7-2022

Cross-lingual transfer learning for statistical type inference

Zhiming LI

Xiaofei XIE
Singapore Management University, xfxie@smu.edu.sg

Haoliang LI

Zhengzi XU

Yi LI

See next page for additional authors

Cross-Lingual Transfer Learning for Statistical Type Inference

Zhiming Li
Nanyang Technological University
Singapore
zhiming001@e.ntu.edu.sg

Xiaofei Xie*
Singapore Management University
Singapore
xfxie@smu.edu.sg

Haoliang Li
City University of Hong Kong
Hong Kong, China
haoliang.li@cityu.edu.hk

Zhengzi Xu
Nanyang Technological University
Singapore
zhengzi.xu@ntu.edu.sg

Yi Li
Nanyang Technological University
Singapore
yi_li@ntu.edu.sg

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

ABSTRACT

Hitherto statistical type inference systems rely thoroughly on supervised learning approaches, which require laborious manual effort to collect and label large amounts of data. Most Turing-complete imperative languages share similar control- and data-flow structures, which make it possible to transfer knowledge learned from one language to another. In this paper, we propose a cross-lingual transfer learning framework, Plato, for statistical type inference, which allows us to leverage prior knowledge learned from the labeled dataset of one language and transfer it to the others, *e.g.*, Python to JavaScript, Java to JavaScript, etc. Plato is powered by a novel kernelized attention mechanism to constrain the attention scope of the backbone Transformer model such that model is forced to base its prediction on commonly shared features among languages. In addition, we propose the syntax enhancement that augments the learning on the feature overlap among language domains. Furthermore, Plato can also be used to improve the performance of the conventional supervised-based type inference by introducing cross-language augmentation, which enables the model to learn more general features across multiple languages. We evaluated Plato under two settings: 1) under the cross-domain scenario that the target language data is not labeled or labeled partially, the results show that Plato outperforms the state-of-the-art domain transfer techniques by a large margin, *e.g.*, it improves the Python to TypeScript baseline by +14.6%@EM, +18.6%@weighted-F1, and 2) under the conventional monolingual supervised scenario, Plato improves the Python baseline by +4.10%@EM, +1.90%@weighted-F1 with the introduction of the cross-lingual augmentation.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Software notations and tools**.

*Xiaofei Xie is the corresponding author.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534411

KEYWORDS

Deep Learning, Transfer Learning, Type Inference

ACM Reference Format:
Zhiming Li, Xiaofei Xie, Haoliang Li, Zhengzi Xu, Yi Li, and Yang Liu. 2022. Cross-Lingual Transfer Learning for Statistical Type Inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534411

1 INTRODUCTION

Deep learning (DL) has achieved tremendous success in many applications such as image classification and audio recognition. Recently, DL has also been widely applied in software engineering tasks and obtains superior results over the traditional rule-based approaches, such as clone detection [42, 44], code summarization [6, 49], code translation [29], *etc.*

To apply deep learning techniques, large amount of labelled data is required for the training of high-performance neural networks. However, it is well-known that manual labeling of data samples for deep learning is extremely laborious and expensive [28]. It is more challenging for software engineering tasks, since labeling requires considerable domain knowledge. Hence, it would be extremely valuable if we are able to learn models for new languages based on existing labelled data on another language, avoiding the need to invest additional efforts in labelling.

Transfer learning is becoming increasingly popular, where a model developed for a domain is reused as the starting point for training a model for another similar domain. The key purpose of transfer learning is to learn more general features on the data to improve the generalization in another domain. For example, in natural language processing, some techniques [20, 30] have been proposed to transfer the knowledge between two languages (*e.g.*, English and Nepali). Considering the similarities between different programming language, a natural idea is to adapt the model trained from one language to another language based on transfer learning. Although transfer learning has been extensively studied in the fields of computer vision (CV) and natural language processing (NLP), there is still little research on its applications in program analysis tasks.

However, learning from source code is usually more challenging than in other domains such as images and natural languages. Comparing with other tasks, it is more challenging to capture program semantics with deep learning, due to the complex program

structures, *e.g.*, loops, function calls, recursion, and arithmetic calculations. The existing study has shown that DL models learning from programs can easily overfit to some tokens while it is difficult to learn the real program semantics [48]. It is unclear whether the existing transfer learning techniques on CV and NLP can still work well on program domain.

In this paper, we study cross-lingual transfer learning for statistical type inference of optionally-typed programming languages, *i.e.*, adapting the type inference tool trained on programs written in one language to programs in another language. Type inference [5, 23, 38] aims to automatically deduce the type of variables or functions in a dynamic programming language, which is a fundamental program analysis technique used in bug localization, program understanding, reverse engineering and de-obfuscation [17, 22]. There have already been some recent attempts on DL-based type inference of optionally-typed languages [5, 23]. These techniques adopt the mono-lingual supervised learning approach, which works on a given set of labelled data of the same language, while the trained model is known to have limited transferability to other datasets.

Motivated by the fact that the data labeling process for entity types in optionally-typed programming languages is not only labor-intensive but also demands significant expertise knowledge. It is of great potential if we were able to leverage existing labelled dataset from another language to warm start a type inference tool for a new optionally-typed language with scarce data. Notice that most Turing-complete imperative languages share similar control- and data-flow structures (*e.g.*, variable definitions, *if-else* branches, and loops), which makes the transfer of cross-lingual knowledge possible. To this end, we propose Plato, a cross-lingual transfer learning framework, aiming to train type inference models with better transferability (*i.e.*, learn more general features). The key insight of improving transferability is to increase attention on *domain-invariant* features while decreasing attention on *domain-specific* features (*i.e.*, language details). Specifically, we first perform reaching definition analysis to determine how closely related different tokens are in terms of the type inference. This information together with the syntax information in abstract syntax tree is then encoded as a novel *kernelized attention* mechanism, which is used as the backbone of our novel *kernelized model*. The idea is to constrain the attention scope of variables in a code sequence during training. Besides, we apply a syntax enhancement strategy which uses srcML [12] meta-grammar representation to enhance the input representation of the model in order to increase the feature overlap among language domains. Finally, we adopt a κ- bagging ensemble strategy that combines kernelized model and unkernelized model for the inference. It is to compensate the negative effect of kernelized model on language-specific corner cases.

To evaluate the effectiveness and usefulness of Plato, we conducted experiments on three different scenarios. 1) The target language dataset is not labeled. We adopt Plato on two popular optionally-typed programming languages: Python and JavaScript, *i.e.*, to use the model trained from the labelled dataset in one language to make predication on the unlabelled data of another language. We compared Plato with three widely used domain adaptation techniques [16, 20, 40]. 2) The target language dataset is partially labeled. 3) Plato can also be used on the conventional

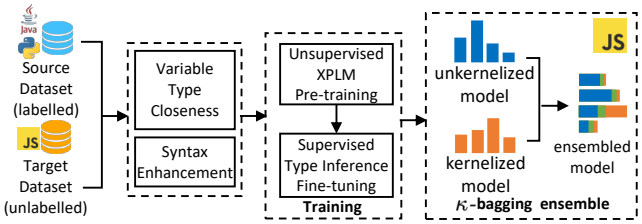


Figure 1: Overview of Plato.

mono-lingual supervised based setting, where more general program semantics could be learned from multiple languages. The results demonstrate that our method significantly outperforms the baseline methods under all settings, *e.g.*, under the first setting, from Python to JavaScript, Plato improves the best domain adaptation baseline performance by +14.6% and +18.6% in terms of EM and weighted-F1. For the second setting, Plato consistently excels the baseline model under all ratios of target domain data. And for the third setting, Plato improves the Python supervised baseline by +4.10%@EM and +1.90%@weighted-F1.

In summary, we made the following contributions.

