_t}(\tau)) \tag{2}$$

where y_{s_t} is the ground truth type for the source code token s_t. By optimizing this signal, the model can learn to adjust it's internal parameters for common types according to their true distribution in code. Figure 5, illustrates how Equation 1 normalizes a neural network output to probabilities.

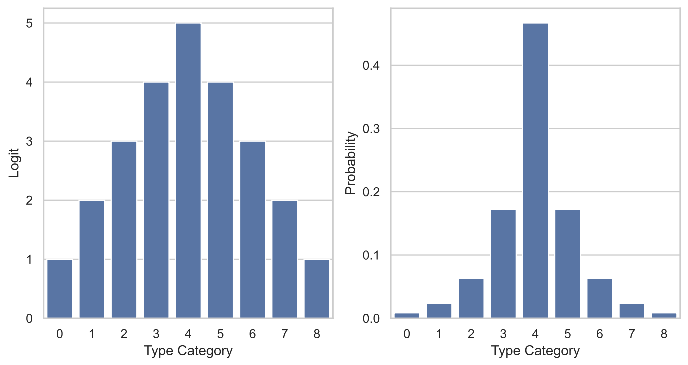


Fig. 5: The softmax equation, Equation 1, forces model outputs into probabilities. Left: Raw values from the network regarding the log likelihood of a category. Right: Equation 1 forces the distribution into a probability distribution with a cumulative sum of 1.

However, infrequent and user-defined types are not represented by the model's fixed type set (shown in the red block of Task 1 in Figure 4), and cannot be learned as previously described. To reiterate, this is because the model's output is finite. We address this issue with **Task 2**, capable of learning infrequent and user-defined types (blue block in Figure 4).

4.2.2 Task 2: Learning User-Defined Types

Probabilistic type inference approaches perform best when a sample of type annotations are reflective of the population. This is very difficult for user-defined types, like class declarations and type interfaces, which typically occur infrequently within the scope of a single project. This is a major reason why data driven approaches fail on user-defined types. On the contrary, if this problem is mapped into a matching task from declarations to uses of those type, then there are examples in practically every project. Despite class declarations and corresponding uses being different code entities, according to the contextual embedding, we can establish an alignment task and adjust those distant embeddings to be translated into nearby embeddings. Thus, DiverseTyper can leverage rare types into many good training examples of matching across thousands of projects irrespective of the sparsity of individual types across the entire corpora. In the following paragraphs, we examine how the aforementioned similarity is defined and learned by DiverseTyper.

To learn infrequent and user-defined types, the model uses deep similarity learning [30] to associate type declarations with the respective annotation. In machine learning, we can define similarity in many arbitrary ways as similarity is a subjective measure. We use a loss that compares embeddings to other embeddings with respect to the embeddings' labels (again could be subjectively assigned). If two embeddings correspond to the same item, according to the label, and the distance between the embeddings is large, then the similarity would be low when it should be high; this can be corrected with the model producing *better* embeddings. To elucidate a more formal concept of similarity, we first introduce the notion of a *triplet*, the fundamental building block to Task 2.

Similarity for types is a relative measure defined by grouping same and different types. For a particular type x_t, the model finds a type x_t^+ with the same label, and different type x_t^-. Together these three elements define a triplet as (x_t, x_t^+, x_t^-). A distinct property of a triplet is that there is a notion of similarity between the reference point or *anchor* x_t, the *positive* point x_t^+, and *negative* point x_t^-. In Task 2 of Figure 4, the black points are same labeled types with the *anchor* being the focal point of the circle. The red dots are *negative* points where the label is different than the anchor. The red points should be moved closer to the center of the correct point through the optimization of a training loss. In Figure 4 the negative examples have arrows to demonstrate the direction the model is moving the points in order to correct the prediction.

To use the triplet (x_t, x_t^+, x_t^-) for learning user-defined types, DiverseTyper randomly constructs triplets from embeddings it produces (orange circle in Figure 4). In Task 2 of Figure 4, the types anchor and positive will be annotations that are both the same, i.e. `GraphQlClient` and another annotation of `GraphQlClient`, with a negative annotation of a different label, i.e, `GraphQlClientOptions`. The novelty of DiverseTyper is that the declarations, i.e, `class GraphQlClient() {...}` are valid positive annotations, despite the differing pretrained embeddings indicating they are different entities; we override this model assumption by dictating a new notion of similarity; how a developer interprets these code entities.

By selecting our triplet in this manner, indiscriminate of declaration and annotation, DiverseTyper learns an optimal representation of user-defined types and has the capability of using any declaration for type inference irregardless of if the model has seen it before. The unique combination of pre-trained vectors with type clustering is what makes our model perform so well for never before seen types.

More formally the goal is a final representation where the same labeled types and differently labeled types are separated by a margin or distance m. Using the aforementioned

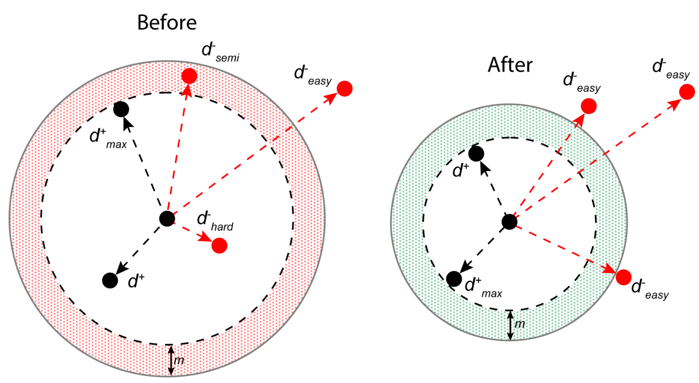


Fig. 6: Illustration of triplet loss with semi-hard negatives. The center is an anchor type surrounded by the same types (black dots) and exist within d^+_{max} where d is the L_2 distance between points. The distance d^+_{max} defines a neighborhood shown with the visualization of a circle. Differently labeled types (red dots) can exist within d^+_{max} (hard negative), $d^+_{max} + m$ (semi-hard negative), and greater than the neighborhood d^- (easy negatives). By optimizing the triplet loss in the left circle, the model adjusts the embeddings so a lower loss occurs. This is accomplished by moving + points (the same types) closer to the center and moving − points (different types) away, further than the margin (dotted boundary). The final result (right circle) is the optimized type space where ∀ − points, $d^- > d^+_{max} + m$.

notation, *anchor*, *positive* (+), and *negative* (-) to represent the labels of similarity and h_s for the embedding of an ith type location,

$$\|h_{s_i} - h^+_{s_i}\| + m < \|h_{s_i} - h^-_{s_i}\| \tag{3}$$

$\forall\{h_{s_i}, h^+_{s_i}, h^-_{s_i}\} \in \mathbb{T}$ where $\mathbb{T}$ is the set of all possible triplets in the mini-batch, or iteration of training. The notation h_{s_i}, $h^+_{s_i}$, $h^-_{s_i}$, are the same anchor, positive, and negative versions (as used above) but referring to the embedding h_s (orange circle in Figure 4). The embeddings are considered positives and negatives by their respective type label; the same type is positive and the different type label is a negative. With all three representations, h_{s_i}, $h^+_{s_i}$, $h^-_{s_i}$, the triplet loss for a mini-batch with B examples is defined as

$$\mathcal{L}_{\text{TRIPLET}}(h_s, h^+_s, h^-_s) = \sum_i^B \left[\|h_{s_i} - h^+_{s_i}\| - \|h_{s_i} - h^-_{s_i}\| + m\right]_+ \tag{4}$$

