Exploring Type Inference Techniques of Dynamically Typed Languages

C. M. Khaled Saifullah Muhammad Asaduzzaman† Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
†School of Computing, Queen's University, Canada
{khaled.saifullah, chanchal.roy}@usask.ca †muhammad.asaduzzaman@cs.queensu.ca

***Abstract*—Developers often prefer dynamically typed programming languages, such as JavaScript, because such languages do not require explicit type declarations. However, such a feature hinders software engineering tasks, such as code completion, type related bug fixes and so on. Deep learning-based techniques are proposed in the literature to infer the types of code elements in JavaScript snippets. These techniques are computationally expensive. While several type inference techniques have been developed to detect types in code snippets written in statically typed languages, it is not clear how effective those techniques are for inferring types in dynamically typed languages, such as JavaScript. In this paper, we investigate the type inference techniques of JavaScript to understand the above two issues further. While doing that we propose a new technique that considers the locally specific code tokens as the context to infer the types of code elements. The evaluation result shows that the proposed technique is 20-47% more accurate than the statically typed language-based techniques and 5-14 times faster than the deep learning techniques without sacrificing accuracy. Our analysis of sensitivity, overlapping of predicted types and the number of training examples justify the importance of our technique.**

***Index Terms*—type inference, word embedding, localness, dynamically typed language**

I. Introduction

Dynamically typed programming languages enable developers to write less verbose code by removing the burden of specifying types in code, thus support quick prototyping. Such dynamic type systems allow language designers to avoid spending considerable time developing a type system to ensure the completeness of the program at compile time. However, the development and usages of TypeScript[1], Flow[2] and Closure[3] indicate that leading software companies are now considering static typing as an important part of developing code. Recent research results also show the benefits of static typing. For example, Gao et al. [1] find that adding type annotations in JavaScript can help to avoid 15% of the reported bugs. Prior studies [2], [3] also show that static type systems help in understanding undocumented code, fixing type issues and solving semantic errors, thus have a positive impact on the maintenance of software. Finally, building code completion tools also requires type information. For example, method completion tools remove irrelevant method names based on the type of receiver variable [4]. The lack of type information thus makes the dynamically typed languages difficult to provide precise completion proposals. Therefore, it is important to add type information to dynamically typed languages.

Existing type inference techniques that learn from code examples can be divided into two broad categories. First, there are techniques [5]–[9] that are designed to resolve the types of online code snippets. These techniques are developed and tested for the code snippets written in statically typed languages (such as Java). While the language is statically typed, those online code snippets often do not contain type declarations [7]. This makes it difficult to determine which libraries need to be imported to compile the code correctly. Techniques such as StatType [8] and COSTER [9] fall under this category. For example, StatType [8] uses statistical machine translation whereas COSTER [9] leverages a combination of three similarity measures for inferring types. While both techniques achieve high precision and recall, they are only tested for code snippets written in Java. However, it is not clear whether such techniques can provide similar performance for dynamically typed languages, such as JavaScript.

The other group of techniques are specifically designed and tested for dynamically typed programming languages (i.e., JavaScript). For example, Raychev et al. [10] developed the technique JSNice that predicts the type of a code element based on the types of the surrounding code elements that are connected with the target code element through the dependency graph. Malik et al. [11] developed a technique, called NL2Type, that leverages JSDoc, comments, the formal signatures of the functions, and a recurrent neural network model to infer the types of the functions and its parameters. DeepTyper [12] uses a neural machine translation-based approach to infer the types of JavaScript code elements. Type inference techniques for JavaScript receive significant performance gain from using a deep learning technique. However, a problem with deep learning techniques is that they require considerable training time. While training often considers a one-time operation, supporting a new library or adding more training examples require retraining the model. This motivates us to investigate the type inference of JavaScript further.

In this paper, we first conduct an empirical study to understand how the type inference techniques developed for the statically typed language (i.e, Java) perform for the dynamically typed language (i.e, JavaScript). To answer the question,

[1]https://www.typescriptlang.org/
[2]https://flow.org/
[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures

we apply two state-of-the-art techniques developed for Java to predict the types of JavaScript code elements. While doing the empirical study, we find code elements specific to a type usage are closely located to it, also known as locally specific. Thus, we investigate how to capture such localness property more accurately. We use a combination of word embedding, context similarity, and local model to infer the types of code elements. Finally, we compare the proposed technique with not only the techniques developed for Java but also the deep learning based type inference techniques developed for JavaScript. Thus, our study is based on the following research questions.

RQ1: How do the techniques developed for Java perform for type inference of code written in JavaScript?

The results of the empirical study show that the performance of both competing techniques drops significantly when applied to source code written in JavaScript.

RQ2: Can we develop a type inference technique that can address the limitations of techniques discussed in RQ1?

We observe 20-47% performance gain for our proposed technique than the techniques developed for inferring types in Java code snippets.

RQ3: How do the deep learning techniques developed for JavaScript perform in comparison with the proposed technique?

Findings from the study are aligned with prior studies [13]–[16], indicating that our proposed technique can significantly reduce the training time with comparable performance.

Thus, we make the following three contributions.

1) Conduct an empirical study to evaluate the performance of type inference techniques that are developed for Java code snippets for JavaScript code snippets.
2) Propose a technique that uses local specific code tokens as context, Word2Vec, context similarity, and a local model to infer the types of the code elements of JavaScript.
3) Conduct a comparative study of the proposed technique with the deep learning-based state-of-the-art techniques along with the extensive analysis of the result.

The rest of the paper is organized as follows. Section II presents a motivating example for the study. Section III presents prior studies related to our work. Section IV introduces the dataset and explains the evaluation procedure followed by the answers of three research questions in Section V, VI and VII. Section VIII provides further insights about our proposed technique and Section IX summarizes the key findings of the study. We discuss threats to the validity of the study in Section X. Finally, Section XI concludes the paper.

II. Motivating Example

Let us consider a JavaScript function as shown in Fig. 1 that takes any physical body *imposter*, three dimensional *force* and *point* as parameters, and applies impulse on the *imposter* by the *force* at that *point*.

