NL2Type: Inferring JavaScript Function Types from Natural Language Information

Author names omitted for double-blind review

Abstract*—JavaScript has become one of the most popular programming languages and is widely used in and beyond the web. Being dynamically typed, it lacks the type safety of statically typed languages, leading to suboptimal IDE support, difficult to understand APIs, and unexpected runtime behavior. Several gradual type systems for JavaScript have been proposed, e.g., Flow and TypeScript, but they rely on developers to annotate code with types, which remains a burdensome and manual task. This paper presents NL2Type, a learning-based approach for predicting likely type signatures of JavaScript functions. The key idea is to exploit natural language information in source code, such as comments, function names, and parameter names, a rich source of knowledge that is typically ignored by type inference algorithms. We formulate the problem of predicting types as a classification problem and train a recurrent, LSTM-based neural model that, after learning from an annotated code base, predicts function types for unannotated code. We evaluate the approach with a corpus of 162,673 JavaScript files from real-world projects. NL2Type predicts types with a precision of 84.1% and a recall of 78.9% when considering only the top-most suggestion, and with a precision of 95.5% and a recall of 89.6% when considering the top-5 suggestions. The approach clearly outperforms JSNice, a state-of-the-art approach that analyzes implementations of functions instead of natural language information, and DeepTyper, a recent type prediction approach that is also based on deep learning. Beyond predicting types, NL2Type serves as a consistency checker for existing type annotations. We show that it discovers 39 inconsistencies that deserve developer attention (from a manual analysis of 50 warnings), most of which are due to incorrect type annotations.

I. Introduction

JavaScript has become one of the most popular programming languages. It is widely used not only for client-side web applications but, e.g., also for server-side applications running on Node.js [1], desktop applications running on Electron, and mobile applications running in a web view. However, unlike many other popular languages, such as Java and C++, JavaScript is dynamically typed and does not require developers to specify types in their code.

While the lack of type annotations allows for fast prototyping, it has significant drawbacks once a project grows and matures. One drawback is that modern IDEs for other languages heavily rely on types to make helpful suggestions for completing partial code. For example, when accessing the field of an object in a Java IDE, code completion suggests suitable field names based on the object's type. In contrast, JavaScript IDEs often fail to make accurate suggestions because the types of the code elements are unknown. Another drawback is that APIs become unnecessarily hard to understand, sometimes forcing developers to guess what types of values a function expects or returns. Finally, type errors that would be detected at compile time in other languages may remain unnoticed in JavaScript, which causes unexpected runtime behavior.

```
/**
  * Calculates the area of a rectangle.
  * @param {number} length The length of the rectangle.
  * @param {number} breadth The breadth of the rectangle.
  * @returns {number} The area of the rectangle in meters.
  *                   May also be used for squares.
  */
getArea: function(length, breadth) {
  return length * breadth;
}
```

Fig. 1: Function with JSDoc annotations. The annotations include comments, parameter types, and the return type.

To mitigate the lack of types in JavaScript, several solutions have been proposed. In particular, gradual type systems, such as Flow [2] developed by Facebook and TypeScript [3] developed by Microsoft, use a combination of developer-provided type annotations and type inference to statically detect type errors. A popular format to express types in JavaScript are JSDoc annotations. Figure 1 shows an example of such annotations for a simple JavaScript function. The main bottleneck of these existing solutions is that they rely on developers to provide type annotations, which remains a manual task.

Previous work has addressed the type inference problem through static analyses of code [4]–[7]. Unfortunately, the highly dynamic nature of languages like JavaScript prevent these approaches from being accurate enough in practice. In particular, analyses that aim for sound type inference yield various spurious warnings.

This paper addresses the type inference problem from a new angle by exploiting a valuable source of knowledge that is often overlooked by program analyses: the natural language information embedded in source code. We present NL2Type, a learning-based approach that uses the names of functions and formal parameters, as well as comments associated with them, to predict a likely type signature of a function. Type signature here means the types of function parameters and the return type of the function, e.g., expressed via `@param` and `@return` in Figure 1. We formulate the type inference task as a classification problem and show how to use an LSTM-based recurrent neural network to address it effectively and efficiently. The approach trains the machine learning model based on a corpus of type-annotated functions, and then predicts types for previously unseen code.