This formula simply means, that the model incurs a loss when the distance between the anchor and negative point (different labels) is less than the distance between the anchor and positive point (same labels); a violation of the type space. The margin is added so that the loss occurs even if the negative point is within the extra boundary. $\mathcal{L}_{\text{TRIPLET}}$, (Equation 4) rewards an embedding $h^+_{s_i}$ that is closer to the anchor h_{s_i} and penalizes $h^-_{s_i}$ that exists within the margin m. Equation 4 calculates the loss across all possible triplets $\mathbb{T}$ in the mini-batch B. Calculating the loss across all points is less ideal, computationally, as many points easily result in a 0 loss, i.e, they are correctly situated. Realistically, only a few triplets provide a valuable loss; namely ones where the similarity notion is violated and the loss significant. To adjust for the mentioned inefficiency, we use a triplet mining technique called semi-hard negative mining [31] that helps find the most valuable triplets and optimize those.

Figure 6 demonstrates the selection of semi-hard negatives denoted in the red margin. The green circle in Figure 6 is a converged similarity representation where training is complete. Practically, until perfect convergence occurs between all types, there will always occur a decreasing loss.

Subsequent of above, we find that $\mathcal{L}_{\text{Triplet}}$ converges faster than $\mathcal{L}_{\text{Class}}$ due to the relative simplicity of aligning embeddings, but has an increased variance that reduces the effectiveness of $\mathcal{L}_{\text{Class}}$. This trade off means excellent performance of user-defined types with some degradation of the common classifier. We can counter this effect quite significantly by judiciously combining the errors from the loss functions such that each loss is optimized as best as possible. We explore this is the following section.

4.2.3 An Optimal Balance of Losses

As described above, the model learns different losses **Task 1** and **Task 2**: common types for **Task 1**, and user-defined types for **Task 2**. In practice, deep multi-task learning models have claimed improvements in performance by sharing representations, in our case, the pre-trained vector in yellow in Figure 4 between both tasks [32]. The ideal contribution from each task is not known *a priori* and typically requires searching for a good weighing strategy. In lieu of searching for the perfect strategy with trial and error techniques, like a grid-search, an alternative learnable weighing technique can be used; the model can learn the best weighing scheme as it trains (learning to learn as the model is learning). The learnable weighing technique can be described as learning to estimate the uncertainty of the loss [32] from the two typing tasks. A recent empirical survey [29] for optimal multi-task weighing strategies demonstrated that uncertainty losses [32], [33] performed best. We follow Kendall *et al.* [32] approach of combining a discrete output (categorical) and continuous output (similarity). The combined loss follows,

$$\mathcal{L} = \frac{1}{\sigma^2_{\text{Class}}}\mathcal{L}_{\text{Class}} + \frac{1}{2\sigma^2_{\text{Triplet}}}\mathcal{L}_{\text{Triplet}} + \log\sigma_{\text{Class}} + \log\sigma_{\text{Triplet}} \tag{5}$$

where σ represents the standard deviation. σ^2_{Class} and $\sigma^2_{\text{Triplet}}$ represent the cumulative learned variance (uncertainty) per task. For learning stability, the model learns $\log\sigma^2$ rather than regressing on σ^2. For more details on this, please refer to Kendall *et al.* [32].

We can interpret Equation 5 as a combination of the losses with weights for each one. Higher σ values will decrease the impact of the loss signal from the corresponding task, and smaller σ values increase it. Finally, the regularizing terms, $\log\sigma_{\text{Class}}$ and $\log\sigma_{\text{Triplet}}$, penalize the model when the scale of the σ values are too large. The loss asymptotically goes to zero as both sigmas approach infinity, and while the the model would have a zero loss, it wouldn't learn either task! In summary, the loss from Equation 5 can be viewed simply as a *learned* weighted loss with some bias, i.e.,

$$\mathcal{L} = \omega_1\mathcal{L}_{\text{Class}} + \omega_2\mathcal{L}_{\text{Triplet}} + b \tag{6}$$

A major benefit of a *learned* weighting strategy, like Equation 6, is that the final weights are automatically determined

TABLE 2: Building an Arbiter for Metric and Probability Type-spaces

Method	User $\text{Def}_{\text{embedding}}$	$\text{kNN}_{\text{Similarity} \lessdot}$	$\text{kNN}_{\text{User Def}}$	$\text{Class}_{\mathbb{P}}$	$\text{Class}_{\text{Labels}}$	Attention	Data %	Accuracy %
Sort							0%	90.92
Neural Network	✓						10%	82.76
Neural Network	✓	✓					10%	83.43
Neural Network	✓	✓	✓	✓	✓		10%	91.67
Neural Network	✓	✓	✓	✓	✓	✓	10%	91.72
Neural Network		✓	✓	✓	✓		10%	91.96
Neural Network		✓	✓	✓	✓	✓	10%	91.20
Neural Network		✓	✓	✓	✓		50%	**94.14**
Neural Network		✓	✓	✓	✓	✓	50%	**93.54**

The highest performing arbiter has a configuration consisting of a neural network with inputs of user-defined type labels and similarity scores, common-type labels and probability scores, and 50% of the training data.

by the model over the data; this is clearly preferable to hand-engineering the weights in each problem setting.

The derived (combined) loss, Equation 6, allows DIVERSETYPER to focus on both aspects of type prediction **jointly**: the optimization of common type classifications and the clustering of rare and user-defined types. This work to our knowledge, is the first to apply an effective, general multi-tasking approach to type prediction; this approach may also benefit other SE settings that performance on two tasks must be effectively balanced.

In order to use a trained DIVERSETYPER, we must define its inference methods.

4.3 Inference

This section introduces how DIVERSETYPER makes either: (1) a common type prediction or (2) a user-defined type prediction. This is done with an *arbiter*; which (like a human arbiter) settles "disputes" between the common-type and user-defined type predictions, as we now explain.

4.3.1 Common Type or User-Defined Type?

During inference, DIVERSETYPER outputs a common-type guess (purple arrow in Figure 4) and a user-defined type embedding (orange arrow in Figure 4). The common-type guessing mechanism is the classification layer that outputs common-types from the fixed set of types with probabilities per Section 4.2.1. The user-defined type guessing mechanism is the closest neighbor lookup using the user-defined type embedding first defined in Section 4.2.2.