TypeScript enables optional types to be added to the JavaScript code. While developers may benefit from such type

(a) JavaScript Code

```
1  function applyImpulse(impostor, force, point){
2    var worldPoint = new this.BJSCANNON.Vec3(point.x, point.y, point.z);
3    var impulse = new this.BJSCANNON.Vec3(force.x, force.y, force.z);
4    impostor.physicsBody.applyImpulse(impulse, worldPoint);
5  }
```

(b) TypeScript Code

```
1  function applyImpulse(impostor: any, force: Vector3, point: Vector3) : any {
2    var worldPoint : Vector3 = new this.BJSCANNON.Vec3(point.x, point.y, point.z);
3    var impulse : Vector3 = new this.BJSCANNON.Vec3(force.x, force.y, force.z);
4    impostor.physicsBody.applyImpulse(impulse, worldPoint);
5  }
```

Fig. 1. An example of a JavaScript code and the corresponding code in TypeScript.

annotations of code elements (see Fig. 1 (b)), annotating an existing codebase is a time consuming operation, requires expertise and often can be erroneous [12]. An automated technique that can learn from existing type annotations of codebases and can recommend types of JavaScript code elements as a developer types code can be useful in this case.

Furthermore, software engineering tasks such as code completion can be difficult. For example, the parameter *imposter* is annotated as *any*. Therefore, if a user requests for code completion at Line 4 by typing a dot (.) after *imposter*, the completion system will fail to recommend anything as it does not have any type information.

Finally, the function does not have any JSDocs or line comments. Thus, techniques that depend on JSDoc and line comments, such as NL2Type [11], will not not work in this case. On the other hand, since the function is very small in size, COSTER [9] will not be able to collect any code tokens outside the top and bottom four lines. This can impact the performance of the technique.

III. Related Study

A. Empirical Studies on Type Inference

A number of studies explore the usefulness of type inference in dynamically typed languages. Hackett and Guo [17] analyze JavaScript snippets and show that a type inference engine can increase the performance of different functionalities of a website by 50%. Pradel et al. [18] analyze scripts in the runtime, find inconsistencies of types, report them as bugs. Gao et al. [1] investigate the TypeScript[1] and Flow[2] for detecting buggy code and find that around 15% of bugs can be detectable by both engines. Ray et al. [19] conduct a large scale empirical study on GitHub projects and find that statically typed languages are less defect prone than dynamically typed languages. The above works either examine buggy type annotations or the importance of a type inference engine whereas we focus on investigating type inference techniques of a dynamically typed language (i.e., JavaScript).

B. Type Inference in Statically Typed Languages

Techniques developed for statically typed language, such as Java, can be divided into two groups: linking code from text and linking code from code.

Linking code from text based techniques use documentation [5], [20], [21], bug reports [22], [23], emails [24] and

posts from online Q&A sites [6] to find appropriate types in Java code snippets. However, these approaches suffer from accuracy due to the lack of documentation [25] and informal nature of bug reports and posts [9].

Baker [7] is the first to link code from the other code tokens situated within the same scope and uses an iterative deducing technique to infer types of code elements. However, the technique fails to infer types of a number of code elements due to strict scope rules [8], [9]. StatType [8] uses the original code fragment as the source language and type resolved code fragment as the target language. It leverages a statistical machine translation technique to find mapping between them. The technique performs poorly for the types having a lesser number of examples [9]. COSTER [9] is another technique that can infer types of code elements in Java code snippets based on the type usage contexts and three different similarity measures (i.e, occurrence likelihood, context similarity, and name similarity scores). In our RQ1, we find that COSTER is not applicable for dynamically typed languages (such as JavaScript) since it cannot capture the differences in the structure of languages.

C. Type Inference in Dynamically Typed Languages

Inferring types in case of dynamically typed languages such as JavaScript, Python, Ruby and so on can be categorized into three groups: type annotation, program analysis-based and probabilistic type inference.

TypeScript[1] developed by Microsoft and Flow[2] developed by Facebook are the type annotation based solutions. However, developers need to manually annotate the type information, which requires considerable time and effort [12].

Program analysis-based type inference techniques are largely formal and static rule based [26]–[34]. Such approaches are unable to perform well for statistically uncomputable functions such as *eval* [11], [17].

JSNice [10] is the first work on probabilistic type inference that constructs a dependency network between the code elements of known properties and unknown properties. The unknown node of the dependency network is predicted using the conditional random field. DeepTyper [12] is a neural machine translation-based technique that considers the stream of code tokens as the source language and the stream of types as the target language. Based on the bidirectional Recurrent Neural Network (RNN), the technique learns the mapping between the source and target languages. The technique then infers types based on the mapping. NL2Type [11] is another deep learning-based type inference technique for JavaScript that focuses on the parameters and return types of functions. The technique creates contexts based on the JSDoc, comments and the formal signatures of methods. Those natural texts are preprocessed and passed through a Word2Vec and a bidirectional Long Short Term Memory based neural network to learn the nonlinear relations. Both NL2Type and DeepTyper outperform JSNice whereas we develop a technique that significantly reduce the training time without sacrificing the accuracy (see Section VII).

IV. Experimental Design

This section describes the dataset and the experimental settings of our study.

TABLE I
Dataset Overview

	Total	**Training**	**Testing**
No. of Projects	774	697	77
No. of Files	100,805	90,724	10,081
No. of Tokens	25,997,343	23,455,632	2,541,711

A. Dataset Description

For evaluating the performance of type inference techniques, we use the dataset developed by Hellendoorn et al. [12]. The dataset consists of the top 1000 open source JavaScript projects selected based on the star count whose code elements are annotated by developers using TypeScript (ts). We successfully retrieve 774 projects in September 2019 out of those 1000 projects. The rest of the projects are either deleted or made private. Thus, we cannot include them in our study. Table I shows an overview of the dataset we used for our experiments. The dataset contains more than 100K files. All projects are parsed using the TypeScript compiler[1]. The compiler returns a type for each variable, class object, literal, function's return type, and parameter. The type of a code token represents an instance in our dataset.

We use the ten-fold cross-validation technique where the collected projects are divided into ten different folds. Nine out of those ten folds are used to train, and the remaining fold is used for testing. We repeat the process ten times by changing the test fold and record the performance of each competing technique. The final result is calculated by taking the average performance of all ten folds.

B. Evaluation Procedure

We use the precision, recall, and F_1 score to measure the performance of compared techniques. We present the code example for each instance in the test dataset to a technique for inferring the type of that code element. We consider the recommendation is *relevant* if the actual type is present in the top-k recommendations. The precision, recall, and F_1 score are defined as follows.

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \tag{1}$$

$$Recall = \frac{recommendations\ made \cap relevant}{recommendations\ requested} \tag{2}$$

$$F_1 Score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{3}$$

Here, *recommendations requested* refers to the total number of instances in the test set. *Recommendations made* is the number of instances for which a technique infers types. *Recommendations requested* is the number of instances required inference. We use a two-tailed Wilcoxon signed-rank test [35] to determine whether the difference between the performance

of two compared techniques are statistically significant or not. For each evaluation metric (i.e., precision, recall and F_1 score), we collect the result of each competing technique for each fold as one data point and compare ten data points obtained from ten different folds with that of the compared technique.