There are four reasons why NL2Type works well in prac-

tice. First, developers use identifier names and comments to communicate the semantics of code. As a result, most human-written code contains meaningful natural language elements, which provide a rich source of knowledge. Second, source code has been found to be repetitive and predictable, even across different developers and projects [8]. Our work exploits the observation that this repetitiveness also extends to natural language elements in code and to their relation with types. Third, probabilistic models, such as the deep learning model used by NL2Type, are a great fit to handle the inherently fuzzy natural language information. Recent advances in natural language processing also find deep neural networks to be particularly effective in solving various prediction tasks [9]–[12]. Finally, our work benefits from the fact that some developers annotate their JavaScript code with types, giving NL2Type sufficient data to learn from.

We are aware of two existing approaches, JSNice [13] and DeepTyper [14], that also use machine learning to predict types in JavaScript. JSNice analyzes the structure of code, in particular relationships between program elements, to infer types. Instead, we consider natural language information, which allows NL2Type to make predictions even for functions with very little code. Moreover, our approach is language-independent, as it does not depend on a language-specific analysis to extract relations between program elements. DeepTyper uses a sequence-to-sequence neural network to predict a sequence of types from a sequence of tokens. Similar to us, they also consider some natural language elements of the code. However, their approach considers only identifier names, not comments, missing a valuable source of type hints, and they frame the problem as sequence-to-sequence translation, while we frame it as a classification problem.

We envision NL2Type to be valuable in several usage scenarios. For code that does not yet have formal type annotations, the approach serves as an assistance tool that suggests types to reduce the manual annotation effort. For code that already has type annotations, NL2Type checks for inconsistencies between these annotations and natural language information, which exposes incorrect annotations, misleading identifier names, and confusing comments. Another usage scenario is improving type-related IDE features, such as code completion or refactoring, for code that does not have any type annotations.

We evaluate NL2Type with 162,673 JavaScript files from open-source projects. After learning from a subset of these files, the approach predicts types in the remaining files with a precision of 84.1% and a recall of 78.9%, giving an F1-score of 81.4%. When considering the top-5 suggested types, precision and recall even increase to 95.5% and 89.6%, respectively. Comparing our approach to JSNice [13] and DeepTyper [14], we find that NL2Type significantly outperforms both approaches, with an increase of the F1-score by almost 30%. When combining NL2Type with JSNice, we find that 27.8% of all correctly predicted types are found exclusively by NL2Type, showing that our approach not only improves upon, but also complements existing work. Beyond predicting likely types for code where annotations are missing, we use NL2Type to check for inconsistencies in existing type annotations. We rank the reported inconsistency warnings by the confidence of the prediction and manually inspect the top 50. 39 out of 50 warnings are valuable, in the sense that developers should fix an incorrect type annotation or improve a misleading natural language element in the code. Finally, the approach is efficient enough for practical use. Training takes 93 minutes in total, and predicting types for a function takes 72ms, on average.

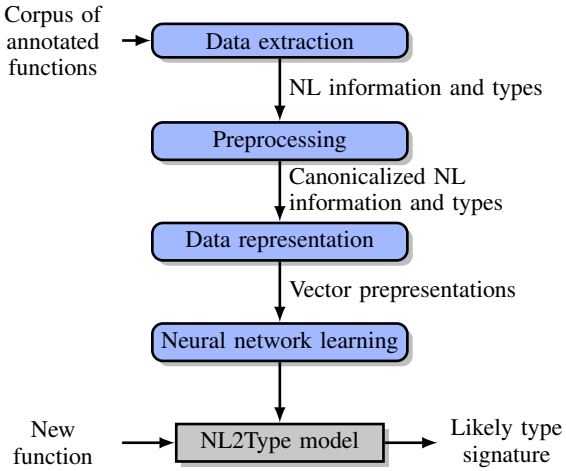

Fig. 2: Overview of the approach.

In summary, this paper contributes the following:

- The insight that natural language information is a valuable, yet currently underused source of information for inferring types in a dynamically typed language.
- A neural network-based machine learning model that exploits this insight to predict type annotations for JavaScript functions.
- Empirical evidence that the approach is highly effective at suggesting types and that it clearly outperforms state-of-the-art approaches.
- Empirical evidence that the approach is effective at finding inconsistencies between type annotations and natural language elements, a problem not considered before.