The neighborhood lookup is an efficient k-nearest-neighbor (kNN) search algorithm[3] across all training set declarations and uses of those declarations. DIVERSETYPER adds the testing set declarations only because the declarations are available at the time the corresponding type can be predicted; the model has never seen these test declarations before. If the model is successful at never before seen types, these new declarations will be embedded near relevant usages. Finally, the search returns the distance which $\in (0, 1]$ with 0 being an exact match. When calculating the similarity, we can take $1 -$ distance.

We note that the similarity measure is <u>not</u> a probability measure, looks nothing like Figure 5, and thus the two are not comparable unless some mapping is applied. This presents a quandary: given two incommensurate measures, how would an arbiter resolve a "dispute" when different type labels are offered by the two? In the following section, we discuss how a special mapping can be baked into a neural network, automatically picking the best type.

4.3.2 Arbiter

The arbiter first obtains a list of common-types and their probabilities from the common-type guesser. Next, a k nearest neighbors search of user-defined types is performed, returning a second list of user-defined types and the respective similarities. The arbiter combines unique types from each, sorts them, and returns L_{mixed}, a set of mixed types. However, sometimes both type prediction mechanisms present similar scores, so, how to choose the very best type?

As an initial step, we compare our approach with a baseline "sorting" approach despite the two different metrics: probability and similarity. This approach consists of combining both sets and sorting irrespective of type metric. To our surprise, this simple baseline performed better than expected, with an accuracy of 90.92%. In a later analysis of the type spaces in Section 6.2, it can be inferred that (extremely) low distance points often yield the correct prediction, comfortably overriding an incorrectly predicted common-type's probability. Likewise, the probabilistic predictions are highly confident when correct in the BERT-family models and thus can easily override the distance of an incorrect user-defined type. The baseline is effective in most cases but there are some scenarios where the correct answer is enigmatic. This is where we find performance of a specifically trained neural network arbiter beats the simple sorting baseline.

We tried several designs for the neural network in Table 2. We manipulate the network's access to various inputs: the similarity embedding, similarity distance, user-defined type label, common-type probability, common-type label, attention, and amount of data trained on. From the above ablations, we find the best performing arbiter uses the top 5 common-type prediction probabilities and user-defined type similarity scores with the respective type labels. We create a dataset with the described inputs in Table 2. The output label is 0 if the common-type mechanism gets the answer correct and 1 if the user-defined type mechanism gets the solution correct. The arbiter's neural network is trained on a holdout portion of the training data while the remainder is used to learn the kNN search. The trained model is selected by performance on a validation set and evaluated on the test set per Table 2. The trained model was very good as a binary classifier picking between the two type predictors (common

3. https://github.com/spotify/annoy

TABLE 3: Multi-Task Type Annotation Datasets

Approach	Type Inference Dataset			User Definition Dataset		
	Projects	Files	Annotations	Projects	Files	Annotations (Definitions/Usage)
TypeBert [13]	20,860	1,474,418	12,920,988	✗	✗	✗
DiverseTyper	20,860	1,474,418	12,920,988	14,309	225,551	3,204,180 (50% / 50%)

DIVERSETYPER uses two joint learning objectives to learn common-type and user-defined type associations. We provide no additional type inference data to demonstrate the effectiveness of the supplemental objective. The true ratio of definitions to usage is 32% / 68% but we over-sample definitions such that each batch has both a definition and its corresponding use.

vs. user-defined type). As seen in Figure 7, this classifier has a strong receiver operating characteristic (ROC) curve with an area under the curve (AUC) of .93. This ROC and AUC demonstrates that the classifier is effective at arbitrating the two type recommendation mechanisms. The next section examines the performance of DIVERSETYPER with several research questions (RQs).

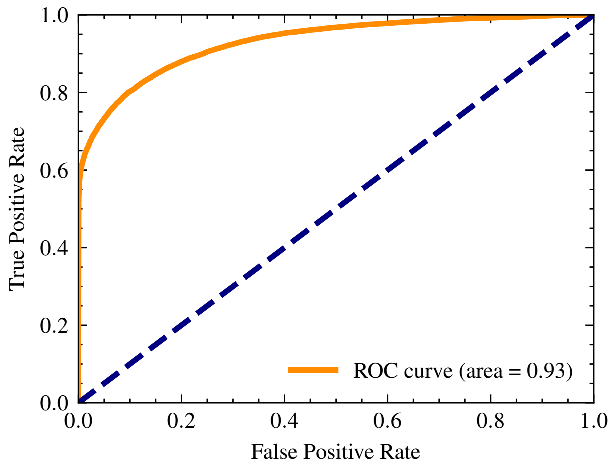


Fig. 7: Receiver Operating Characteristic (ROC) curve of the Arbiter. Area Under the Curve (AUC) is .93 which means it is an excellent classifier.

5 QUANTITATIVE EVALUATION

In this section we present the dataset DIVERSETYPER is trained and evaluated on, metrics for type inference evaluation, and our baselines for DIVERSETYPER.

Then we answer the following research questions:

RQ1: How effective is DIVERSETYPER compared to baseline approaches?
RQ2: Can DIVERSETYPER predict user-defined types?
RQ3: How does DIVERSETYPER perform on previously unseen types?

5.1 Dataset

We use the same 20,860 projects collected in TypeBert [13] for the type inference dataset and maintain the same data splits between train, test, and validation. This dataset contains human-annotated types on variables, parameters, functions and method declarations. Types range from common-types, i.e., `number` and `string` to library and user-defined types like `dynamodb` and `Point`. This dataset does not have the user-defined type declarations but only the annotations. We supplement this dataset with additional data extracted from the same set of projects that includes user-defined type declarations across the existing dataset splits in TypeBert [13]. The purpose of a project level data split is twofold: (1) so files seen at test time are never seen at training time and (2) to accurately compare the contribution of training on user-definitions.

To extract user-defined type declarations and the locations of their use, we wrote a code parser to localize user declarations denoted with keywords `interface` and `class`. The parser finds use of a user-defined type corresponding to the declaration within project scope. The supplemental user-defined types dataset contains 362,759 (32%) declarations and 1,141,734 (68%) uses across 14,309 projects and 225,551 files. This supplemental dataset is only used in training and is not used to evaluate the model's performance. We have released the supplemental dataset in our GitHub repository.