V. **RQ1**: How Do the Techniques Developed to Infer Types in Java Code Snippets Perform For JavaScript Code Snippets?

A. Motivation

Techniques that infer types in online Java code snippets showed great performance [8], [9]. Prior studies of those techniques also argued that such techniques can easily be adapted to dynamically typed programming languages (such as JavaScript). We are interested in learning the effectiveness of those techniques to detect types in JavaScript code snippets. Results from the study can help us to decide whether we can reuse those techniques for JavaScript code snippets or further modification is required.

B. Approach

We choose two state-of-the-art techniques, StatType [8] and COSTER [9], that are developed to detect types in code snippets written in Java language. We made necessary changes to adapt those techniques for JavaScript language. COSTER collects both local and global contexts to capture the type usage context of code elements. The technique collects any types, keywords, function calls and operators that appear within the top and bottom four lines as the local context. The global context consists of methods outside of the local contexts that are invoked on the receiver variable and that use the code element or the receiver variable as the parameter. To adapt the technique for JavaScript, we leverage the TypeScript compiler to collect both contexts. We then leverage a combination of three different similarity measures to predict types of code elements. In case of StatType, we collect the stream of the resolved code elements as the source language and the stream of types of the code elements as the target language. For the target language, similar to COSTER, we collect the type of code elements such as variables, class objects, literals, function's return type and parameter. Next, we use the same Phrasal [36] tool used by the authors to train and test the statistical machine translation model. We then collect the precision, recall and F_1 score (F_1) for top-1, top-3 and top-5 recommendations. We summarize the results in Table II.

TABLE II
Performance comparison of statically typed language based techniques StatType and COSTER for JavaScript snippet

Technique	Recc.	Prec.	Rec.	F_1
StatType	Top-1	28.69	25.73	27.13
	Top-3	37.29	35.25	36.24
	Top-5	49.36	47.28	48.30
COSTER	Top-1	17.34	12.84	14.75
	Top-3	27.39	24.18	25.69
	Top-5	32.61	28.41	30.37

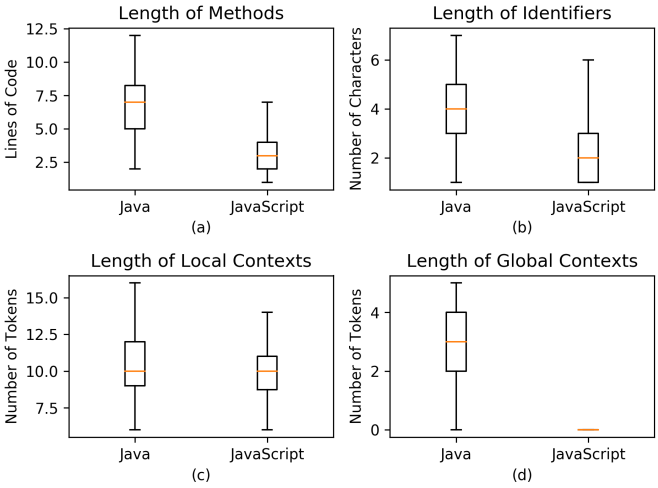


Fig. 2. Comparing between Java and JavaScript datasets in terms of the length of methods, identifiers and two types of contexts.

C. Evaluation

Both StatType and COSTER did not perform well for detecting types in JavaScript files, as shown in Table II. For the top-1 recommendation, the precision and recall of StatType are 28.69% and 25.73%, respectively. Performance improves as we consider more recommendations but can only be considered as mediocre. For example, the precision and recall reach to 49.36% and 47.28% for the top-5 recommendations,respectively. COSTER performs comparatively worse than the StatType. The precision and recall of COSTER are 9.9-16.75% and 11.07-18.87%, respectively, lower than StatType for all recommendations. However, both techniques performed remarkably well when applied for code snippets written in Java [8], [9].

D. Discussion

To understand the reasons that contributed to such poor performance, we determined the differences between Java and JavaScript languages considering the length of methods, identifiers, and the contexts (i.e., local and global context) collected by COSTER. We used the GitHub dataset of COSTER [9] for Java and our dataset for JavaScript. For both datasets, we consider the number of lines as the length of a method, the number of characters as the length of an identifier, and the number of tokens as the length of the local and global contexts. Fig. 2 summarizes the results. The reasons behind COSTER's poor performance can be derived as follows.

First, the global context does not play a significant role in JavaScript. This is because functions in JavaScript are typically small in lengths compared to that of Java. Therefore, globally related tokens are difficult to find in JavaScript. While the median function length of Java is 6 lines, the number drops to 3 for JavaScript, as shown in Figure 2(a). Moreover, the median length of global context in Java is 2.5 times higher than that of JavaScript (see Figure 2(d)). However, the differences drops significantly when we compare the length of local

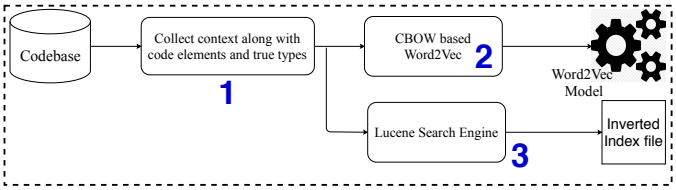


Fig. 3. Overview of the training step of the proposed technique.

contexts between Java and JavaScript datasets, as shown in Figure 2(c)). The median values are 9 and 10 respectively. Thus, COSTER has a hard time collecting globally related tokens for JavaScript comparing to Java and fails to show similar performance.

Second, the name of identifiers in JavaScript is comparatively shorter than that of Java, as shown in Figure 2(b). While the median identifier length in JavaScript is 2 , the value reaches to 4 for Java. Thus the name similarity measure in COSTER do not perform well when tested for JavaScript code snippets.

StatType fails to infer types that are less popular. Our dataset for JavaScript is dominated by *any* type (46.62%). Such dominance in the dataset causes StatType to be biased to the *any* type. Thus, we can conclude that type inference techniques developed for Java code snippets are not equally effective for JavaScript code snippets. All these motivate us to investigate the problem further.

VI. **RQ2**: Can We Develop a Type Inference Technique that Can Address the Limitations of Techniques Discussed in RQ1?

A. Motivation

Previously we observe that source code elements in JavaScript are also locally specific, meaning code elements that are related to a type usage are closely located. COSTER already attempted to capture such localness property of the source code by considering tokens that appear within the top and bottom four lines of a code element whose type needs to be inferred. However, COSTER did not perform well for JavaScript that made us interested in investigating other approaches (such as word embedding) to capture code elements that are related to a type usage context.