II. Learning a Model to Predict Types

This section describes NL2Type, our learning-based approach for predicting the type signatures of functions from natural language information embedded in code. Figure 2 gives an overview of the approach, which consists of two phases: a *learning phase*, shown in blue in the top part of the figure, which learns a neural model from a corpus of code with type annotations, and a *prediction phase*, shown in gray in the bottom part of the figure, which uses the learned model to predict types for previously unseen code. To prepare the given code for learning, a lightweight static analysis extracts natural language and type data (Section II-A) and preprocesses these data using natural language processing techniques (Section II-B). Section II-C describes how NL2Type transforms the data into a representation that captures the semantic relations between words, which is then fed into a neural network that

Extracted function data:

n_f	c_f	c_r	t_r
getArea	Calculates the area of a rectangle.	The area of the rectangle in meters. May also be used for squares.	number

Preprocessed function data:

n_f	c_f	c_r	t_r
get area	calculate area rectangle	area rectangle meter may also use square	number

Fig. 3: Example of data extraction and preprocessing.

learns to predict type signatures (Section II-D). Once the model is trained, querying it with natural language information extracted from a previously unseen function yields a likely type signature for the function (Section II-E).

A. Data Extraction

The goal of the data extraction step is to gather natural language information and type signatures associated with functions. To this end, a lightweight static analysis visits each function in the given corpus of code. We focus on functions with JSDoc annotations, an annotation format that is widely used to specify comments and types. For each JavaScript function, the analysis extracts the following:

Definition 1 (Function data): For a given function f, the extracted function data is a tuple (n_f, c_f, c_r, t_r, P) where

n_f = name of the function f
c_f = comment associated with f
c_r = comment associated with return type of f
t_r = return type of f
P = sequence of parameter data

The sequence P of parameter data is a sequence of tuples (n_p, c_p, t_p) where

n_p = name of the formal parameter p
c_p = comment associated with p
t_p = type of p

For example, the upper table in Figure 3 shows the function data extracted from the JavaScript code in Figure 1. We omit the parameter data for space reasons.

B. Preprocessing

To prepare the natural language information extracted in the previous step for effective learning, we preprocess the function data using natural language processing techniques. The goal of this step is to canonicalize natural language words and to remove uninformative words.

At first, we tokenize all natural language data into words. The approach tokenizes the extracted comments, c_f, c_r, and c_p, on the space character. For the extracted names of functions and parameters, n_f and n_p, we tokenize each name based on the camel-case convention, which is the recommended naming convention in JavaScript. For example, the name "getRectangleArea" is tokenized into three words: "get", "Rectangle", and "Area". Beyond camel-case, other tokenization techniques for identifier names [15] could be plugged into NL2Type.

After tokenization, the approach removes all punctuation, except for periods, and converts all characters to lowercase. By converting to lowercase, we reduce the vocabulary size without losing much semantic information. The approach also removes stopwords, i.e., words that appear in various contexts and therefore do not add much information, such as "the" and "a". Finally, the approach lemmatizes all words, i.e., it reduces the inflicted forms of a word, e.g., "running", "runs", "ran", to its base form, e.g., "run".

For our running example in Figure 1, the lower table in Figure 3 shows the function data after preprocessing.

C. Data Representation

To feed the extracted data into a machine learning model, we need to represent it as vectors. The following describes our vector representations of natural language words and of types.

1) Representing Natural Language Information: To enable NL2Type to reason about the meaning of natural language words, we build upon word embeddings, a popular technique to map words into a continuous vector space. The key property of embeddings is to preserve semantic similarities by mapping words that have a similar meaning to similar vectors. For example, assuming we map words into a 3-dimensional space, then "nation" and "country" may have vectors $[0.5, 0.9, -0.6]$ and $[0.5, 0.8, -0.7]$. In practice, embeddings map words into larger spaces; we use vectors of length 100 for our evaluation.