To evaluate DIVERSETYPER, we follow standard evaluation procedure from previous works [12], [13], [17] and only use human-annotated types for evaluation. The intuition is that compiler inferred types are typically easy and saturate performance scores, where as, human annotations are more difficult and meaningful. Following standard practice we allow the model to train with both the "easy" compiler inferred types and the "hard" human-annotations. Also in line with other works, we exclude the wildcard type `any` in our evaluation. Finally, a key practice is to perform deduplication on a dataset [34]. Code duplication between and within training and evaluation sets has historically existed in previous works, prior to Allamanis [34] demonstrating duplication leads to artificially elevated evaluation scores. To conclude this section, we provide the breakdown of top types in each dataset. Observe the type breakdown in Table 4. User-defined types occur infrequently while common-types

TABLE 4: Top Types In Datasets

Type-Inference Dataset			User Definition Dataset		
Types	Count	Data %	**User Defs**	Count	Data %
string	2,103,227	16.19	Node	7,583	.0584
void	1,324,632	10.20	State	6,843	.0527
number	1,213,432	9.34	Props	6,536	.0503
array	915,837	7.05	User	5,798	.0446
object	635,155	4.89	Context	5,367	.0413
Promise	549,219	4.23	Type	3,961	.0305
boolean	514,801	3.96	Player	3,386	.0261
Sum	**7,256,303**	**55.87**		**39,474**	**.3039**

account for >55% of the original type annotations. We believe the stark distributional difference between common-types and user-defined types necessitates the separate mechanism for predicting user-defined types. Again, this is because the vast majority of types are often project-specific, locally defined, and only occur only a handful of times. It is important to note that DIVERSETYPER contextualizes user-definitions, so different declarations with the same name, such as User or State, will be represented by a separate data point. This is particularly useful for the model because it can condition on small differences in the definition such as the presence of attributes.

5.2 Metrics

We use Top-1 Accuracy (exact match) and Top-5 Accuracy (correct prediction in the top 5 guesses) for subsets of types exactly in line with previous works. The categories are as follows:

Top 100: The most frequent types such as native types `int` and `string` and types not considered user-defined within the top 100 rank [12], [13], [17].

Other: Types that are common but occur outside of the top 100 and are not user-defined. Examples would be commonly used library types like `ArrayBuffer`, `Entity`, `FunctionComponent` just to name a few.

User Defined: Types that correspond to a class, enum, or type interface where the type is declared within the same project scope. Examples would be developer specified types that occur quite rarely if at all in other projects, i.e., `KindaShiftView`, `VRMSpringBone`, `Iterm2ColorName`.

Unknown: In previous works with fixed type vocabularies [9], [10], [11], [12], [13] type inference models would predict `UNK` if the type exceeded its classification capabilities. TypeBert [13] accounted for `UNK` predictions by counting them against the performance of TypeBert. This meant that ∼8% of predictions in the test set were automatically considered incorrect as a function of its model architecture. DIVERSETYPER has no type limitations, and *never predicts* `UNK` as it can always defer to the user-defined similarity vector.

5.3 Baselines

We compare DIVERSETYPER to three TypeScript baselines: LambdaNet [17], OptTyper [12], and TypeBert [13].

LambdaNet a graph neural network (GNN) approach that links variables and logical constraints to approximate a type dependency graph. The architecture can predict common-types in the top 100 and user-defined types available in the type-space with a pointer network.

OptTyper performs probabilistic type inference across the top 100 most-frequent types. OptTyper combines a continuous interpretation of logical constraints derived by static type inference with the natural constraints learned from deep learning across large code bases.

TypeBert uses BERT-style pre-training with large scale corpora in addition to a large fine-tuning dataset to train a type inference model. The implementation is similar to sequence tagging in NLP.

5.4 RQ1: Effectiveness of DIVERSETYPER

We evaluate the effectiveness of DIVERSETYPER over the type categories in Section 5.2. We also perform ablation analysis by varying different elements of its architecture to understand why DIVERSETYPER is effective.

Type Performance

As shown in Table 5, we report the top-1 accuracy and top-5 accuracy across type categories defined in metrics. DIVERSETYPER has the strongest Top 1 overall scores at 79.71% accuracy overall a +8.59% absolute improvement over TypeBert. DIVERSETYPER scores the highest Top 5 accuracy overall meaning that DIVERSETYPER is providing more relevant scores across all of its predictions. Both DIVERSETYPER models demonstrate Top-5 scores higher than TypeBert and LambdaNet which is notable because LambdaNet uses static analysis and pointer mechanisms to predict user-defined types. This means that not only does DIVERSETYPER do a better job at recognizing user-defined types, but is additionally capable of referencing declarations from its kNN search even when the declarations are unavailable or missing in the source code.

We observe a trade-off in the top 40,000 types (Top 100 and Others). This might be due to complications arising from training of **Task 1** and **Task 2** together. A possible source of this, is that often developers override library types such as `Node`, `State`, `User`, `Context` etc. where the common-type classifier gets disrupted in favor of learning user-defined type declarations. The aforementioned types are also the most frequent user-defined types as seen back in Table 4.

Ablations

Deep learning models are difficult to understand and ablations provide insights to the model's learned representations. We vary the architecture in meaningful ways to gain insights to how different methods affect prediction capability. We organize our ablation results in Table 6.

First we define a base version of DIVERSETYPER, $\mathrm{DT}_{\mathrm{base}}$, where the model is trained jointly on the two tasks: **Task 1** and **Task 2**. This model is evaluated with **only** the common-type classifier. The purpose of this evaluation is to demonstrate how much accuracy was lost in the traditional classifier from the jointly learned objectives. We observe that others and user-defined types perform very poorly. This indicates that the type representation r_τ (defined in section 4.2.1) is not a meaningful type representation of user-defined types anymore. This intuition is confirmed with the performance of the same model using the user-defined kNN search. The model that employs the kNN search is denoted **base$_{\mathbf{e2e}}$ + U.D network$_{\mathbf{e2e}}$ + Arbiter**. From a training perspective, the network is equivalent to $\mathrm{DT}_{\mathrm{base}}$ and yet performs +50%. This indicates that the user-defined type representations are *moving* in favor of the similarity representation. This comparison between the two models with the same training but different inference mechanisms shows the effectiveness of our proposed learning approach.

In order of incremental improvement, we try a model where we use **base** and only sample from the user-defined types when the classifier predictions `UNK` viz. the model does not know the type. **base + `UNK` filler** improves performance but not considerably.

TABLE 5: Accuracy Comparisons with DIVERSETYPER Across Binned Types

Model	Top 1 Acc %				Top 5 Acc %			
	Top 100	Other	User Def	Overall	Top 100	Other	User Def	Overall
LambdaNet [17]	66.9	N/R	53.4	64.2	86.2	N/R	77.7	84.5
OPTTyper [12]	76	N/R	N/R	N/R	N/R	N/R	N/R	N/R
TypeBert [13]	**89.51**	**50.49**	41.40	71.12	**98.51**	**70.34**	55.02	81.88
DIVERSETYPER$_{\text{ARBITER}}$	83.51	46.92	**72.53**	**79.30**	92.94	60.53	**81.59**	**88.59**
DIVERSETYPER$_{\text{ARBITER}_{\text{NN}}}$	84.78	43.29	**71.56**	**79.71**	90.88	53.25	**77.18**	**85.62**