B. Technique Description

In this section, we describe our proposed technique that infers the types of JavaScript code elements (i.e., variables, class objects, literals, function's return types and parameters). The steps of the technique are discussed below.

1) Collect Type Usage Context: We parse each JavaScript file using the TypeScript compiler, create the Abstract Syntax Tree (AST) and collect the type usage context for each variable, class object, literal, function's return type and parameter. We refer to them as the code element unless specified otherwise. The type usage context of a code element consists of tokens that include the types of identifiers, keywords, function calls, class objects, and operators within the top and bottom four lines of that code element. Our selection of four lines is based on the fact that we obtain the best result using this setting. For example, the context for class object $impulse$ at line 3 would be $function$, any, $Vector3$, $Vector3$, var, $Vector3$, $=$, new, $BJSCANNON$, $Vec3$, $Vector3.x$, $Vector3.y$, $Vector3.z$, var, $=$, new, $BJSCANNON$, $Vec3$, $Vector3.x$, $Vector3.y$, $Vector3.z$, $applyImpulse$, $Vector3$, $Vector3$. The primary motivation for choosing such a context is two-folded. First, the context contains locally specific code tokens which are inspired by the principle of naturalness [37] and localness [38] properties of the source code. Second, we consider the type information of code elements (i.e., identifiers) rather than their lexical information. Such context showed good performance in prior studies [8], [12].

We use an inverted index structure to organize type usage contexts along with their associated types. Each type usage context appears as a document and tokens of those documents are used to index those sets of documents. Such an index structure allows us to quickly retrieve types whose usage context matches with that of a query context. Instead of implementing the inverted index structure from the scratch, we use the implementation available in the Lucene search engine [39].

2) Train Model: Next, we apply the Continuous Bag of Word (CBOW) architecture of Word2Vec [40] technique on the training dataset (2 in Fig. 3). Word2Vec [40] is a word embedding technique that takes words/tokens from the training contexts as input and creates a d-dimensional continuous vector space. Each word/token is then represented by a vector in such a way that, if plotted in a vector space, semantically similar words/tokens appear close to each other [41]. There are two ways to create such a word vector. The first one is the Continuous Bag of Word (CBOW) architecture where words/tokens are embedded into the vectors based on their context. The other one is the Skim-gram architecture where the context is embedded based on the word/token. We used the former one since it considers the whole context as one observation and predicts the type based on the context during the inference step. For example, if contexts having tokens such as *Identifier* and *ImportKeyword* are found frequently for the type *String*, the Word2Vec model will learn that context with these tokens are very closely related to the type *String*. During inference, if any context with such tokens are found, the model will predict the type *String* with a higher probability value.

3) Infer Types: To infer the type of a code element, we follow the following sequence of steps. First, our proposed technique collects the type usage context of a target code element that can be a variable, a class object, a literal, return type of a function or a parameter. We use the term query element to refer to the target code element, the associated type usage context as the query context and the actual type of the code element as the true type. For our example shown in Fig. 4, code element $resolved$ at line 10 is the query code element, $ResolvedUrl$ is the true type and the code tokens within 6-14 formulate the query context.

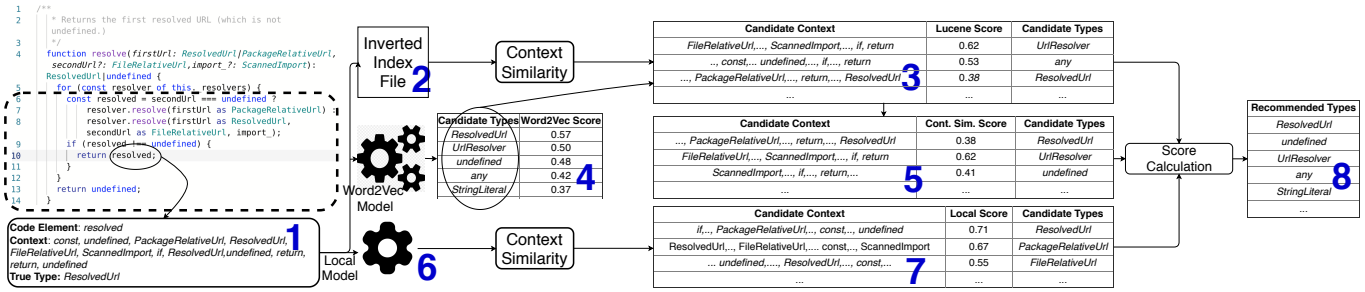


Fig. 4. Overview of the inference steps with an example of the proposed technique.

Second, we pass the query context (C_q) to the Inverted Index File (2 in Fig. 4) and it returns a ranked list of all contexts with associated types that are stored during the training time. We then collect the top 500 usage contexts and their associated types. We use the term candidate context to refer to any element of the list of contexts returned by the search engine. The types associated with those contexts are referred to as the candidate types. Next, we calculate the context similarity score between the query and each candidate context. To do that, we apply the Cosine [42] similarity method that use Eqn. 4 to calculate the context similarity ($Sim_{con}(C_q, C_{ci})$) between query context (C_q) and each candidate context (C_{ci}).

$$Sim_{con}(C_q, C_{ci}) = \frac{N_o}{N_m} \times cosine(C_q, C_{ci}) \tag{4}$$

Here, N_o is the number of tokens of the candidate context that appear in the same order of that of query context, and $N_{matched}$ is the number of tokens that are matched. The equation returns a score between 0 to 1, signifying the similarity between the query and each candidate context. In our example in Fig. 4, the candidate context of type $URLResolver$ has the highest context similarity score of 0.62 followed by the candidate context of type any and $ResolvedUrl$.

Third, we present the query context to the previously trained Word2Vec model. Since the Word2Vec model is learned using the entire training dataset, we refer to this as a global model. The model returns a score for each type that represents how similar the type is given the query context. We pick top-k types based on the Word2Vec score (4 in Fig. 4) and refine the list of candidate contexts based on these types (5 in Fig. 4). In Fig. 4, Word2Vec score of $ResolvedUrl$ becomes the highest followed by $UrlResolver$, $undefined$, and so on.