- We proposed a cross-lingual transfer learning framework for statistical type inference, which is the first of its kind to the best of our knowledge. The framework is powered by the kernelized attention mechanism capturing variable type relations and the syntax enhancement techniques to improve the transferability of the model.
- We demonstrate the feasibility of exploiting the similarity/transferability between different languages in supporting cross-lingual program analysis tasks. Our work opens up new opportunities for a wide range of learning-based approaches to be further studied in the future, especially to apply transfer learning in software engineering tasks with multiple languages.
- We conducted extensive experiments to demonstrate the usefulness and effectiveness of our approach on real-world datasets. The results show that Plato significantly outperforms other domain adaptation techniques as well as traditional rule-based models.
- We demonstrate that Plato can also be used to improve the performance of supervised based methods based on the cross-language augmentation. Specifically, by learning more general features among languages, the model can mitigate the overfitting issue in traditional mono-lingual supervised based methods.
- We have made our tool and data available on our website [2].

2 METHODOLOGY

In this section, we present our framework Plato for cross-lingual transfer learning of statistical type inference in detail.

2.1 Overview

Figure 1 gives an overview of our Plato framework, which consists of four major parts: (1) variable type closeness matrix extraction, (2) syntax enhancement, (3) training and (4) ensemble-based inference. The inputs to our system include the source code sequence, its

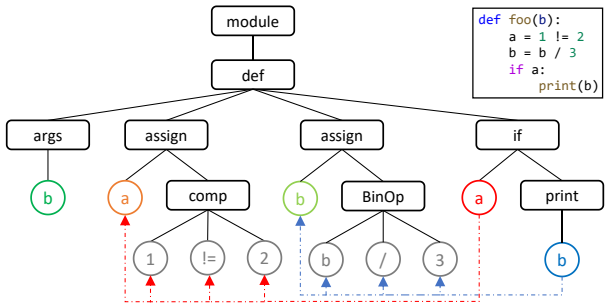


Figure 2: Type-closeness graph of the sample code.

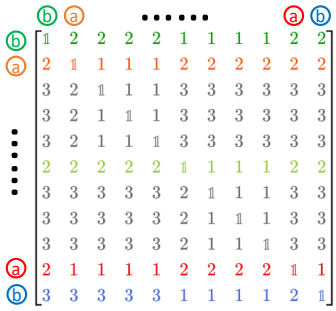


Figure 3: Variable type closeness adjacency matrix A_K^Q obtained from program shown in Figure 2.

corresponding srcML meta-grammar sequence and variable type closeness matrix. The output is the trained model that can predict the corresponding type annotations for each token in the given code sequence.

Given an optionally-typed language, our key insights in achieving cross-lingual transfer learning of type inference are to *exploit the task-relevant features common to the type systems of different programming languages, and to reduce the impact of the irrelevant features*. For example, the def-use relationship between variables has a strong connection to their types, which can be assumed as a common knowledge in many programming languages. For the simple code snippet, "`a = 1!=2; ... print(a)`", we can infer the type of the variable "a" in the "`print`" statement as Boolean, based on the first statement "`a = 1!=2`" where "a" was previously defined. Such knowledge may seem trivial, but is difficult for deep learning-based models to pick up without prior knowledge.

To obtain such knowledge, we first perform a reaching definition analysis [4] on each program for both the source and target dataset. For each program, based on the result of reaching definition analysis, we define a measurement (*i.e.*, an adjacency matrix) using graph kernel, which we call *Variable Type Closeness (VTC)* (see Definition 2.3), to quantify the closeness of different tokens in a code sequence in terms of types. During training, instead of learning with the traditional attention without constraint, we use the kernelized attention based on VTC in order to regularize model to focus on the most relevant features for type inference. In this way, the trained model constrains the attention scope of a token only to the tokens related to its type in the sequence and eliminates those that are irrelevant, therefore it can decrease the negative effect (noise) of the irrelevant features which hinder the transferability.

We further propose a syntax enhancement strategy which is to augment the input representation with srcML meta-grammar [15, 33]. With srcML meta-grammar representation, features shared between different language domains are augmented such that common semantics can be learnt.

One problem is that the kernelized attention model may not be perfect in some predictions. For example, it may overfit to some language-specific features that are mismatched with target languages (see Section 2.5). To mitigate this challenge, we propose an ensemble-based strategy that combines the kernelized model and un-kernelized model (*i.e.*, attention model learned from code sequence directly without being constrained by kernel) during the inference. With such an ensemble strategy, the kernelized and unkernelized models complement each other and produce better results.

2.2 Variable Type Closeness

In this part, we introduce the concept of variable type closeness and how it is derived from graph kernel.

Kernelized Attention. For traditional attention mechanism, the embedding of a word depends on its relations with all the other words in an input sequence, *i.e.*, there is no constraint to its attention scope during gradient-based learning. For example, consider a code sequence "`var a := true; var b := 0`", when calculating embedding for the token "a", the traditional attention takes all the other tokens in the input sequence into consideration. Yet, when performing type inference for "a", the statement "`var b := 0;`" is irrelevant and should not be considered when making the prediction. On the other hand, if the prediction is erroneously based on "b" or "`0`", the model would hardly generalize. Therefore, we propose a kernelized attention mechanism, which uses a shortest-path graph kernel [8] to constrain the attention scope of tokens in code sequence. In this way, given a query token, the model tends to use the set of tokens that are more useful for the type inference, *i.e.*, the closest tokens in the closeness graph for prediction.

To define such a graph kernel, we first introduce a *type-closeness graph* data structure and define a distance measurement based on the graph. We then present how to derive the *variable type closeness* adjacency matrix with an example.

Type-Closeness Graph. Intuitively, a type closeness graph (TCG) is an annotated AST with extra RDA edges derived from reaching definition analysis on the Control Flow Graph (CFG).

Definition 2.1 (Type-Closeness Graph). A *type-closeness graph* is a graph $G = (V, N, E_{AST}, E_{RDA})$, where V and N are terminal and non-terminal nodes from the AST, respectively, E_{AST} are AST edges, and E_{RDA} contains edges between pairs of terminal nodes $v_i, v_j \in V$ if and only if v_i is within a reaching definition of v_j on the control flow graph.

An example TCG is shown in Figure 2, where the circles represent terminal nodes V, the rectangles represent non-terminal nodes N, and the solid (dashed) lines represent E_{AST} (E_{RDA}).

Type-Closeness Distance. The *type closeness distance* (TCD) is a distance measure $d(\cdot,\cdot)$ defined over the *type closeness graph*. The smaller the TCD between the target token v_i and the token v_j, the more important the token v_j is for the type inference of v_i.

Definition 2.2 (Type-Closeness Distance). For a pair of terminal nodes $v_i, v_j \in V$, the type-closeness distance from v_i to v_j is defined as $d(v_i, v_j) = \min(d_{LCA}(v_i, v_j), d_{RDA}(v_i, v_j))$, where d_{LCA} and d_{RDA} are the lowest common ancestor (LCA) distance and the reaching definition distance, respectively.

The LCA-distance from v_i to v_j is defined as the length of the shortest path between v_i and the lowest common ancestor [3] of v_i and v_j. More formally,

$$d_{LCA}(v_i, v_j) = d_{AST}(v_i, LCA(v_i, v_j)), \tag{1}$$

where $LCA(\cdot,\cdot)$ denotes the lowest common ancestor of two nodes and $d_{AST}(\cdot,\cdot)$ denotes the distance of the path between two nodes on the AST. For example, as shown in Figure 2, consider node b and node a, their lowest common ancestor is the non-terminal node `if`, and it takes two hops from node b to reach node `if`, therefore $d_{LCA}(b, a) = 2$. Intuitively, with the d_{LCA}, a token is closer to another token within the same statement, compared with other tokens from other statements.