TABLE 6: DIVERSETYPER Ablations

Model	Top 1 Acc %				Top 5 Acc %			
	Top 100	Other	User Def	Overall	Top 100	Other	User Def	Overall
DT$_{\text{base}}$	82.28	24.69	23.41	59.77	95.76	41.48	36.49	73.10
DT$_{\text{base + UNK filler}}$	82.46	33.56	46.40	68.67	96.59	59.44	78.69	79.43
DT$_{\text{TypeBert + U.D. network}}$	**89.69**	**50.49**	41.39	71.16	**98.57**	**70.34**	55.0	81.86
DT$_{\text{base}_{\text{e2e}}\text{ + U.D. network}_{\text{e2e}}\text{+Arbiter}}$	**83.51**	46.92	**72.53**	**79.30**	92.94	60.53	**81.59**	**88.59**
DT$_{\text{base}_{\text{e2e}}\text{ + U.D. network}_{\text{e2e}}\text{+ Arbiter}_{\text{NN}}}$	**84.78**	43.29	**71.56**	**79.71**	90.88	53.25	77.18	**85.62**

base: DIVERSETYPER with only common-type classifier (same as TypeBert) used for evaluation
base + UNK filler: fill UNK predictions with top 1 guess from user-defined type mechanism.
TypeBert + User-Defined network: initialize DIVERSETYPER with TypeBert's weights. These weights are not changed.
base$_{\text{e2e}}$ + User-Defined network$_{\text{e2e}}$ + Arbiter: Use basic (sort) arbiter to pick top 1 type between common-type and user-defined type mechanism.
base$_{\text{e2e}}$ + User-Defined network$_{\text{e2e}}$ + Arbiter$_{\text{NN}}$: Use neural network arbiter to pick top 1 type between common-type and user-defined type mechanism.
* e2e: learned jointly with end-to-end training
* NN: Neural network

The next ablation is to initialized DIVERSETYPER with TypeBert weights and train with only the user-defined supplemental dataset while holding the TypeBert weights stationary. This model has the label **TypeBert + U.D network**. We can see that the user-defined types were not learned by the model and this model has almost the same performance as TypeBert. We anticipate that this akin performance is because the final type representations learned in TypeBert drop relevant features about user-defined types in the process of partitioning the common-type space. Interestingly, the performance of this model marginally surpasses TypeBert indicating the network learned relevant information pertaining to the Top 100 in the user-defined type dataset.

The final two ablations are our best models where we use the base model, plus the kNN search, and the arbiter for inference. The first model uses only the sorting arbiter and is labeled **base$_{\text{e2e}}$ + U.D network$_{\text{e2e}}$ + Arbiter**. In this ablation, DIVERSETYPER is capable of using the information learned during training to match declarations with annotations. This model performs best in user-defined type exact-match and top-5 overall. The second of the two best DIVERSETYPER models uses the same base model, kNN search, but with a neural network arbiter. This model is denoted **base$_{\text{e2e}}$ + U.D network$_{\text{e2e}}$ + Arbiter$_{\text{NN}}$** and performs the best overall in exact matches. These models reinforce the hypothesis that the model is capable of taking advantage of user-defined type matching with declarations with a sizeable performance increase in the overall and user-defined types category.

DIVERSETYPER improves overall performance **8.59%** over TypeBert, a **13.38%** percent increase of TypeBert's improvement over LambdaNet.

5.5 RQ2: Prediction on User-Defined Types

In this section we evaluate DIVERSETYPER's capabilities on user-defined types. Observe the reported user-defined type accuracy in Table 5 and Table 6. In comparison with other approaches DIVERSETYPER performs 31.13% better than TypeBert across user-defined types with almost the same architecture. The performance of DIVERSETYPER informs us that the user-defined similarity embedding is substantially more effective for rare and user-defined types than a fixed vocabulary. The performance improvement can be attributed two characteristics of the user-defined similarity embeddings. First, **Task 2** introduces a representational "slack" by clustering similar representations rather than strict *partitioning* of dimensional space. Second, unlike other deep similarity learning approaches [15], [18] that only group similar annotations, DIVERSETYPER places the `class` and `interface` declarations into the user-defined type-space. By learning the relationship between declaration and annotation, DIVERSETYPER makes the user-defined type set generalizable to novel `class` and `interface` declarations.

DIVERSETYPER improves user-defined type accuracy on LambdaNet by **19.13**% and **31.13**% over TypeBert.

5.6 RQ3: Performance on Never Seen Types

We test DIVERSETYPER with the hardest type annotations, i.e., types that have never been seen before. By evaluating the model on never seen types, we gauge how well DIVERSETYPER will do on brand-new types added by developers. Table 7 shows that DIVERSETYPER scores

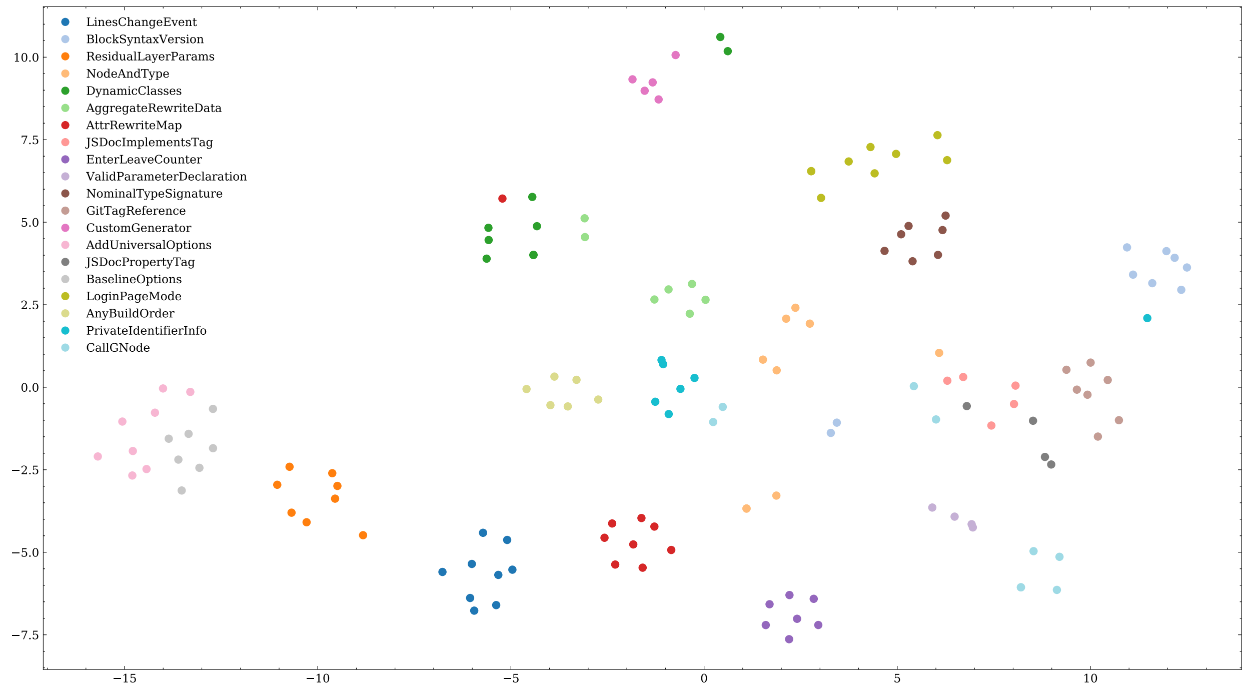


Fig. 8: t-SNE plot of aligned user-defined types and the respective usage. When a developer defines a new type and requires type inference of this new type, the usage embedding will be clustered with the definition making type inference of new user-defined types possible with high accuracy.