Fourth, we present the query context to the local model (6 in Fig. 4). By local model we mean the contexts inferred so far within the project associated by their types. To create the local model, we save the context and the top-1 recommendation result after each inference as long as we remain on the same project. If we consider that the given JavaScript snippet in Fig. 4 is the only code in the project, then the local model for our example will consists of the contexts of all code elements before line 10 that are inferred so far with their top-1 types. Contexts of local models are referred to as candidate local contexts and the top-1 types are referred to as candidate local types. The motivation behind considering the local model is two-folded. First, the global model can be biased by the types that have a very large number of examples, such as any. The local model can help to skip such bias by capturing the project specific similarities. Second, the use of a local model along with a global model is found effective in literature [15]. We calculate the context similarity between the query context and each candidate local context using Eqn. 4 (7 in Fig. 4). In our example, candidate local types such as $ResolvedUrl$ are found having the highest local score followed by $PackageRelativeUrl$, $FileRelativeUrl$ and so on.

Finally, we sort the list of candidate types based on three scores: Word2Vec score, context similarity score and local score (8 in Fig. 4). Word2Vec score ($Sim_{word2vec}\ (T_{ci}, C_q)$) dictates how similar the ith candidate type (T_{ci}) is with the query context (C_q). Context similarity score ($Sim_{con}\ (C_q, C_{ci})$) tells how similar the ith candidate context (C_{ci}) is with the query context (C_q) and the local context similarity score ($Sim_{local}\ (C_q, C_{lci})$) captures the localness tendency of the query context (C_q) with respect to the ith local candidate context (C_{lci}). We use the Eqn. 5 to calculate the score of the ith candidate type ($Score(T_{ci})$):

$$\begin{aligned} Score(T_{ci}) = \alpha \times Simword2vec\ (T_{ci}, C_q) \\ + \beta \times Simcon\ (C_q, C_{ci}) \\ + \gamma \times Simlocal\ (C_q, C_{ci}) \end{aligned} \tag{5}$$

Here, α, β and γ are the coefficients of Word2Vec, context similarity, and local context similarity scores, respectively. The coefficients are tuned using Hill Climbing Adaptive Learning algorithm [43]. The sorted list of candidate types is considered as the top-k recommendations of the proposed technique. We calculate the precision(Prec.), recall (Rec.) and F_1 score (F_1) for top-1, top-3 and top-5 recommendation for the compared techniques, as shown in Table III.

C. Evaluation

Our proposed technique outperforms both StatType and COSTER with a big margin, as shown in Table III. The precision of the proposed technique for the top-1 recommendation is 24.45% higher than StatType and 35.77% higher than COSTER. The recall of the proposed technique is 20.48% and

TABLE III
PERFORMANCE COMPARISON OF STATTYPE, COSTER AND THE PROPOSED TECHNIQUE

Algorithm	Recc.	Prec.	Rec.	F_1
StatType	Top-1	28.69	25.73	27.13
	Top-3	37.29	35.25	36.24
	Top-5	49.36	47.28	48.30
COSTER	Top-1	17.34	12.84	14.75
	Top-3	27.39	24.18	25.69
	Top-5	32.61	28.41	30.37
Proposed Technique	Top-1	**53.14**	**46.21**	**49.43**
	Top-3	**65.27**	**60.11**	**62.58**
	Top-5	**79.24**	**73.51**	**76.27**

33.37% higher than that of StatType and COSTER respectively. The differences increase as we increase the number of recommendations. In case of the precision, the difference increases from 24.45 to 29.88% for StatType and 35.8 to 46.63% for COSTER when we increase the number of recommendation from 1 to 5. Similarly recall increases from 20.48 to 26.63% for StatType and 33.37 to 45.1% for COSTER. The differences of performances are also statistically significant.

VII. **RQ3**: HOW DO THE DEEP LEARNING TECHNIQUES DEVELOPED FOR JAVASCRIPT PERFORM IN COMPARISON WITH THE PROPOSED TECHNIQUE?

A. Motivation

Despite having promising results in different problems, prior studies [13]–[16] show that deep learning techniques are not always the best solution for different software engineering problems. Two state-of-the-art deep learning-based type inference techniques are developed for JavaScript programs [11], [12]. This section compares our proposed technique with those two state-of-the-art techniques to understand the importance of using deep learning techniques.

B. Approach

We collected the publicly available code base of DeepTyper[4] and NL2Type[5]. Similar to our approach, DeepTyper collects the code elements and their types using TypeScript compiler. The technique then feeds them into a bidirectional recurrent neural network to map the source language (i.e., code elements) to the target language (i.e., types) using a sequence to sequence learning architecture. On the other hand, NL2Type collects the JSDoc[6], line comments and the formal signatures of the functions. Since the approach has no support for the TypeScript, we implement an interface to collect the dataset for NL2Type using TypeScript compiler.

C. Evaluation

DeepTyper and our proposed technique can detect types of variables, class objects, literals, function's return types and parameters. However, NL2Type focuses on detecting only the function's parameters and return type. Thus, we compare our technique with DeepTyper for all code elements. We calculate the precision (Prec.), recall (Rec.) and F_1 score (F_1). The results of DeepTyper and our proposed technique are shown in Table IV.

TABLE IV
PERFORMANCE COMPARISON OF DEEPTYPER AND THE PROPOSED TECHNIQUE FOR ALL CODE ELEMENTS

Technique	Recc.	Prec.	Rec.	F_1
DeepTyper	Top-1	**54.21**	45.29	49.35
	Top-3	**66.82**	58.67	62.48
	Top-5	**80.19**	68.24	73.73
Proposed Technique	Top-1	53.14	**46.21**	**49.43**
	Top-3	65.27	**60.11**	**62.58**
	Top-5	79.24	**73.51**	**76.27**

As shown in Table IV, DeepTyper has higher precision than our proposed technique. However, the difference is not much significant, ranges between 1-1.5%. However, our technique achieves better recall and F_1 scores than the DeepTyper. The differences become more noticeable as we increase the number of recommendations. For example, our technique achieves 5.27% higher recall values than the DeepTyper for the top-5 recommendations. All differences are statistically significant. We applied Choen's d [44] to measure the effect size. We found that the effect size is negligible (0.1) for precision. Next, we collect the results of the function's return type and their parameters for the proposed technique and DeepTyper. We compare the results with that of NL2Type.

TABLE V
PERFORMANCE COMPARISON OF DEEPTYPER, NL2TYPE AND THE PROPOSED TECHNIQUE FOR FUNCTION'S RETURN TYPE AND THEIR PARAMETERS

Technique	Recc.	Prec.	Rec.	F_1
DeepTyper	Top-1	62.85	50.17	55.80
	Top-3	75.28	61.28	67.56
	Top-5	**86.81**	70.11	77.57
NL2Type	Top-1	**63.57**	52.17	57.31
	Top-3	**77.24**	63.22	69.53
	Top-5	85.22	71.28	77.63
Proposed Technique	Top-1	61.35	**55.72**	**58.40**
	Top-3	74.28	**68.22**	**71.12**
	Top-5	84.83	**75.81**	**80.07**

Similar to the previous experiment, our proposed technique lacks precision by 0.5-3% whereas achieves a better recall of 3-7% comparing with other techniques. Again, we observe that all differences are statistically significant. However, we observe a negligible effect size for precision.