Next, we introduce d_{RDA} that captures the def-use relations between tokens. Specifically, given a variable node v_i, we hard-wire it to the set of nodes $V_D = \{v | (v_i, v) \in E_{RDA}\}$ that comprise its reachable definition statement. For example, the dashed lines in Figure 2 illustrates the RDA edges of node b. The RDA-distance d_{RDA} from v_i to all $v_j \in V_D$ is defined to be 1, while for others that are unreachable, the distance are set to be $+\infty$. More formally,

$$d_{RDA}(v_i, v_j) = \begin{cases} 1, & if\ (v_i, v_j) \in E_{RDA} \\ +\infty, & otherwise \end{cases} \tag{2}$$

Finally, given two nodes v_i, v_j in the TCD space, the type-closeness distance from v_i to v_j is defined as the minimum of their LCA distance and RDA distance: $d(v_i, v_j) = \min(d_{LCA}(v_i, v_j), d_{RDA}(v_i, v_j))$.

Variable Type Closeness. Based on the TCD distance measurement, we derive the *variable type closeness* adjacency matrix, which is used as an input to our model to regularize its learning behavior.

Definition 2.3 (Variable Type Closeness). Given a code sequence $\mathbf{x}$, for each token $t \in \mathbf{x}$, the *variable type closeness* vector of t, denoted as $\mathbf{A}^{\mathbf{t}}_{\mathbf{x}}$, is defined as a distance vector that consists of the distance of t from all the tokens in $\mathbf{x}$ under the TCD defined space, *i.e.*, $\mathbf{A}^{t}_{\mathbf{x}} = [d(t, t')]_{t' \in \mathbf{x}} \in \mathbb{R}^{1 \times |\mathbf{x}|}$. Then by stacking the distance vectors of all tokens within $\mathbf{x}$, forms the *variable type closeness* adjacency matrix of sample $\mathbf{x}$: $\mathbf{A}_{\mathbf{x}} \in \mathbb{R}^{|\mathbf{x}| \times |\mathbf{x}|}$.

Figure 3 shows the *variable type closeness* (VTC) adjacency matrix derived from the TCG graph of the example program $\mathbf{x}$ shown in Figure 2. The *variable type closeness* distance vector of b: $\mathbf{A}^{b}_{\mathbf{x}}$ is illustrated in the last row of the matrix. For example, the LCA of b and b is `def`, which takes three hops to reach from b through E_{AST}: $d_{LCA}(b, b) = 3$, and there is no RDA edge that connects them: $d_{RDA}(b, b) = +\infty$, thus the first element of its distance vector $\mathbf{A}^{b}_{\mathbf{x}}[0] = \min(d_{LCA}(b, b), d_{RDA}(b, b)) = 3$; and in order to reach 3 in the definition statement, it takes one hop from b through the E_{RDA}: $d_{RDA}(b, 3) = 1$ (illustrated as dashed lines), while $d_{LCA}(b, 3) = 3$, thus $\mathbf{A}^{b}_{\mathbf{x}}[8] = \min(d_{LCA}(b, 3), d_{RDA}(b, 3)) = 1$.

Table 1: Subset of unified srcML elements.

	Js	*Py*	*Java*	*srcML*
Function definition	function	def	NA	def
Equality	===	==	==	==
Non-equality	!==	!=	!=	!=
Logical AND	&&	&	&, &&	&amp
Logical OR	||	|	|, ||	|
Exception	throw	raise	throws	throws

2.3 Syntax Enhancement

For human programmers who manage to master one language, it is relatively easy to switch to another, because many reserved keywords and operators share the same syntactic and semantics roles across different language domains. Deep learning models are hard to exploit this similarity easily, hence limiting the transferability. To address this, we propose a strategy to augment the syntactic feature overlap shared among different language domains by incorporating a srcML meta-grammar embedding into the input representation beyond the source code embedding. At the high level, srcML meta-grammar provides each token in a code sequence with a corresponding markup tag that represents the abstract syntax role of that token which is unified among languages. Table 1 shows a subset of the unified tags provided by srcML. Specifically, given a code sequence $\mathbf{x} = (x_1, ..., x_n)$ and its corresponding srcML sequence $\mathbf{s} = (s_1, ..., s_n)$, we map them to their respective embedding $emb(\mathbf{x}) = (emb(x_1), ...emb(x_n))$ and $emb(\mathbf{s}) = (emb(s_1), ...emb(s_n))$. Then the augmented input representation $\mathbf{c}$ is the weighted sum of $emb(\mathbf{x})$ and $emb(\mathbf{s})$. Formally:

$$\mathbf{c} = emb(\mathbf{x}) \odot \alpha + emb(\mathbf{s}) \odot \beta \tag{3}$$

where α and β are weight vectors for $\mathbf{x}$ and $\mathbf{s}$ respectively, and $\odot$ denotes element-wise multiplication. In our work, we used srcML to extract the meta grammar representation for Java, and since srcML does not support Python and TypeScript, we implement an approximate meta grammar mapping for the two optionally-typed languages on our own. Our empirical results (see Section 3.3) demonstrate that the syntax enhancement technique is useful in improving the transferability across domains.

2.4 Training

In this work, we use BERT since it is shown that BERT based model can achieve state-of-the-art performance by leveraging self-supervised pre-training [26]. Specifically, we use a two-stage training mechanism following [13, 43]: (1) self-supervised cross programming language model (XPLM) pre-training, and (2) supervised type inference fine-tuning.

2.4.1 Unsupervised XPLM Pre-Training. In this phase, we use data from multiple language sources to pre-train the XPLM model. As shown in the model architecture in Figure 4, during the self-supervised pre-training stage, for each code sequence sample $\mathbf{x}$, the XPLM backbone model receives two inputs: namely, its corresponding augmented input vector $\mathbf{c}$ and *variable type-closeness* adjacency matrix $\mathbf{A}_{\mathbf{x}}$. The detailed formulation of the model is given in Equation (4).

$$\kappa\text{-}\mathbf{emb}(\mathbf{x}_i) = \mathbf{g}_{\sigma}(\mathbf{A}_{\mathbf{x}}) \odot attn(\mathbf{c}_i; \mathbf{c}) \cdot \mathbf{c} \tag{4}$$

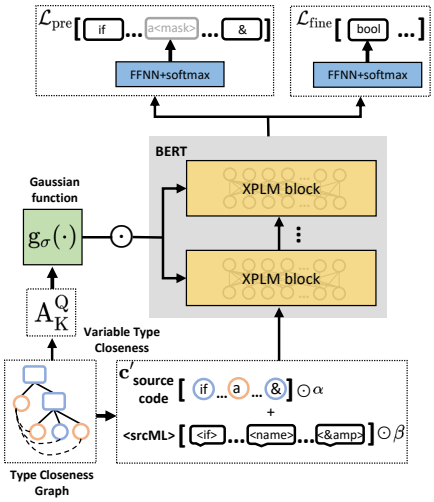


Figure 4: Model architecture.

Algorithm 1 κ- bagging BERT

Input:
submodels:
S = {unkernelizedBERT : BERT, kernelizedBERT : κ-BERT};
dataset: D = $\{x_1, x_2, ..., x_n\}$;
confidence threshold: θ;
combination weight: λ;

$D' \leftarrow \emptyset$
for $i \leftarrow 1, 2, ..., |D|$ **do**
  obtain logit of each sample from each model:
  $h_{\text{BERT}} \leftarrow \mathbb{1}^{\theta}[\text{BERT}(x_i)]$
  $h_{\kappa\text{-BERT}} \leftarrow \kappa\text{-BERT}(x_i)$
  $h_{\text{ensemble}} \leftarrow \lambda \cdot h_{\kappa\text{-BERT}} + (1-\lambda) \cdot h_{\text{BERT}}$
  $D' \cup \{\frac{1}{|S|}\text{argmax } h_{\text{ensemble}}\}$
end for
Output:
output ensembled predictions: D′

$\mathbf{c}_i$ is the augmented vector of a token $\mathbf{x}_i$ in the sample code sequence $\mathbf{x}$. We first obtain the attention vector $attn(\mathbf{c}_i; \mathbf{c}) \in \mathbb{R}^{1\times|\mathbf{x}|}$ of $\mathbf{c}_i$ w.r.t all the vectors in $\mathbf{c}$. Then we constrain the attention by taking element-wise multiplication of $attn(\mathbf{c}_i; \mathbf{c})$ with a regulatory weight vector $\mathbf{g}_\sigma(\mathbf{A}_x)$, where $\mathbf{g}_\sigma(\cdot)$ is a radial basis function kernel [41] parameterized by a learnable parameter σ. Intuitively, the more distant two tokens are in the TCD defined space, the smaller their regulatory weight is. In this way, the model is constrained from using tokens that are irrelevant for embedding. Finally, by taking dot product with $\mathbf{c}$, we obtain the kernelized attention embedding for token $\mathbf{x}_i$.