TABLE 7: DIVERSETYPER on Never Seen Types

Type Accuracy							
Top 1 Acc				Top 5 Acc			
Top 100 0%	Other 19.44%	User Def 80.56%	Overall 100%	Top 100 0%	Other 19.44%	User Def 80.56%	Overall 100%
N/A	1.723	54.82	44.50	N/A	3.62	58.48	47.82

9.04% of test set contains types never seen before. Top 100 accounts for 0% of never seen types. Percents under type categories are proportion of never seen types. For example, 80.56% of never seen types are user definitions.

a commendable 44.50% top-1 accuracy on types it has never seen. DIVERSETYPER performs even better for user-defined types, at 54.82% top-1 accuracy. This result is more consequential when we see that user-defined types occupy about ∼81% of never seen types. This is a promising result for a deep learning based approach where performance is typically dependent on comprehensive examples from training data.

DIVERSETYPER's user-defined type mechanism successfully annotates never seen types ∼**55%** of the time.

6 QUALITATIVE EVALUATION

Our qualitative evaluation serves to elucidate the inner workings of DIVERSETYPER and its user-defined type similarity predictions. In this section we answer the following research question:

RQ4: Can we visually inspect DIVERSETYPER's performance over other methods?

RQ5: How does DIVERSETYPER cluster typical user-defined types?

6.1 RQ4: Inspecting Consequential Annotations

Figure 2 is a snippet from a popular Microsoft GitHub repository. We can see that the type `CodeSandboxLanguage` is defined with the `type` keyword indicating it is a user-defined type. It is defined within a TSX file (TypeScript's equivalence to JSX) and imported into the main `.ts` file. The user-defined type `CodeSandboxLanguage` is used in the definition of an object `createPackageJson`. DIVERSETYPER recognizes the intra-project type declaration and properly assigns it in the object. DIVERSETYPER recommends both lexical similar types such as `Language` and hints at functionality preservation with relevant type `RemoteDebugLanguage` for a sandbox environment. TypeBert only recommends lexical similar types such as `Language`, `ILanguage`, and `LanguageCode`. LambdaNet considers the correct type `CodeSandboxLanguage` to be Out-Of-Vocabulary (OOV) because it did not populate in it's list of possible types.

The outcomes from the three models are not surprising given their advantages and disadvantages. LambdaNet is restricted to 100 types and the types it discovers within the project. LambdaNet has failed to discover `CodeSandboxLanguage` and thus declared it outside of its predictive capability. TypeBert has seen contexts, especially in pretraining, where semantically similar notions of `CodeSandboxLanguage` occur with types `Language` and `LanguageCode`; thus the recommendation. Even if TypeBert had an unbounded classification layer, which is not currently possible in machine learning, it is still less likely that TypeBert would get a majority of user-defined types correct. This is because TypeBert must accurately predict the exact type out of an unbounded list of types where as DIVERSETYPER only has to match the correct declaration to the context and use it correctly.

Figure 3 is a code snippet from another popular coding repository containing LeetCode webapp code. The user-defined type `LeetCodeSolutionProvider` is defined within the same file as the class usage. LambdaNet does not have `LeetCodeSolutionProvider` in its top 5 predictions, but shows some relevant predictions. We observe in other files within the same project, LambdaNet had relevant user-defined type predictions, but fails to place them as the top guess for reasons we do not know; likely a violation of type constraints according to LambdaNet since the type is in the same file. TypeBert fails at most user-defined types as they exceed the vocabulary limit and are not defined in its architecture; this is expected because TypeBert cannot predict infrequent and rare types. DIVERSETYPER accurately predicts the right type where as TypeBert and LambdaNet do not. We observe this similar outcome for other user-defined types in other files within the same project: `LeetCodePreviewProvider`, `LeetCodeExecutor`, `LeetCodeStatusBarController`, `LeetCodeTreeDataProvider`. Most impressively for DIVERSETYPER, these types are all defined and used **once**, demonstrating DIVERSETYPER's capability on rare and infrequent types.

DIVERSETYPER correctly associates *rare* and *infrequent* user-defined types within the same file and across the same project.

6.2 RQ5: Typical User-Defined Type Clusters

The neural type embeddings learned by DIVERSERTYPER are information rich due to the extremely large corpus used for fine-tuning. The visualization of such type embeddings demonstrate important type relationships learned during this fine-tuning. Embedding visualizations reach many data exploratory domains [35], [36], [37]. Commonly, the embedding visualizations are crafted by transforming the high dimensional data into two dimensions while preserving the overall structure of the data. The t-SNE algorithm aims to perform dimensionality reduction to lower dimensions that humans can interpret (2D or 3D), while preserving the structure of the high-dimensional data, as the model interprets it. Figure 8 and Figure 9 are created with the t-SNE algorithm [38]. A visualization of specific embeddings, such as a category of types, can indicate performance across these embeddings. If the type clusters of the embeddings are indistinguishable, then the kNN search will likely not return the correct answer as a kNN search is a function of neighboring data points. An embedding space that is not clustered properly across types is undesirable as the model will fail to generalize properly.

Originally, the pre-trained embeddings are purely context driven, meaning that similarly occurring contexts amongst variables and function names will appear co-located in embeddings. DIVERSETYPER has altered this embedding space to shift context dissimilar sequences, such as type declarations, in a manner that is useful to typing. While this transformation is the goal, the model must to maintain relative structure of the embeddings in places that are not directly relevant to typing. Examples of this include general code syntax and the semantics the model derives from a sequence. DIVERSETYPER must balance maintaining existing code semantics while aligning type declarations derived from those code semantics; especially to generalize on code snippets. We now direct the reader to Figure 8.

Figure 8 is a visualization of the most difficult types to classify, i.e, never seen user-defined types. To visualize the aforementioned clustering of infrequent user-defined types, we plot all user defined types in a t-distributed stochastic neighbor embedding (t-SNE) and select points based on whether the model has seen the type before. Listed in Figure 8, are 20 user-defined types that DIVERSETYPER has never seen in training and occur extremely infrequently. We can see that each type, represented by various colors, is grouped into small clusters of like-typed annotations. This shows that our approach for aligning user-defined types works correctly. In the same figure, we exam the relatedness of never before seen types with the purpose of maintaining semantic meaning. In Figure 8, left, `BaselineOptions` and `AddUniversalOptions` stand out as co-located and complementary in embedding structure. Upon further inspection of why this might be, it is clear that the types are related by instantiation, specifically, a subclass from the base class. The learned complex relationships, such as instantiation, are encouraging for future work because complex type relationships exist. Some of these complex type relationships include but not limited to union and inheritance types (outside the scope of this work), and are potentially tractable for this model architecture. On the right side of Figure 8, it can be observed that `JSDocImplementsTag` is close to `JSDocPropertyTag`. Again, with further inspection to how these tags are used in real projects, we find that both return `SymbolDisplayPart` related types. We conclude from the visualization that the model exhibits a basic understanding of how types are used *and* the complexities within the defined type behavior even when the types are incredibly sparse.