D. Discussion

In our evaluation, we see that the proposed technique is very competitive with the state-of-the-art deep learning-based type inference techniques. However, deep learning techniques require a significant amount of time and memory. Therefore, we are interested to compare the time and memory requirement

[4]https://github.com/DeepTyper/DeepTyper
[5]https://github.com/sola-da/NL2Type
[6]https://devdocs.io/jsdoc/

with DeepTyper and NL2Type. We consider the time required for extracting code tokens, training the technique and inferring types. To compare the memory requirements, we consider two different aspects. First, we calculate the size of models and indexes. Second, we calculate both the random access memory (RAM) and the video random access memory (VRAM) usages. The time required to parse the JavaScript code into the desired dataset is the code extraction time, time to train the neural network or collecting Word2Vec and inverted index file is the training time and the time to infer a code element is the inference time. The size of the neural network or the word2vec model is the model size, the size of the inverted index file is the index size, and the amount of CPU memory consumed while training is the RAM consumption and the amount of GPU memory consumed while training is the VRAM consumption. All the results are shown in Table VI. For fair comparison we used the same server having 12 CPUs of Intel Xeon processor with 2.10 GHz processing speed each, 32 GB of memory and NVIDIA Tesla K20c with 4GB of memory.

TABLE VI
TIME AND MEMORY COMPARISON OF PROPOSED TECHNIQUE(PRO. TECH.), DEEPTYPER AND NL2TYPE

Criteria	Pro. Tech.	DeepTyper	NL2Type
Code Extraction time (Hr)	**9.3**	9.6	**9.3**
Training Time (Hr)	**9.9**	63.7	54.3
Taining Time w/o GPU (Hr)	**9.9**	134	103
Inference Time (ms)	**7.29**	12.73	10.52
Model Size (MB)	**5.8**	71	57.9
Index Size (MB)	42.6	-	-
RAM consumed (MB)	**482**	1492	752
VRAM consumed (MB)	-	761	398

Our proposed technique requires the lowest amount of memory and time, as shown in Table VI. More importantly, the proposed technique does not require any GPU support. The compared techniques are time and memory efficient when GPU is provided. However, the proposed technique is 5-7 times faster when all the techniques are executed with the help of GPU and 10-14 times faster when run without GPU in case of training. The difference in training time can have a good impact, if a user wants to complete training on a cloud server. For example, to complete the training in Amazon EC2 instances[7], the proposed technique will need 20-30 USD whereas DeepTyper needs 100-150 USD based on GPU or CPU instance, and NL2Type needs 80-120 USD. The above results clearly show that the proposed technique is faster than the compared techniques. In case of memory requirements, the proposed technique takes 17-30% less memory to store the model, 1.5 to 3 time less RAM consumption than the compared techniques. Additionally, the proposed technique does not require any VRAM. Thus, the proposed technique outperforms all the compared techniques in terms of memory requirements.

[7]https://aws.amazon.com/emr/pricing/

VIII. DISCUSSION

This section investigates our design decisions and provides further insights about our proposed technique.

A. Sensitivity Analysis

The goal of this analysis is to validate different design decisions that we make to build our proposed technique. Recall that we consider all code elements within the top and the bottom four lines of a code element to collect the type usage context of that element. These include identifiers, keywords, function calls, class objects, and operators. Our selection of four lines is based on the fact that increasing the number of lines beyond this point increases the execution time without increasing the accuracy of the technique. To understand the effect of different contexts, and similarity scores in case of the performance of the proposed technique, we conduct a set of studies. Our initial context, C_0, contains all tokens from the top four lines. We use Lucene search engine to index types based on the associated contexts. To understand the effectiveness of using the search engine and the context C_0, we build a model M_0 with training examples. Given the context of a code element as a query, we leverage Lucene to search for types whose contexts match with the query context and to return a ranked list of types. We build another model M_1 where the search engine uses all tokens from the top and bottom four lines. Next, we include the context similarity score with M_1 to predict the types and we refer to this model as the M_2. We then use CBOW based Word2Vec technique to build another model, M_3, to understand the importance of using the word embedding technique. Finally, we leverage a local model with M_3 to understand the impact of using a local model. We train and test our technique using precision, recall and F_1 scores for all of the above cases.

TABLE VII
SENSITIVITY ANALYSIS

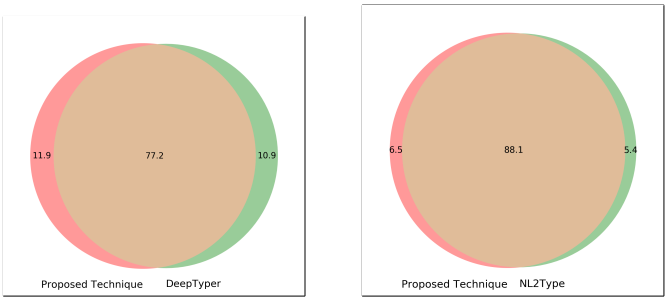
Model	Description	Recc.	Prec.	Rec.	F_1
M_0	C_0+ Lucene	Top-1	21.43	11.86	15.27
		Top-3	31.86	21.73	25.84
		Top-5	42.86	30.73	35.80
M_1	C_1+ Lucene	Top-1	24.38	16.73	19.84
		Top-3	35.72	29.77	32.47
		Top-5	46.25	35.27	40.02
M_2	M_1+ Context Similarity	Top-1	31.83	22.51	26.37
		Top-3	43.62	34.57	38.57
		Top-5	53.72	40.82	46.39
M_3	M_2+ Word2Vec	Top-1	42.18	29.62	34.80
		Top-3	64.28	42.17	50.93
		Top-5	75.17	50.42	60.36
M_4	M_3+ Local model	Top-1	**53.14**	**46.21**	**49.43**
		Top-3	**65.27**	**60.11**	**62.58**
		Top-5	**79.24**	**73.51**	**76.27**

Considering only tokens in the top four lines and using Lucene search engine, we obtain 21.43% of precision and 11.86% of recall for the top-1 position, respectively. Next, we combine Lucene with tokens from the top and bottom four lines. This helps to improve precision and recall by 2.95% and 4.57%, respectively. The inclusion of the context similarity

score improves both precision and recall that reach to 31.83% and 22.51%, respectively. The inclusion of the Word2Vec with the previous model helps to obtain 42.18% precision and 29.62% recall. Finally, we obtain the best result when we consider all sources of information and all similarity measures. The precision and the recall reach to 53.14%, and 46.21% for the top-1 recommendation. We also observe a similar scenario for the top-3 and top-5 recommendations. Results from our study show that the ranked list of types generated by only using the Lucene search engine is not much effective. The reason we use Lucene is to quickly find a list of types whose usage contexts match with the query context. We employ additional sources of information to further refine and rank that list of types.