More formally, a word embedding is a map $E : V \to \mathbb{R}^k$ that assigns to each word $w \in V$ in the vocabulary a k-dimensional vector of real numbers. To learn word embeddings, NL2Type builds upon Word2Vec [16], which takes a set S of sentences composed of words in V and learns the embedding of a word w from the contexts in which w occurs. Context here means the words preceding and following w, where the number of context words to consider is a configurable parameter (ten in our evaluation).

NL2Type learns two word embeddings: an embedding E_c for words that occur in comments and an embeddings E_n for words that occur in identifier names. The rationale for having two instead of just one embedding is that identifier names tend to contain more source code-specific jargon and abbreviations than comments. To learn E_c, the set of sentences S consists of all sequences of words in the preprocessed comments c_f, c_r, and c_p. For example, for the word "rectangle" in the lower table in Figure 3, the comments c_f and c_r give two sequences of words in which "rectangle" occurs. For a larger corpus of code, many more such sequences are available. Similarly, to learn E_n, the set of sentences S consists of the sequences of words in the preprocessed identifier names n_f and n_p. For both embeddings, we consider only words that occur at least five times in the training data, to prevent the embedding from overfitting to few contexts.

A possible alternative to learning word embeddings from data extracted from a code corpus would be to use publicly available, pre-trained embeddings, e.g., the Google News word

embeddings.[1] However, such pre-trained embeddings are typically trained on sentences that use a different vocabulary than that found in real-world JavaScript code or on sentences where some words have a different meaning than in source code. For example, words like "push" or "float" may have a different meaning in a programming context than in common usage, while other words, e.g., "int", occur often in a programming context but not at all in common usage.

2) Representing Types: In addition to the natural language information, which is the input to NL2Type, we must also represent the to-be-predicted types as vectors. Given the set T_{all} of all types that occur either as a function return type t_f or as a parameter type t_p in the training corpus, the approach focuses on a subset $T \subseteq T_{all}$ of frequently occurring types. The reason for bounding the size of T is that types have a long-tail distribution, i.e., a few types occur very frequently while many other types occur only rarely (Section V-E). Predicting more frequent types covers a large percentage of all type occurrences, whereas predicting less frequent types is more difficult, as there is less data to learn from. For a specific size $|T|$, we select the $|T|-1$ most frequent types from T_{all} and add an artificial type "other" that represent all other types and that indicates that NL2Type cannot predict the type. The size of T is a configuration parameter and we evaluate its influence in Section V-E. For the evaluation, we consider the 1,000 most common types, including the built-in types of the JavaScript language, e.g., `boolean` and `number`, and custom types, e.g., `Graphics` and `Point3d`.

Given the set T, we represent a type $t \in T$ using a one-hot vector, i.e., a vector of length $|T|$, where all elements are zero except for one specific element set to one for each word. For example, the type `boolean` may be represented by a vector $[0, 0, 1, , 0, .., 0]$ that consists of 999 zeros and a single one.

D. Training the Model

Based on the vector representations of natural language information and types, NL2Type learns to predict the latter from the former. We use a neural network-based machine learning model for this purpose because neural networks have been shown to be highly effective at reasoning about natural language information. Specifically, we adopt a recurrent neural network based on long short-term memory (LSTM) units. Recurrent neural networks are well suited for ordered input data, such as sequences of natural language words. LSTMs are effective for data with both long-term and short-term dependencies. They have been successfully applied to a number of problems in natural language processing that are similar to our classification problem, such as sentiment analysis, which classifies texts into different categories [11], [12], [17]. The following describes the data points used for training the model and the architecture of the neural network.

1) Data Points: We transform the extracted and preprocessed function data into a set of data points. Each data point represents a single type and the natural language information associated with it. We distinguish two kinds of data points, one for return types and another for parameter types.

Definition 2 (Data points): A data point is a pair (N, t) of natural language information N and a type t. Given the function data (n_f, c_f, c_r, t_r, P) of a function, where P is a sequence $P = [(n_p^1, c_p^1, t_p^1), .., (n_p^{|P|}, c_p^{|P|}, t_p^{|P|})]$ of parameter data, we have two kinds of data points:

1) One data point for the return type with:
   $N = (n_f, c_f, c_r, n_p^1, .., n_p^{|P|})$ and $t = t_r$.
2) $|P|$ data points for the parameter types with:
   $N = (n_p^i, c_p^i)$ and $t = t_p^i$.