The model is then pre-trained with masked language model (MLM) loss, next sentence prediction (NSP) loss [13] together with a regularization loss of σ. The regularization loss is used to constrain the attention scope from getting large during training. The detailed loss is as follows:

$$\begin{aligned}\mathcal{L}_{\text{pre}} &= \mathcal{L}_{\text{MLM}} + \mathcal{L}_{\text{NSP}} + \mathcal{L}_{\sigma} \\ &= \sum_{x_{[\text{MASK}]} \in \mathbf{x}_{[\text{MASK}]}} -x_{[\text{MASK}]} \log[P(x_{[\text{MASK}]} | \mathbf{x} \backslash x_{[\text{MASK}]})] \\ &+ \sum_{(s_i, s_j) \sim S \times S} -y_{ij} \log[P(\hat{y_{ij}} | [s_i : s_j])] + \lambda\sigma^2\end{aligned} \tag{5}$$

where $\mathbf{x}_{[\text{MASK}]}$ is a set of randomly sampled masked tokens in a sample $\mathbf{x}$, (s_i, s_j) is a pair of randomly selected code sequence and y_{ij} is a binary label indicates whether they follow each other, $\hat{y_{ij}}$ is the model's predicted probability, λ is a hyper-parameter chosen on a validation set.

2.4.2 Supervised Type Inference Fine-Tuning. After obtaining a pre-trained language model from the self-supervised pre-training stage. We fine-tune this model on our downstream type inference task in a supervised manner. In this supervised learning phase, we have the labelled source language samples S and the labelled target language samples T. Note that, the number of T is usually small or zero, indicating that we have little or no labelled target language data.

The input of the supervised fine-tuning stage is the same as the pre-training stage, shown in Figure 4. To allow the model making prediction, we attach a linear layer (FFNN+softmax in Figure 4) after the last hidden layer of the pretrained XPLM to predict the types for each tokens. We fine-tune all the parameters in the model with a classification loss on the labelled parallel corpus of code sequence and type annotations. Specifically, the fine-tune loss function is as follows:

$$\begin{aligned}\mathcal{L}_{\text{fine}} &= (1 - f(\gamma)) \cdot \mathcal{L}^S_{\text{fine}} + f(\gamma) \cdot \mathcal{L}^T_{\text{fine}} \\ &= (1 - f(\gamma)) \sum_{(\mathbf{x_i}, \mathbf{y_i}) \in S} -\mathbf{y_i} \log[P(\hat{\mathbf{y}}_\mathbf{i} | \mathbf{x_i})] \\ &+ f(\gamma) \sum_{(\mathbf{x_j}, \mathbf{y_j}) \in T} -\mathbf{y_j} \log[P(\hat{\mathbf{y}}_\mathbf{j} | \mathbf{x_j})],\end{aligned} \tag{6}$$

where $\mathcal{L}^S_{\text{fine}}$ and $\mathcal{L}^T_{\text{fine}}$ are the negative log likelihood loss for samples that are from the source domain and the target domain, respectively. $P(\hat{\mathbf{y}}_\mathbf{i} | \mathbf{x_i})$ and $P(\hat{\mathbf{y}}_\mathbf{j} | \mathbf{x_j})$ denotes the output probability distribution over the possible type classes for source language sample $\mathbf{x_i}$ and target language sample $\mathbf{x_j}$. We embrace a decay training scheme [16], $f(\gamma) = \frac{2}{1+\exp(-\gamma)} - 1$ is a regularization term used to control the weight for the loss of the source and target language samples, $\gamma \in [0, 1]$ is the training process. Intuitively, when the size of both dataset are imbalanced(*i.e.*, $|S| >> |T|$), we force the model to gradually focus more on the target domain data and less on the source data during the training process in order to mitigate the imbalance problem.

Note that, S usually represent the full source language data that has been labeled, but the size of T could be changed. Based on the size of T, we define different scenarios:

- $|T| = 0$ *and the model is trained to predict on the target language.* In this setting, all the target training data is not labelled and we conduct the unsupervised cross-lingual domain adaptation from only source language.
- $|T|$ *is a small number and the model is trained to predict on the target language.* In this setting, a small part of target training data (*i.e.*, partially) is labelled and we conduct the cross-lingual transfer learning from source language data as well as the given target language data.

```
Js Sets   :  var foo = new Set([1, 2, 3])
Py Sets   :              foo = {1, 2, 3}
Py Lists  :              foo = [1, 2, 3]
```

Figure 5: An example kernel corner case.

- *$|T|$ is a small number and the model is trained to predict on the source language.* This setting corresponds to the common supervised based learning on the source language S. The difference is that we also have some labeled training data with other languages (*i.e.*, T) beyond the full labeled source language training set. Here, PLATO considers T as the augmented data (*i.e.*, the cross-language augmentation) and trains a model for the type inference on the language S.

2.5 Ensemble-Based Inference

While the kernelized model is able to use explicit syntactic and semantic relations to improve the performance of type inference, it may fail to cover some corner cases. For example, features within the kernelized attention scope may be language-specific, thus do not generalize to other language domains and lead to negative transfer. As shown in Figure 5, if we use Python as the source language and TypeScript as the target language, by using the graph kernel, the XPLM would be constrained to leverage "`{,}`" and "`[, ]`" as primal features to classify Python *Sets* and *Lists*. However, when applied to TypeScript *Sets* sample, the kernelized model would potentially leverage "`[, ]`" which results in erroneously classifying the variable"`foo`" into *Lists* instead of *Sets*. To this end, we propose an ensemble strategy which combines the kernelized and unkernelized model during inference stage such that the combined model can make the best of both worlds.

Algorithm 1 shows the detail of the ensemble strategy, which we call κ- bagging. Specifically, it is a bagging-based regression ensemble strategy [9]. Given two submodels, the unkernelized model BERT and the kernelized model κ-BERT, we first pass the sample in the test set through both the kernelized and unkernelized models to obtain their corresponding output probability distribution $\mathrm{h}_{\mathrm{BERT}}$ and $\mathrm{h}_{\kappa\text{-BERT}}$ (Line 10-11). Then we apply an indicator function $\mathbb{1}^{\theta}$ on $\mathrm{h}_{\mathrm{BERT}}$ which returns the original probability distribution vector if its maximum value is larger than the confidence threshold θ otherwise it returns a zero vector of the same size, such that the ensemble model only uses the prediction of the unkernelized model when it is confident enough. Finally, the output ensemble distribution $\mathrm{h}_{\mathrm{ensemble}}$ is the weighted sum of $\mathrm{h}_{\mathrm{BERT}}$ and $\mathrm{h}_{\kappa\text{-BERT}}$ using a combination weight $\lambda \in [0, 1]$ (Line 12). θ and λ are hyper-parameters empirically selected on the validation set.

3 EVALUATION

We have implemented PLATO based on PyTorch with about 6K lines of code.[1] To demonstrate the effectiveness and usefulness of PLATO in the cross-lingual type inference task, we evaluate under three settings (see Section 2.4.2): (1) no labeled target language data available (**NTL**), (2) partial labeled target language data available (**PTL**) and (3) the supervised learning on source language data (**SL**). Specifically, we study the following research questions.

- **RQ1:** How effective is PLATO compared with other domain adaptation techniques without any labeled target language data?
- **RQ2:** How do different components of PLATO affect the results?
- **RQ3:** How effective is PLATO when partial labeled target language data available?
- **RQ4:** How useful is PLATO in improving the supervised based methods by introducing cross-domain knowledge?