B. Analysis of Overlapping

We are interested in learning how different type inference techniques complement each other. We consider DeepTyper, NL2Type and our proposed technique for this analysis. For this study, we consider the top-1 recommendation and we refer to a test example as a data point. The overlapping of correctly predicted types between DeepTyper and our proposed technique is shown in Figure 5a. Since NL2Type only focus on predicting the type of function's parameters and the return type, we use Figure 5b to show how many of the correctly predicted types overlap between NL2Type and our proposed technique.



(a) (b)

Fig. 5. Venn diagram showing the overlap of data points correctly predicted by the porposed technique, DeepTyper and NL2Type.

Among all the data points that are correctly predicted by either DeepTyper or our proposed technique, 77.2% of those data points are common between both techniques. 11.9% and 10.9% of those data points are by only our proposed techniques and by DeepTyper, respectively. The percentage of overlapping between our proposed technique and NL2Type is 88.1% considering all the data points that are correctly predicted by any of these two techniques. While 6.5% of those data points are correctly predicted by our proposed technique only, the value drops to 5.4% for NL2Type. The above findings have two important implications. First, our proposed technique can correctly predict types that cannot be detected by the other two techniques, indicating the usefulness of our proposed technique. Second, we see an opportunity to improve the performance of type inference by combining recommendations of DeepTyper or NL2Type with that of our proposed technique in future.

C. Effect of the Number of Training Examples

We are interested in learning how our proposed technique performs for types having a different number of examples in the training dataset. We then compare the result with that of DeepTyper. Two observations can be made from such an analysis. First, such an analysis can help us to understand whether the competing techniques can able to predict rarely seen types. Second, it evaluates the effectiveness of the techniques for frequently occurred types. To do the analysis, we divide the test cases into two groups: (a) cases belong to *any* type and (b) cases belong to the remaining 2,666 types. We separate the test cases of *any* types because it is very dominant (46.62%) in the dataset. Next, we divide the test cases in the other group into five sub-groups based on the usage frequency of those types in the dataset. We refer to the first group as the very unpopular types (VU). The usage frequency of the types in this group is no more than 5% of the total examples. The types whose usage frequency ranges between 6-25% fall under the unpopular types (UP). The usage frequency of the next two groups ranges between 26-50% and 51-75%. They are referred to as the popular (P) and very popular types (VP), respectively. The last group is called the extremely popular types (EP) whose usage frequency ranges between 76-100%. We report the performance of our proposed technique across these five groups of types and we compare the result with that of DeepTyper. We discard NL2Type from this analysis because the technique cannot infer all types that are detected by DeepTyper or by our proposed technique. Thus, no fair comparison can be made possible.

TABLE VIII
EFFECT OF NUMBER OF TRAINING EXAMPLES

Type (% of data)	DeepTyper			Proposed Technqiue		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
any (46.62%)	**63.24**	51.27	56.63	62.75	**58.29**	**60.44**
VU (1-5)%	18.43	9.43	12.48	**24.83**	**18.62**	**21.28**
UP (6-25)%	22.19	11.49	15.14	**29.17**	**22.43**	**25.36**
P (26-50)%	26.14	14.23	18.43	**32.18**	**25.73**	**28.60**
VP (51-75)%	**44.76**	30.75	36.46	43.35	**36.82**	**39.82**
EP (76-100)%	**69.95**	56.18	62.31	67.42	**61.82**	**64.50**

The recall of our proposed technique is higher than the DeepTyper for all different groups of types by 5-12%, as shown in Table VIII. Our proposed technique has lower precision in the case of *any*, very popular and extremely popular types. However, the differences are very small, ranges between 0.5-2.5%. On the contrary, our proposed technique has 6.40%, 6.98%, and 6.04% higher precision than DeepTyper for the very unpopular, popular and less popular types, respectively.

D. Limitations

This section discusses the limitations of our proposed technique.

First, the TypeScript compiler returns *any* type if it cannot bind the type properly. We observe in some cases our technique disagrees with the compiler by returning types other than the *any*. However the type returned from our technique in some of these cases found to be more appropriate. For example, the TypeScript compiler returns *any* as the type of the code token *isValidPrivKey* (see line 2) as shown in Fig. 6 whereas our technique finds it as a function that returns a *boolean* value.

```
1  Promise<KeystoreFile | null> {
2    if (!isValidPrivKey(privateKey)) {
3      throw new Error('Invalid private key');
4    }
```

Fig. 6. An example where TypeScript compiler extract wrong type.

Second, 2.4% of functions in our dataset contain only one line of code. Our technique fails to detect types of code elements in those cases because those code elements have very little or no prior context to collect. For example, the proposed technique fails to infer the types of *arr* and *i* as shown in Fig. 7. One possible solution is to consider the documentation in those cases to address such issues. However, such cases are not very common.

```
1  export const filter = (i, arr => {
2    return -1 !== indexOf(arr, i)? true: false;};
3  }
```

Fig. 7. An example where the proposed technique failed to infer types.

IX. Key Findings

In our study, we find the following key findings that might motive future research on type inference.

(a) Type inference techniques may not generalize across different programming languages: In our RQ1, we found that the type inference techniques developed for Java performed poorly for the JavaScript language. This mostly contributed by the differences in programming language structures. JavaScript methods tend to be small in size and developers typically use short names for identifiers compared to that of Java. Furthermore, the *any* type is more popular is JavaScript than any other types. Such an imbalance in type usages makes it difficult to infer types in JavaScript. Thus, future research should consider evaluating type inference techniques across different programming languages to achieve generalizability.

(b) Understanding the differences between programming languages needs a priority: While analyzing the result in RQ1, we found several differences between Java and JavaScript languages that affect the performance of type inference techniques. Thus, it is important to understand the differences between programming languages so that tool developers can make informed decisions. While COSTER was unable to find a global context in JavaScript code snippets due to the short method length, that was not the case for the local context. Such an understanding helped us to decide what needs to be changed to address the limitations of COSTER. Thus, future research should focus more on understanding how programming languages are different from each other.