Naturally, we are curious about DIVERSETYPER's capability of clustering common-types with the user-defined type mechanism despite not being trained to do so. Again, we visualize the clustering of common-types with a t-SNE plot in Figure 9. From Figure 9, it can be concluded that common-types are clustered in a similar fashion to user-defined types, but with less defined margins or white space between groups. With **Task 2** only trained on user-defined types and common-types learned by a common-type classifier in **Task 1**, we expect DIVERSETYPER to completely deprioritize the learning of common-type similarity, yet surprisingly maintains proper structure. One might ask, *"Why does common type clustering matter when one can use the common type classifier?"*. The usefulness of common-type clustering is in the case of an arbiter misprediction. If the arbiter picks the user-defined type mechanism over the common-type classifier, the clustering of common-types should provide some redundancy. For example, if a common-type matches a real type annotation from the user-defined search (potentially an infrequent case of overriding a native type), the type prediction will still be correct.

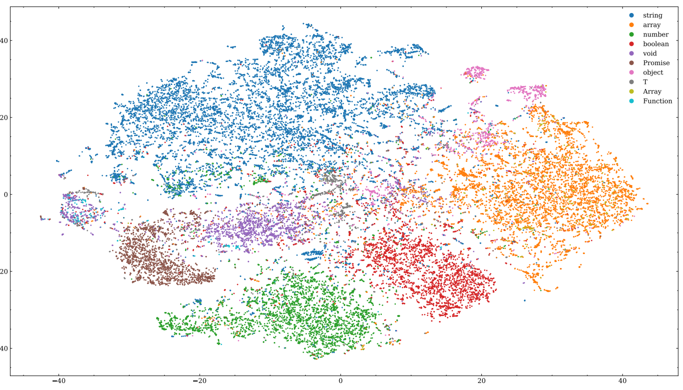


Fig. 9: t-SNE plot of commonly used types. DIVERSETYPER's inherits a the strong performance of common-types across its classification layer. Additionally DIVERSETYPER demonstrates effective clustering in commonly occurring types. If a developer overrides a common-type, i.e., `string`, DIVERSETYPER has both a common type guess and user-defined type guess that the arbiter can choose from.

> DIVERSETYPER groups rare user-defined types in a similar fashion to developers with regard to semantic and syntactic relatedness.

7 Related Work

Pre-Trained Foundational Models
Large scale pre-trained models [16], [39], [40], [41], [42] coined *foundational* models [43], are stacked transformers [44] with various autoencoding objective functions [45], [46] on large unlabeled data. Their success in natural language processing (NLP) has warranted its application in other fields such as computer vision (CV) [47], [48] and software engineering [20], [21], [23]. The extent of foundational "learning" is hotly debated [49] and examined in a software engineering context [50]. Albeit, the performance improvements from pre-training is undeniable as it has set benchmarks for many SOTA tasks across various domains.

Multi-Task Learning
Multi-task learning attempts to efficiently learn multiple objectives from a shared representation [51]. Multi-task learning is prevalent in machine learning fields of natural language processing [52], computer vision [53], and speech recognition [54], but is seldom used in software engineering [55], [56]. Prior approaches use a naïve weighted sum of losses where the losses are uniformed or manually weighed. New approaches include dynamically weighing tasks from gradients [57] and uncertainty [32], [33]. The dynamics of multi-task learning is still not very well understood but has been effective across several applications.

Type Inference
Dynamic type inference techniques [58], [59] and type checkers [60], [61], [62], [63] achieve soundness by enforcing type constraints. Dynamic type-checking provides the convenience of not requiring annotations, and/or having to fix compile-time errors; however, dynamic checking may miss coding errors un-executed parts of programs.

Machine learning can help programmers more conveniently make better use of static type-checking by suggesting type annotations. This works by learning natural type distributions across corpora of code [64]. Hellendoorn *et al.* [9] interpreted type annotation as a tagging task with DeepTyper. Pradel *et al.* [11] designed separate sequence models to infer function types in Python and validate with a type checker. Wei *et al.* [17] used insights from [14] to train a GNN from type dependency graphs. Allamanis *et al.* [18] proposed a graph based approach to predict types with similarity learning and parametric type matching. Mir *et al.* [15] uses an approach akin to Allamanis with more data and improved results. Jesse *et al.* [13] uses pre-training to improve sequence tagging of types. This work extends the previous works by introducing a novel training approach and a data set for learning user-defined and rare types. Unlike DIVERSETYPER, existing approaches do not *contextualize* and provide new *associations* for novel developer defined types.

8 Conclusion

DIVERSETYPER presents a test of our hypothesis that user-defined type declarations and the corresponding type annotations can be aligned and used in type predictions. We demonstrated that deep learning models could learn to encode novel class and interface declarations, leverage the learned representations to guess rare and difficult user-defined types, and extend to never before seen types. Finally, we believe that our approach can be applied to other applications of machine-learning to software engineering, where developers can freely proliferate concepts, (*e.g.*, functions, interfaces, classes, generics, exceptions) and thus arbitrarily transcend any vocabulary limits pre-set by machine-learning models.

Acknowledgments

We gratefully acknowledge support from NSF CISE (SHF) Grant No. 1414172 and No. 2107592. We thank Vincent Hellendoorn for his thoughtful review and comments. Additionally, we would like to thank the reviewers for their comments and efforts to refine this manuscript.

References

[1] S. Hanenberg, S. Kleinschmager, R. Robbes, E. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Softw. Engg.*, vol. 19, no. 5, p. 1335–1382, Oct. 2014. [Online]. Available: https://doi.org/10.1007/s10664-013-9289-1

[2] A. Stuchlik and S. Hanenberg, "Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time," in *Proceedings of the 7th Symposium on Dynamic Languages*, ser. DLS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 97–106. [Online]. Available: https://doi.org/10.1145/2047849.2047861