3.1 Experimental Setup

3.1.1 Dataset Preparation. In our experiments, we selected three languages including two optionally-typed languages (*i.e.*, Python and TypeScript) and one strongly-typed language (*i.e.*, Java). Specifically, for TypeScript, we used a TypeScript dataset [23] provided by Hellendoorn *et al.* We generated the corresponding AST of each sample using Esprima [1]. For Python, we used the dataset provided by Allamanis et al. [5] and extracted corresponding ASTs from their self-defined graphs. For Java, we used the CodeSearchNet dataset [25] and extracted the ASTs using JavaParser and extracted the type annotations for variables, function parameters, and return types using srcML [10]. After eliminating the samples that cannot be parsed, we collected 13,248 Python programs with 325,129 variables, 28,587 TypeScript programs with 977,072 variables, and 9,126 Java programs with 100,869 variables, all programs are at function-level. We pre-train the backbone XPLM model on a large multi-lingual corpus that consists of 1.1M datapoints across the above mentioned three languages collected from GitHub.

Label Calibration. The sets of types for different languages may vary. For example, there are 13,491, 15,050, and 4,108 types in the TypeScript, Python, and Java datasets, respectively. To facilitate transferability, we need to calibrate the types such that the labels in the training samples (*e.g.*, the source language) and the test samples (*e.g.*, the target language) have the same labels if they have similar functionalities and data structures. Specifically, we have the following configurations:

(1) For **RQ1** and **RQ2**, we assume there is no any labeled target language data and aim to evaluate the transferability of type system among different languages. Following the similar setting in computer vision and natural language processing domains, we relabel the datasets such that the source language and the target language have the same co-domain labels set. Specifically, we mainly consider the commonly-used types in both source and target languages. We consider 7 meta-types in Python and Java, *i.e.*, *boolean*, *integer*, *float*, *bytes*, *string*, *list*, and *dict*. For TypeScript, there is only one numeric type: *number*, thus we consider 5 meta-types *boolean*, *number*, *string*, *list*, and *dict*.

(2) For **RQ3**, to simulate the real life scenario, we assume that there some target language data that have labels. Here, we consider a more practical setting by using all types in both source language and target language. Suppose T_s and T_t are the set of types in the source language data and the target language data, respectively, the co-domain types we used are $T_s \bigcup T_t$.

(3) For **RQ4**, we compared PLATO with the state-of-the-art supervised based techniques by adding some cross-language data. The

[1] The implementation details and more results can be found on the website [2].

Table 2: The comparative results with different methods on overlapped types.

Methods	Python → TypeScript		Java → TypeScript		TypeScript → Python		Java → Python	
	EM	weighted-F1	EM	weighted-F1	EM	weighted-F1	EM	weighted-F1
TAPT	0.601	0.546	0.608	0.587	0.518	0.483	0.496	0.437
MMD	0.574	0.550	0.595	0.571	0.557	0.503	0.512	0.426
ADV	0.530	0.501	0.560	0.540	0.540	0.504	0.512	0.435
Supervised$_i$	0.552	0.503	0.513	0.512	0.499	0.484	0.404	0.420
PLATO	**0.747**	**0.736**	**0.736**	**0.704**	**0.588**	**0.579**	**0.532**	**0.473**
Improvement (Δ)	14.6%	18.6%	12.8%	11.7%	3.10%	7.50%	2.00%	3.60%
Supervised$_o$	TypeScript → TypeScript				Python → Python			
	0.866		0.869		0.723		0.711	

expectation is that, by introducing the cross-domain knowledge, the trained model by PLATO can learn more general features from multiple domains. Specifically, we aim to conduct the type inference on the labeled dataset S by augmenting the monolingual data with another small labeled dataset T that uses a different programming language from S.

3.1.2 Evaluation Metrics. The data distribution of the type system is imbalanced, *e.g.*, *string* takes up a much larger proportion than all the other types in all languages. Therefore, the widely used *Exact Match* (EM) [5, 23] is suboptimal, because when using EM, a weak classifier that is biased towards predicting types with the highest occurrences in the training set could still get spuriously good result. Therefore, we use weighted-F1 to account for the precision and recall trade-off as well as the data imbalance. Formally, weighted-F1 calculates the F1-score for each class and takes their average weighted by support:

$$\text{weighted-F1} = \sum_{i \in C} \frac{|C_i|}{|C|} \text{F1-score}_i, \tag{7}$$

where $|C_i|$ is the size of class C_i, and $|C|$ is the size of the entire dataset. In our evaluation, we report both the EM values and the weighted-F1 scores.

3.1.3 Configurations. We used a BERT [13] encoder with 4 stacked attention layers, 4-headed attention as the backbone XPLM. The dimension of all the token embedding is 256. We train the models using Adam optimizer [27] with a initial learning rate of 10^{-4}. All models are fine-tuned for 30 epochs and early stopped if the validation performance does not improve for 10 consecutive steps. For the inputs, we truncate the input sequence at a maximum length of 700 in order to fit in the memory. We conducted all experiments on a Ubuntu 16.04 server with 24 cores of 2.2GHz CPU, 251GB RAM and two GeForce RTX 3090 GPU with a total of 48GB memory.

3.2 RQ1: Comparison with Baselines When No Labeled Target Data Is Available

Baselines. To evaluate the transferability among different languages and effectiveness of our method under the NTL setting, we compared PLATO with three popular domain adaptation methods, which are widely used in text and image classification tasks [16, 20, 40].

TAPT. We adopted the Task-Adaptive Pre-Training (TAPT) [20, 24], which leverages the task-relevant data to adapt the pretrained backbone model to specific downstream domain, as a baseline. Specifically, TAPT utilizes the unlabeled task-specific samples from both the source and target domain to further fine-tune the pretrained language model such that it is much more task- and domain- relevant. In this work, we use the whole unlabeled corpus from both the source and target programming languages to adapt the XPLM.

MMD. We adopted Maximum Mean Discrepancy (MMD) [18, 40] as the second baseline domain adaptation method. The key idea of MMD is to minimize the latent feature discrepancy between the source and the target domains such that they become indistinguishable by the model. Concisely, MMD attaches a discrepancy loss term on the last hidden layer of the backbone model and maximizes the loss during the type inference training phase.

ADV. For the third baseline, we adopted adversarial domain adaptation [11, 16]. ADV transfers knowledge from the source to the target domain by using reversed gradient drawn from domain classification loss to confuse the features from the source and target domains. Specifically, ADV introduces a gradient reversal layer on top of the last hidden layer of the backbone model. A domain classifier is used to distinguish the domain of samples. The updated gradient from the domain classifier is reversed by the gradient reversal layer before being used to update the model.

Setting. We randomly split the target language dataset into train-validation-test sets in 70-15-15 proportions. The validation set is used for the choice of hyper-parameters and the test set is for the model evaluation.

In addition to the three baselines, we also calculate the results with supervised learning as reference. Specifically, we train a classifier with supervised learning on the in-domain source dataset (denoted as Supervised$_i$) and then use the classifier to evaluate the out-domain dataset without any domain adaptation techniques, which can be regarded as the lower bound of the domain adaptation techniques. On the other hand, we adopt supervised learning to train another classifier on the out-domain target dataset and evaluate it on the out-domain dataset (denoted as Supervised$_o$), which can be regarded as the upper bound. For the baseline methods, the loss weight for MMD and ADV are set to be 0.1.

Results. Table 2 shows the detailed results of different methods in terms of exact match and weighted-F1. Columns show the transfer results from different source language domains to different target

Table 3: Results on the impact of different components.