(c) Applications of deep learning techniques need to be carefully justified: Prior studies [13]–[16] show that deep learning techniques may not be the best choice for software engineering problems. Our study also support their findings. The deep neural network can find a nonlinear relation between the data. However, such a relationship may not always exist in the data. Furthermore, deep learning techniques are computationally more expensive than traditional machine learning techniques. It is thus important to apply alternative techniques first to justify the need for deep learning techniques. Future research should explain the need for deep learning techniques for the problem first.

X. Threats to Validity

This section discusses threats to the validity of this study.

First, the dataset we used in this study is created by collecting JavaScript projects from GitHub. One can argue that our findings may not generalize to a different dataset. However, we would like to point to the fact that we considered a large number of projects in our study. All these projects are active in the development and have a long development history. Thus, our results should largely carry forward.

Second, we choose the Word2Vec algorithm to determine embedding of code tokens and the cosine similarity as the string similarity measure. Other word embedding techniques or string similarity measures can give different results. However, we obtained the best results using the Word2Vec algorithm and the cosine similarity measure.

Third, we re-implemented StatType and COSTER to work with JavaScript code snippets. While we cannot guarantee that our implementation do not contain any errors, we took great care to avoid such errors.

XI. Conclusion

This paper explores different aspects of type inference tasks for the dynamically typed programming languages, such as JavaScript. We evaluate two state-of-the-art type inference techniques developed for the statically typed programming language (i.e., Java) to understand the effectiveness of those techniques to detect types in JavaScript code. Results from our analysis show that they could not infer more than 50% code elements accurately for top-5 recommendations. Next, we try to capture the localness property of JavaScript code and propose a technique based on the same principle. The results of the proposed technique are found 20-47% more accurate than the statically typed language based type inference techniques. Finally, we compare the proposed technique with state-of-the-art deep learning techniques developed for inferring types in JavaScript code. We find that our proposed technique is 5-14 times faster than the deep learning techniques without sacrificing accuracy. We also achieve higher recall than deep learning type inference techniques.

Acknowledgments: This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] 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*, 2017, pp. 758–769.

[2] S. Hanenberg, S. Kleinschmager, R. Robbes, É. Tanter, and A. Stefik, "An empirical study on the impact of static typing on software maintainability," *Empirical Software Engineering*, vol. 19, no. 5, pp. 1335–1382, 2014.

[3] 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 International Symposium on Foundations of Software Engineering*, 2014, pp. 155–165.

[4] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Cscc: Simple, efficient, context sensitive code completion," in *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, 2014, pp. 71–80.

[5] B. Dagenais and M. P. Robillard, "Recovering traceability links between an api and its learning resources," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 47–57.

[6] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 832–841.

[7] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 643–652.

[8] H. Phan, H. Nguyen, N. Tran, L. Truong, A. Nguyen, and T. Nguyen, "Statistical learning of api fully qualified names in code snippets of online forums," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 632–642.

[9] C. M. K. Saifullah, M. Asaduzzaman, and C. K. Roy, "Learning from examples to find fully qualified names of api elements in code snippets," in *Proceedings of the 34th International Conference on Automated Software Engineering*, 2019.

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

[11] R. S. Malik, J. Patra, and M. Pradel, "Nl2type: inferring javascript function types from natural language information," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 304–315.

[12] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 152–162.

[13] W. Fu and T. Menzies, "Easy over hard: A case study on deep learning," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 49–60.

[14] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, "500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow)," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 554–563.

[15] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 11th Joint Meeting on the Foundations of Software Engineering*, 2017, pp. 763–773.

[16] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the generalizability of code2vec token embeddings," in *Proceedings of the 34th International Conference on Automated Software Engineering*, 2019.

[17] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for javascript," in *Proceedings of the 33rd International Conference on Programming Language Design and Implementation*, 2012, pp. 239–250.

[18] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 314–324.

[19] 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*. ACM, 2014, pp. 155–165.

[20] H. U. Asuncion and R. N. Asuncion, Arthur U.and Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 95–104.

[21] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 125–135.

[22] A. T. Nguyen, T. T. Nguyen, and H. V. N. T. N. Al-Kofahi, J.and Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011, pp. 263–272.

[23] A. De Lucia, R. Oliveto, and G. Tortora, "Adams re-trace," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 839–842.

[24] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 375–384.

[25] S. Raemaekers, A. van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *Proceedings of the 10th International Conference on Mining Software Repositories*, 2013, pp. 221–224.

[26] F. Logozzo and H. Venter, "Rata: rapid atomic type analysis by abstract interpretation–application to javascript optimization," in *Proceedings of the 19th International Conference on Compiler Construction*, 2010, pp. 66–83.

[27] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards type inference for javascript," in *Proceedings of the 19th European conference on Object-oriented programming*, 2005, pp. 428–452.

[28] S. H. Jensen *et al.*, "Type analysis for javascript. sas, volume 5673 of lecture notes in computer science," 2009.

[29] P. Thiemann, "Towards a type system for analyzing javascript programs," in *Proceedings of the European Symposium On Programming*, 2005, pp. 408–422.

[30] O. Agesen, "Constraint-based type inference and parametric polymorphism," in *Proceedings of the International Static Analysis Symposium*, 1994, pp. 78–100.

[31] A. Aiken and B. Murphy, "Static type inference in a dynamically typed language," in *Proceedings of the 18th Symposium on Principles of Programming Languages*, vol. 91, 1991, pp. 279–290.

[32] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi, "Tejas: retrofitting type systems for javascript," in *ACM SIGPLAN Notices*, vol. 49, no. 2, 2013, pp. 1–16.

[33] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, "Static type inference for ruby," in *Proceedings of the 24th ACM symposium on Applied Computing*, 2009, pp. 1859–1866.

[34] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks, *Dynamic inference of static types for Ruby*. ACM, 2011, vol. 46, no. 1.

[35] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[36] S. Green, D. Cer, and C. Manning, "Phrasal: A toolkit for new directions in statistical machine translation," in *Proceedings of the 9th Workshop on Statistical Machine Translation*, 2014, pp. 114–121.

[37] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 837–847.

[38] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd Joint Meeting on the Foundations of Software Engineering*, 2014, pp. 269–280.

[39] A. Białecki, R. Muir, G. Ingersoll, and L. Imagination, "Apache lucene 4," in *Proceedings of the SIGIR 2012 workshop on open source information retrieval*, 2012, p. 17.

[40] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[41] X. Rong, "word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[42] R. Mihalcea, C. Corley, C. Strapparava *et al.*, "Corpus-based and knowledge-based measures of text semantic similarity," in *Proceedings of the 18th AAAI Conference on Artificial Intelligence*, vol. 6, 2006, pp. 775–780.

[43] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 26th International Conference on Automated Software Engineering*, 2011, pp. 253–262.

[44] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 2013.