For example, for the function in Figure 1, there are three data points:

1) For the return type:
   N = (area, calculate area rectangle, area rectangle meter may also use square, length, breadth)
   t = number
2) For the first parameter:
   N = (length, length rectangle)
   t = number
   For the second parameter:
   N = (breadth, breadth rectangle)
   t = number

Given a set of data points (N, t), the task solved by the neural network is to predict t from N. We train a single model for both return types and parameter types because both tasks are similar and it enables the model to learn from all available data. To feed data points into the neural network, we transform each data point into a sequence of input vectors and an output vector, using the vector representations from Section II-C. Intuitively, the input is the sequence of embeddings of words in the natural language information N, and the output vector is the vector representation of the type t.

To formally define the input vectors, consider a helper function $E^* : w_1, .., w_l \rightarrow \mathbb{R}^{l \times k}$ that takes a sequence of l words, maps each word to a vector representation using the embedding function $E : w \rightarrow \mathbb{R}^k$, and then yields the sequence of these vectors. The embedding E refers to E_n and E_c for names and comments, respectively, as described in Section II-C. To ensure that all input vectors have the same length $l \times k$, no matter how many natural language words the static analysis could extract from the source code, the helper function E^* truncates word sequences to a maximum length and pads word sequences that are too short with zeros. We discuss and evaluate the length limits in Section V-E.

Based on this helper function, the input for a data point that represents a return type is the following sequence of vectors (where $\circ$ chains vectors into a sequence):

$$K_{ret} \circ E^*(c_f) \circ E^*(n_p^1) \circ ... \circ E^*(n_p^{|P|}) \circ E^*(n_f) \circ E^*(c_r)$$

Likewise, the input for a data point that represents a parameter type is the following sequence of vectors:

$$K_{param} \circ E^*(c_p) \circ Z \circ ... \circ Z \circ E^*(n_p) \circ E^*(c_r)$$

The vectors K_{ret} and K_{param} are special marker vectors that indicate to the network what kind of type to predict, i.e.,

[1] https://code.google.com/archive/p/word2vec/

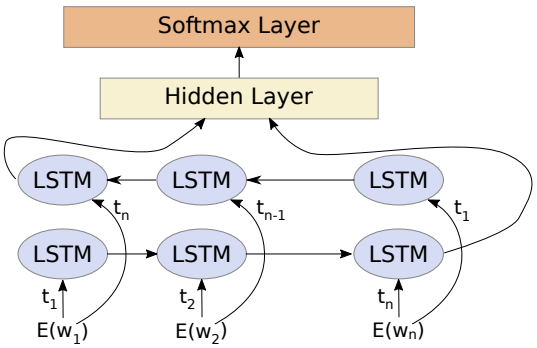

Fig. 4: Architecture of neural network used in NL2Type.

whether the type is a return type or a parameter type. Making the kind of type explicit enables the network to distinguish between both kinds if necessary. The Z vectors are padding vectors of zeros that we use to ensure that the input sequences of return types and parameter types have the same length. In addition to concatenating vectors, each $\circ$ also inserts a vector of ones into the sequence, as a delimiter between the different natural language elements, which helps the network understand the structure of the data.

For instance, recall the three examples of data points given above. The natural language part N of each of them is transformed into a sequence of real-valued vectors based on the embeddings of the natural language words in N. Due to the padding, all three sequences have the same length.

2) Neural Network Architecture: Given the data points described above, NL2Type learns a function $m : \mathbb{R}^{x \times k} \mapsto \mathbb{R}^{|T|}$ where x is the total number of word embeddings in an input sequence and $|T|$ is the number of types we are trying to predict. Using the length limits as set in our evaluation, the network maps a sequence of $x = 43$ vectors of length $k = 100$ to a vector of length $|T| = 1,000$.