Methods	Python → TypeScript		TypeScript→ Python	
	EM	weighted-F1	EM	weighted-F1
w/o SE	0.639	0.610	0.531	0.509
w/o VTC	0.721	0.717	0.554	0.555
-Kernel	0.734	0.720	0.573	0.561
-Sequence	0.721	0.717	0.554	0.555
PLATO	0.747	0.736	0.588	0.579

language domains. For example, for the domain adaptation between optionally-typed languages, column "Python → TypeScript" shows the cross-lingual transfer results from Python to TypeScript. We also included the results of using the strongly-typed language (*i.e.*, Java) as the source language, which represents the scenario that if we do not have an existing labeled optionally-typed language dataset, we can use the strongly-typed language data as the source because their types can be obtained automatically.

Overall, the results demonstrate that our method significantly outperforms the three domain adaptation techniques in terms of all metrics using either optionally-type language (TypeScript/Python) or strongly-typed language (Java) as the source language. Row "Improvement (Δ)" shows the improvement of PLATO over the best result of the baselines. Specifically, from Python to TypeScript, the performance of exact match and weighted-F1 is increased by +14.6% and +18.6%, respectively. From TypeScript to Python, it is increased by +3.1% and +7.5%, respectively. When using Java as the source language: from Java to TypeScript, the results are improved by +12.8%@EM, +11.7%@weighted-F1; from Java to Python, the results are improved by +2.0%@EM, +3.6%@weighted-F1. Interestingly, by using strongly-typed language Java as the source language, we can achieve comparative performance compared with using optionally-typed language as source language. Furthermore, we compare PLATO with the rule-based type inference tools and it is shown that PLATO achieve comparative or even better performance. *E.g.*, CheckJS[2] achieves 79.0%@EM, 69.9%@weighted-F1 while PLATO manages to achieve 74.7%@EM, 73.6%@weighted-F1; Pytype[3] achieves 12.1%@EM, 17.5%@weighted-F1 while PLATO achieves 58.8%@EM, 57.9%@weighted-F1. The results show the transferability of the trained model among languages. With PLATO, one can achieve comparative or even better performance by using cross-lingual labeled data instead of implementing rule-based tool from scratch that requires significant manual effort and expert knowledge.

Consider the results of supervised learning baseline, not surprisingly, Supervised$_i$ performs the worst on the out-domain data due to that it does not have any knowledge of the out-domain target data. Consider Supervised$_o$, we observe that, although our technique has already achieved the best result in the domain adaptation setting, there is still a gap with the supervised learning setting which leaves room for future progress.

[2]https://www.typescriptlang.org/tsconfig/checkJs.html
[3]https://github.com/google/pytype

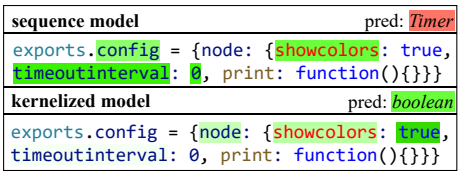


Figure 6: Illustrative example of kernelized model compared with original sequence model. The first and second row denote their attention vectors of variable *showcolors*.

Answer to RQ1: PLATO can significantly outperform the state-of-the-art domain adaptation methods and rule-based tools on NTL, *i.e.*, the target language dataset is not labeled.

3.3 RQ2: Usefulness of Different Components

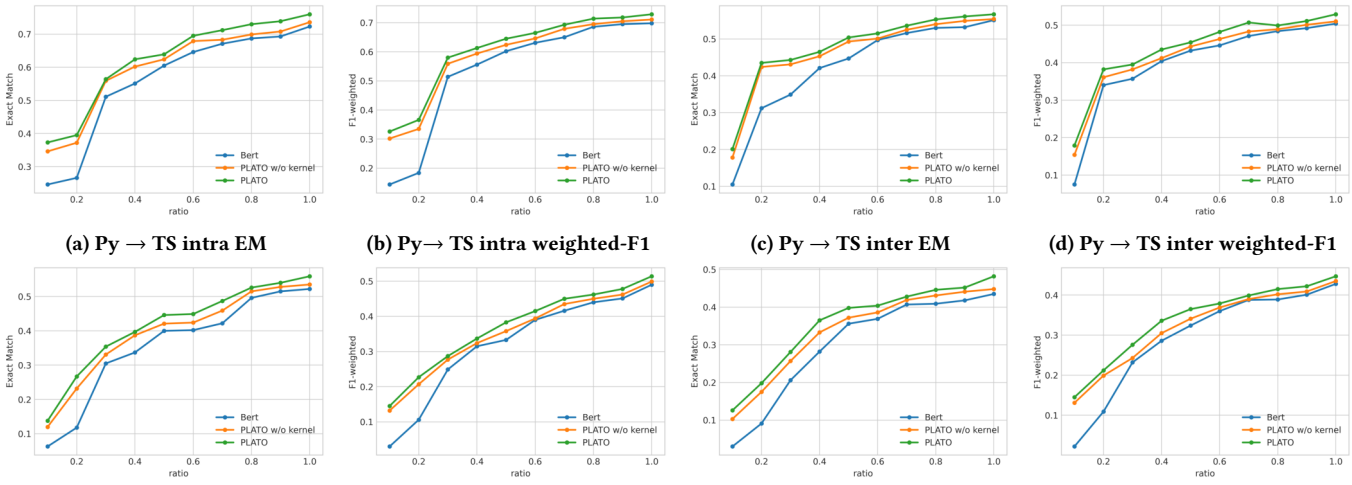
Setup. In this section, we perform an ablation study to study the contribution of different components of our method in the results of *RQ*1. We build the following baselines to evaluate each component:

- **PLATO without syntax enhancement** (*w/o SE*). We remove the component of syntax enhancement and let the neural network to learn the syntax mapping (*e.g.*, different keywords) itself.
- **PLATO without VTC-based Kernelized Attention** (*w/o VTC*). We remove the VTC-based kernelized attention to evaluate its effect.
- **Ensemble Inference**. To evaluate the usefulness of κ- bagging strategy, we use the unkernelized model (*Sequence*) and the kernerlized model (*Kernel*) to perform the inference separately.

Usefulness of Syntax Enhancement. As shown in (Row *w/o SE*) of Table 3, the performance is significantly reduced compared with PLATO. Specifically, after removing the meta-grammar representation, from Python to TypeScript, the results drop by 10.8%@EM, 12.6%@weighted-F1; while for TypeScript to Python, the results drop by 5.7%@EM, 7.0%@weighted-F1. The results indicate that by enhancing the input representation with srcML meta-grammar representation, the overlapped features among language domains are significantly increased thus improve the transferability of the model.

Usefulness of Variable Type Closeness. Consider the results in Row *w/o VTC*, we found that the performance decreases in each task. Note that when removing VTC from the model, it degenerates into using the code sequence without the kernelized attention. Specifically, without VTC, from Python to TypeScript, the exact match and weighted-F1 are decreased by 2.6% and 1.9%, respectively. From TypeScript to Python, the performance is decreased by 3.4% and 2.4%, respectively. It demonstrates the usefulness of the VTC-based kernelized attention strategy.

We provide a case study to further demonstrate the usefulness of the VTC-based kernelized attention in Figure 6. Specifically, we visualize the final layer attention vector of the boolean variable "`showcolors`" for both the original sequence model and the kernelized model. For the sequence model, it is shown that model spuriously leverages the irrelevant token "`timeoutinterval`" for prediction while completely ignoring the ground-truth evidence "`true`".



(a) Py → TS intra EM (b) Py→ TS intra weighted-F1 (c) Py → TS inter EM (d) Py → TS inter weighted-F1

(e) TS → Py intra EM (f) TS → Py intra weighted-F1 (g) TS → Py inter EM (h) TS → Py inter weighted-F1

Figure 7: The evaluation results when partial labeled data is available.