Algorithm 1 Arbiter Algorithm

```
procedure ARBITER($P_{softmax}, E_{user}$)
    $L_{softmax} = ArgSort(P_{softmax})$
    $L_{knn}, D_{knn} = knnSearch(E_{space}, E_{user})$
    $S_{knn} = 1 - D_{knn}$
    $L_{knn\_best}$ = List(); $S_{knn\_best}$ = List()
    for l, d in $L_{knn}, S_{knn}$ do
        if l not in $L_{knn\_best}$ then
            $L_{knn\_best}$.append(l)
            $S_{knn\_best}$.append(d)
        end if
    end for
    $L_{mixed}$ =Reverse($L_{softmax}$)
    $P_{mixed}$ =Reverse($P_{softmax}$)
    for l, d in $L_{knn\_best}, S_{knn\_best}$ do
        $i$ = bisect_left($P_{mixed}$,d)
        $P_{mixed}$ = $P_{mixed}$[:$i$] + d + $P_{mixed}$[$i$:]
        $L_{mixed}$ = $L_{mixed}$[:$i$] + l + $L_{mixed}$[$i$:]
        if $L_{mixed}$.count(l) $> 1$ then
            $i = L_{mixed}$.index(l)
            $P_{mixed}$ = $P_{mixed}$[:$i$] + d + $P_{mixed}$[$i+1$:]
            $L_{mixed}$ = $L_{mixed}$[:$i$] + l + $L_{mixed}$[$i+1$:]
        end if
    end for
    $L_{mixed}$ =Reverse($L_{mixed}$)
    $P_{mixed}$ =Reverse($P_{mixed}$)
    $d = NN(L_{softmax}$[:5]$, P_{softmax}$[:5],
        $L_{knn\_best}$[:5]$, S_{knn\_best}$[:5])
    if $d > .5$ then
        $l_{top1} = L_{knn\_best}$[0]
    else
        $l_{top1} = L_{softmax}$[0]
    end if
    $i = L_{mixed}$.index($l_{top1}$)
    if $i \neq 0$ then
        $L_{mixed} = l_{top1} + L_{mixed}$
        delete($L_{mixed}$[$i$+1])
    end if
    return $L_{mixed}$[:5]
```

[3] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhotak, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundness: A manifesto," *Communications of the ACM*, vol. 58, pp. 44–46, 2015. [Online]. Available: http://cacm.acm.org/magazines/2015/2/182650-in-defense-of-soundness/abstract

[4] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: Quantifying detectable bugs in javascript," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 758–769. [Online]. Available: https://doi.org/10.1109/ICSE.2017.75

[5] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 155–165. [Online]. Available: https://doi.org/10.1145/2635868.2635922

[6] G. Bracha, "Pluggable type systems," 2004.

[7] J. Siek and W. Taha, "Gradual typing for objects," in *European Conference on Object-Oriented Programming*. Springer, 2007, pp. 2–27.

[8] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from" big code"," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015.

[9] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "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*, 2018, pp. 152–162.

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

[11] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "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*, 2020, pp. 209–220.

[12] I. V. Pandi, E. T. Barr, A. D. Gordon, and C. Sutton, "Opttyper: Probabilistic type inference by optimising logical and natural constraints," *arXiv preprint arXiv:2004.00348*, 2020.

[13] K. Jesse, P. T. Devanbu, and T. Ahmed, "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*, 2021, pp. 1483–1486.

[14] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[15] A. M. Mir, E. Latoskinas, S. Proksch, and G. Gousios, "Type4py: Deep similarity learning-based type inference for python," *arXiv preprint arXiv:2101.04470*, 2021.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

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

[18] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: Neural type hints," in *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, pp. 91–105.

[19] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1073–1085. [Online]. Available: https://doi.org/10.1145/3377811.3380342

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[21] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[22] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021.

[23] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," 2020.

[24] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," *arXiv preprint arXiv:2112.02043*, 2021.

[25] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 2021, pp. 610–623.

[26] A. A. Sawant and P. Devanbu, "Naturally!: How breakthroughs in natural language processing can dramatically help developers," *IEEE Software*, vol. 38, no. 5, pp. 118–123, 2021.

[27] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," 2018.

[28] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," 2016.

[29] T. Gong, T. Lee, C. Stephenson, V. Renduchintala, S. Padhy, A. Ndirango, G. Keskin, and O. H. Elibol, "A comparison of loss weighting strategies for multi task learning in deep neural networks," *IEEE Access*, vol. 7, pp. 141 627–141 632, 2019.

[30] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 1735–1742.

[31] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2015.7298682

[32] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," 2018.

[33] L. Liebel and M. Körner, "Auxiliary tasks in multi-task learning," *arXiv preprint arXiv:1805.06334*, 2018.

[34] M. Allamanis, "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*, 2019, pp. 143–153.

[35] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[37] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *nature*, vol. 542, no. 7639, pp. 115–118, 2017.
[38] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.
[39] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv preprint arXiv:1802.05365*, 2018.
[40] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.
[41] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.
[42] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
[43] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill *et al.*, "On the opportunities and risks of foundation models," *arXiv preprint arXiv:2108.07258*, 2021.
[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
[45] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," *arXiv preprint arXiv:2003.10555*, 2020.
[46] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv preprint arXiv:1910.10683*, 2019.
[47] L. H. Li, M. Yatskar, D. Yin, C.-J. Hsieh, and K.-W. Chang, "Visualbert: A simple and performant baseline for vision and language," *arXiv preprint arXiv:1908.03557*, 2019.
[48] C. Sun, A. Myers, C. Vondrick, K. Murphy, and C. Schmid, "Videobert: A joint model for video and language representation learning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 7464–7473.
[49] E. M. Bender and A. Koller, "Climbing towards NLU: On meaning, form, and understanding in the age of data," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 5185–5198. [Online]. Available: https://aclanthology.org/2020.acl-main.463
[50] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" *arXiv preprint arXiv:2108.11308*, 2021.
[51] R. Caruana, "A dozen tricks with multitask learning," in *Neural networks: tricks of the trade*. Springer, 1998, pp. 165–191.
[52] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 160–167.
[53] I. Kokkinos, "Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 6129–6138.
[54] J.-T. Huang, J. Li, D. Yu, L. Deng, and Y. Gong, "Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 7304–7308.
[55] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.
[56] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.
[57] S. Liu, E. Johns, and A. J. Davison, "End-to-end multi-task learning with attention," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 1871–1880.
[58] J.-h. An, A. Chaudhuri, J. S. Foster, and M. Hicks, "Dynamic inference of static types for ruby," *ACM SIGPLAN Notices*, vol. 46, no. 1, pp. 459–472, 2011.
[59] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster, "The ruby type checker," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1565–1572.
[60] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *European Conference on Object-Oriented Programming*. Springer, 2014, pp. 257–281.
[61] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris, "Safe & efficient gradual typing for typescript," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015, pp. 167–180.
[62] "Mypy." [Online]. Available: MyPy.https://github.com/python/mypy/
[63] "Pytype." [Online]. Available: Pytype.https://github.com/google/pytype.
[64] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.

Kevin Jesse is a Ph.D. student at the University of California, Davis with research in the areas of machine learning, natural language processing, and software engineering. He holds a B.S in Computer Engineering and a B.S in Computer Science from the University of California, Santa Cruz and a M.S in Computer Science from the University of California, Davis.

Premkumar T. Devanbu is a Distinguished Professor at the University of California, Davis, ACM Fellow, and co-directs the DECAL Lab. His research is in the areas of empirical software engineering and software engineering applications of machine learning. He was the recipient of the 2021 ACM SIGSOFT Outstanding Research Award.

Anand Sawant is a Postdoctoral scholar at the University of California, Davis working on Machine Learning for Software Engineering. Anand holds a Ph.D. from the Delft University of Technology. He served as Committee Member in the Artifacts-track of ESEC/FSE 2020 and 2021. He was Online Co-Chair organizing committee for ESEC/FSE 2021.