Table 4: Comparison of PLATO's performance with baseline methods on TypeScript dataset

Method	intra-project		inter-project	
	EM	weighted-F1	EM	weighted-F1
LambdaNet	0.646	0.623	0.535	0.481
Transformer	0.695	0.654	0.532	0.484
TypeBert	0.723	0.698	0.551	0.504
PLATO	**0.760**	**0.729**	**0.567**	**0.529**
Δ	+3.70%	+3.10%	+1.60%	+2.50%

Table 5: Comparison of PLATO's performance with baseline methods on Python dataset

Method	intra-project		inter-project	
	EM	weighted-F1	EM	weighted-F1
Typilus	0.516	0.484	0.441	0.412
Transformer	0.463	0.372	0.431	0.425
TypeBert	0.522	0.490	0.435	0.428
PLATO	**0.559**	**0.514**	**0.482**	**0.447**
Δ	+3.70%	+2.40%	+4.10%	+1.90%

Thus the variable is erroneously classified as *Timer*. And by incorporating the VTC-based kernelized attention, model robustly infers the variable as *boolean* based on ground-truth evidence. The visualization shows that VTC-based kernelized attention forces the model to base its inference on relevant, domain-invariant features thus makes it more robust and transferable among language domains.

Impact of Ensemble-based Inference. Rows "-Seuquence" and "-Kernel" show that PLATO substantially outperforms the two sub-models. Note that the results of PLATO w/o VTC and PLATO-Sequence are the same because the sequence model is the version of PLATO without the VTC-based kernelized attention. The κ- bagging ensemble strategy can make the best of the kernelized model and compensate its weakness when dealing with language-specific corner cases.

Answer to RQ2: Each component in PLATO is useful for the cross-lingual transfer learning of statistical type inference task. In conclusion, syntax enhancement improves the performance significantly by introducing feature overlap among language domains. The VTC-based kernelized attention module improves performance consistently. κ-bagging ensemble strategy manages to make the best of it by compensating for the language-specific corner cases.

3.4 RQ3: Using Partial Labeled Target Language Data

Setting. In the real-world settings, during the early stage of an optionally-typed programming language, the type hint annotations of the language provided by developers are scarce, especially for primitive types. Thus, it would be extremely valuable if we were able to quickly build a functional type inference tool by leveraging existing cross-lingual labeled dataset to augment the training data of the model. We select 10%, 20%, …, 100% chunks of samples from the target dataset together with 10% of the known source language dataset to train the model. Note that we only select a few source language data (*i.e.*, 10%) because we try to reduce the effect of the size of source data on the final results. The following baselines are selected to demonstrate the usefulness of PLATO:

- *Bert with Supervised Learning*. Since our method is based on Bert, we fine-tune the pretained XPLM model on the multi-lingual partial labeled data (*i.e.*, the partial labeled target language data and the source data) with fully-supervised learning.
- *PLATO without Kernel*. To show the effect of the kernelized attention on PTL, we evaluate the unkernelized sub-model of PLATO, *i.e.*, removing the kernelized model from PLATO.

We follow the settings in previous works [5, 23] and evaluate the results under two settings: (1) *intra project*: the training and test dataset come from same project sets; (2) *inter project*: the training and test dataset come from different project sets.

Results. Figure 7 shows the results. First, we can see that PLATO w/o kernel steadily outperforms the baseline cross-lingual Bert model under both settings. Particularly, the improvement is more significant when the size of labeled data is small (ratio < 0.5). For example, under the intra-project setting for Python → TypeScript, when ratio = 0.1, it improves the baseline model by +10.0%@EM and +15.8%@weighted-F1. Besides, even when using full target training set (ratio = 1.0), it improves the baseline Bert by +1.3%@EM and +1.2%@weighted-F1. The results demonstrate that using out-domain cross-lingual data with syntax enhancement can already obtain a considerable boost in performance. Then, we consider the results of PLATO, for Python → TypeScript, as shown in the first row of Fig. 7, PLATO significantly improves over Bert and PLATO w/o kernel under all ratios of labeled target data for both the intra- and inter-project settings. For example, under the intra-project setting, when ratio = 0.1, PLATO increases the Bert baseline by +12.7%@EM and +18.2%@weighted-F1; when ratio = 1.0, PLATO manages to increase it by +3.7%@EM and +3.1%@weighted-F1. The result is consistent for TypeScript → Python. The results indicate that using our kernerlized attention can further boost the performance.

Answer to RQ3: As more labeled target language data is available, the performance of PLATO is steadily increased and it outperforms the baseline models consistently under all ratios of target language data.

3.5 RQ4: Evaluation on Supervised Learning

Setting. In this evaluation, we applied PLATO in the supervised based scenario to evaluate whether cross-domain information could be used to improve the learning. We used the same dataset as in RQ3, where all labels of the source language dataset S are available. The conventional supervised based methods directly train and evaluate the model on the dataset with the same language (*i.e.*, S). Differently, PLATO introduces a small part of labeled dataset using other languages (*i.e.*, 10% of the target language data) and performs the transfer learning. For the supervised baselines, we select the following state-of-the-art baselines: for TypeScript, we compare PLATO with TypeBert[26], LambdaNet[43] and the vanilla Transformer model. For Python, we compare with TypeBert, Typilus and Transformer.

Table 4 and Table 5 show the results on TypeScript and Python, respectively. It is obvious that for both TypeScript and Python, PLATO significantly improves all baselines. For example, for intra-project TypeScript inference, PLATO improves over the best TypeBert baseline model by +3.70%@EM and +3.10%@weighted-F1; for inter-project Python inference, it improves the baseline by +4.10%@EM and +1.90%@weighted-F1, indicating that the augmentation from cross-domain language could improve the performance. Our in-depth analysis reveals that the supervised learning on one language dataset tends to be prone to overfitting while the cross-domain augmentation can mitigate this issue. Figure 8 shows a concrete case,

TypeBert pred: *error*

```
handleServerError({isFatal: false, message:
error.message, stackTrace: error.stackTrace,},
error.method,);
```

PLATO sequence pred: *boolean*

```
handleServerError({isFatal: false, message:
error.message, stackTrace: error.stackTrace,},
error.method,);
```

PLATO kernelized pred: *boolean*

```
handleServerError({isFatal: false, message:
error.message, stackTrace: error.stackTrace,},
error.method,);
```

Figure 8: Visualization of models' attention vector for the variable *isFatal* under fully supervised setting.

where we aim to infer the type for the boolean variable `isFatal`. We visualize the attention of the TypeBert model and ours. TypeBert model trained on only TypeScript language heavily focuses on the variable name (*e.g.*, *isFatal*) while completely ignores the ground truth evidence `false`, thus erroneously predicts the variable type. Using the cross-domain information (see PLATO sequence and PLATO kernelized), the overfitting problem can be mitigated and PLATO identifies the more important token (*i.e.*, `false`) for inferring the result *boolean*. Compared with PLATO sequence, PLATO kernelized incorporates the kernelized attention that ignores more irrelevant tokens (*e.g.*, *handleServerError*). Note that, we also considered the impact of the training data size, *i.e.*, adding cross-domain data may increase the size of the training data. Therefore, we control the data size by removing the same amount of data from the training data as the cross-domain data we introduced. The results still show that PLATO can outperform the baselines by using cross-domain data. Due to the space limit, more results can be found on our website.

Answer to RQ4: The supervised approaches on one language data could overfit to some irrelevant tokens. By introducing the small amount of cross-domain data, PLATO can significantly outperforms the baselines.

3.6 Threats to Validity

The implementation of the baselines is a threat to the validity of the results. Since these techniques were not originally built for program analysis tasks, we gave our best efforts in adapting them for our tasks, and fixed all bugs we could identify. The selection of the datasets may not be representative and our results may not generalize. To mitigate this, we selected the two well-known benchmarks which were previously used in type inference tasks. Finally, the label calibration (see Section 3.1) could be another threat to the accuracy of the type prediction. This can be mitigated by outputting specific type names within a meta-type in a ranked list to developers. The data used in the training process could be a threat, we randomly selected the data for all methods in RQ3 and RQ4.

4 RELATED WORK

4.1 Unsupervised Domain Adaptation

As an important case of transfer learning, unsupervised domain adaptation (UDA) has drawn significant attention from the deep

learning communities. The UDA research can mainly be categorized into two streams [37], namely model-centric and data-centric. The goal of model-centric methods are to minimize the distance among domains via feature alignment. Tzeng et al. [40] first proposed using the maximum mean discrepancy (MMD) to minimize the distance between images from two distributions on image classification tasks. Recently, NLP community also started to investigate the possibility of applying the above techniques to language tasks, *e.g.*, sentiment classification [31, 39], POS tagging [47], etc. Pan et al. [35] proposed spectral feature alignment for sentiment classification; the syntax enhancement approach we used in this work lies in this category. The goal of data-centric methods are to bridge the domain gap by manipulating data from source and target domains. Han et al. [21] proposed the *adaptive pre-training*, which adapts contextualized word embeddings from target domain by masked language modeling. Gururangan et al. [20] further introduced *task-specific pre-training* (TAPT) that studies the effect of second-stage pre-training on the transferability across domains. In the software engineering community, transfer learning techniques started to gain attention recently. Nam et al. [34] proposed using transfer learning to improve the performance for cross-project defect prediction. SAR [10] leverages generative adversarial network for API mappings. Although UDA has been broadly explored in the CV and NLP fields, it has not been paid enough attention in the programming language and software engineering community. Yet, considering the fact that we have abundant labeled dataset for high-resource programming languages, there is great potential for knowledge transfer to the relatively low-resource programming languages via UDA.

4.2 Statistical Type Inference

Type inference for optionally-typed language is widely studied in light of the widespread usage of languages such as Python and JavaScript. The ability to infer types automatically makes programming tasks easier, leaving the programmer free to omit annotations while still permitting type checking. Statistical type inference is gaining attention due to its superior performance over traditional rule-based methods. JSNice [38] proposed the first probabilistic type inference system based on conditional random fields (CRFs). DeepTyper [23] introduced the first deep learning based JavaScript type inference model based on recurrent neural networks. And TypeBert [26] achieves the state-of-the-art performance thanks to unsupervised pre-training. Following this line, several deep learning based type inference tools for Python are proposed [5, 36]. Plato advances over these works by allowing the deep learning models to still work even without adequate labeled data.

4.3 Program Representation Learning

Leveraging deep learning models for solving software engineering problems is increasingly gaining popularity. Most of these works focus on monolingual tasks. Zhang et al. [49] used a recurrent neural network for code summarization in Python; code2vec [7] used an attention model for method name prediction in Java; Graph neural networks [50, 51] have been used for the vulnerability detection and security patches tasks of C. Recently, researches started to investigate the power of multi-lingual language models for program analysis tasks. Transcoder [29] introduced a neural transcompiler that is able to translate functions between C++, Java, and Python using unsupervised machine translation.

5 CONCLUSION

In this work, we set out to conduct the first trial of cross-lingual transfer learning of statistical type inference. Our experimental results are positive: by incorporating graph kernel-based kernelized attention, incorporating syntax enhancement using meta-grammar. Our framework not only improves previous domain adaptation baselines significantly when no labeled target language data is available, but also manages to consistently improve the supervised baseline when labeled target language data is available. Our findings indicate great potential of leveraging data across different programming languages for other neural model architectures and other different deep learning-based software engineering tasks. In the future, we plan to extend our method to more code learning based applications such as code search [19] and code summarization [32], and improve the quality of the trained models with existing techniques [14, 45, 46].

ACKNOWLEDGMENTS

This research is partially supported by the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry of Education, Singapore under its Academic Research Fund Tier 1 (21-SIS-SMU-033), Tier 2 (MOE2019-T2-1-040) and Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] 2021. Esprima. https://esprima.org.
[2] 2022. cltl4sti. https://sites.google.com/view/cltl4sti/home.
[3] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. 1976. On finding lowest common ancestors in trees. *SIAM Journal on computing* 5, 1 (1976), 115–132.
[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. 2020. *Compilers: principles, techniques and tools.*
[5] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation.* 91–105.
[6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning.* PMLR, 2091–2100.
[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
[8] Karsten M Borgwardt and Hans-Peter Kriegel. 2005. Shortest-path kernels on graphs. In *Fifth IEEE international conference on data mining (ICDM'05).* IEEE, 8–pp.
[9] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24, 2 (1996), 123–140.
[10] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. SAR: learning cross-language API mappings with little knowledge. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 796–806.
[11] Xilun Chen, Yu Sun, Ben Athiwaratkun, Claire Cardie, and Kilian Weinberger. 2018. Adversarial deep averaging networks for cross-lingual sentiment classification. *Transactions of the Association for Computational Linguistics* 6 (2018),

557–570.
[12] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 516–519.
[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[14] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. Deepstellar: Model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 477–487.
[15] Manaal Faruqui and Chris Dyer. 2014. Improving vector space word representations using multilingual correlation. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*. 462–471.
[16] Yaroslav Ganin and Victor Lempitsky. 2015. Unsupervised domain adaptation by backpropagation. In *International conference on machine learning*. PMLR, 1180–1189.
[17] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769.
[18] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. A kernel two-sample test. *The Journal of Machine Learning Research* 13, 1 (2012), 723–773.
[19] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.
[20] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't Stop Pretraining: Adapt Language Models to Domains and Tasks. *arXiv preprint arXiv:2004.10964* (2020).
[21] Xiaochuang Han and Jacob Eisenstein. 2019. Unsupervised domain adaptation of contextualized embeddings for sequence labeling. *arXiv preprint arXiv:1904.02817* (2019).
[22] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering* 19, 5 (2014), 1335–1382.
[23] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 152–162.
[24] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).
[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
[26] Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1483–1486.
[27] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.
[29] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511* (2020).
[30] Guillaume Lample and Alexis Conneau. 2019. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291* (2019).
[31] Yitong Li, Timothy Baldwin, and Trevor Cohn. 2018. What's in a domain? learning domain-robust text representations using adversarial training. *arXiv preprint arXiv:1805.06088* (2018).
[32] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2021. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *International Conference on Learning Representations*. https://openreview.net/forum?id=zv-typ1gPxA
[33] Tomas Mikolov, Quoc V Le, and Ilya Sutskever. 2013. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168* (2013).
[34] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 382–391.
[35] Sinno Jialin Pan, Xiaochuan Ni, Jian-Tao Sun, Qiang Yang, and Zheng Chen. 2010. Cross-domain sentiment classification via spectral feature alignment. In *Proceedings of the 19th international conference on World wide web*. 751–760.
[36] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.
[37] Alan Ramponi and Barbara Plank. 2020. Neural Unsupervised Domain Adaptation in NLP—A Survey. *arXiv preprint arXiv:2006.00632* (2020).
[38] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.
[39] Jian Shen, Yanru Qu, Weinan Zhang, and Yong Yu. 2018. Wasserstein distance guided representation learning for domain adaptation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
[40] Eric Tzeng, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell. 2014. Deep domain confusion: Maximizing for domain invariance. *arXiv preprint arXiv:1412.3474* (2014).
[41] Jean-Philippe Vert, Koji Tsuda, and Bernhard Schölkopf. 2004. A primer on kernel methods. *Kernel methods in computational biology* 47 (2004), 35–70.
[42] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.
[43] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).
[44] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.
[45] Xiaofei Xie, Wenbo Guo, Lei Ma, Wei Le, Jian Wang, Lingjun Zhou, Yang Liu, and Xinyu Xing. 2021. RNNrepair: Automatic RNN repair via model-based analysis. In *International Conference on Machine Learning*. PMLR, 11383–11392.
[46] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 146–157.
[47] Michihiro Yasunaga, Jungo Kasai, and Dragomir Radev. 2017. Robust multilingual part-of-speech tagging via adversarial training. *arXiv preprint arXiv:1711.04903* (2017).
[48] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.
[49] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.
[50] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *arXiv preprint arXiv:1909.03496* (2019).
[51] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, Shangqing Liu, and Yang Liu. 2021. SPI: Automated Identification of Security Patches via Commits. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–27.