To learn the function m, we use a bi-directional LSTM-based recurrent neural network, as illustrated in Figure 4. The network takes a sequence of $\mathbb{R}^k$ vectors, at each step consumes one vector, and updates its internal state (represented by the "LSTM" nodes). After consuming all the vectors for a single data point, the network feeds the internal state through a hidden layer to the output layer. The output layer uses the softmax function, which yields a vector of real-valued numbers in $[0, 1]$ so that the sum of all numbers is equal to one. That is, the output can be interpreted as a probability distribution. During training, the backpropagation algorithm adapts the weights of the network to minimize the error between the predicted and the expected type.

E. Prediction

Once the model is sufficiently trained, it can predict the types of previously unseen functions. To query the model with a new function, we extract and preprocess all natural language information associated with the function, and create one sequence of input vectors for each type associated with the function (i.e., one sequence for the return type and one sequence for each parameter type). Then, each such input sequence is given to the network, which yields a type vector in $\mathbb{R}^{|T|}$. The type vector can be interpreted as a probability distribution over the types in T. For example, suppose that $T = \{number, boolean, function, other\}$ and that the predicted type vector is $[0.6, 0.2, 0.1, 0.1]$. We interpret this prediction as a 60% probability that the type is "number", 20% that the type is "boolean", 10% that the type is "function", and 10% that the type is any other type. If the most likely type is "other", the network essentially says that it cannot predict a suitable type for the given natural language information.

III. Applications

The previous section describes a general model to predict the return type and the parameter types of functions from natural language information. This model has several applications, which we present in the following. All these applications query NL2Type as described in Section II-E.

A. Suggesting Type Annotations

The perhaps most obvious application of NL2Type is to support developers in the process of annotating code with types by suggesting type annotations. Adding type annotations to functions enables an effective use of type systems for JavaScript, such as Flow and TypeScript, and it provides useful API documentation. For the large number of functions in legacy JavaScript code without type annotations, NL2Type can suggest types during the annotation process. To this end, the developer queries the model for each type and uses the predicted type vector as a ranked list of type suggestions.

B. Improving Type-based IDE Features

IDEs use type information for making suggestions to developers, such as how to complete partial code. For example, consider a developer that implements the body of a function and wants to access a property of a parameter of this function. Without type information, the IDE cannot make any accurate suggestions about the property name. For example, the popular WebStorm IDE will simply suggest an alphabetically ordered list of all identifier names used in the current file. NL2Type can improve these suggestions by probabilistically predicting the parameter type of the function, which the IDE can then use to prioritize the suggested property names.

C. Detecting Inconsistencies

In addition to predicting types for functions that are not yet type-annotated, NL2Type can check existing type annotations for inconsistencies. In this scenario, the approach checks whether the natural language information associated with a type matches the annotated type. Finding mismatches is useful for fixing broken type annotations, for changing misleading identifier names, and for improving confusing comments.

Given an annotated function type, we query the NL2Type model with the natural language information associated with the type and compare the type predicted as the most likely with the actual type. To avoid overwhelming developers with

spurious inconsistencies, the approach ranks all inconsistencies by how certain the model is in its prediction. One possible ranking approach would be to consider the predicted type vectors and to rank inconsistencies by the highest probability in each vector. For example, suppose the type vector is $[0.9, 0.025, 0.025, 0.05]$ but the type represented by the first element does not match the annotated type. Based on the type vector, the model appears to be very certain of its prediction and we would rank this inconsistency high. Unfortunately, this naive ranking approach does not work well in practice because neural networks tend to be too confident in their predictions. The underlying reason, as shown by Guo et al. [18], is that for a softmax function over more than two classes, the output of the softmax function is not a true probability distribution.

Instead of ranking inconsistencies by the highest value in the type vector, we compute a more reliable estimate of the network's confidence [19]. The key idea is to use dropout, i.e., to purposefully deactivate some neurons, during prediction and to measure how much it influences the outcome of the prediction. For every sequence of input vectors, we query the model multiple times, each time deactivating some probabilistically selected neurons, and record the predicted type vectors. We then measure the variance of the type vectors and consider a prediction with lower variance to be more confident. Finally, we rank all potential inconsistencies by their confidence and report the ranked list to the developer.

IV. Implementation

We implement NL2Type in Python based on several existing tools and libraries. For the data extraction, the implementation parses every JavaScript file using the JSDoc tool [20], which extracts the comments, the function name, and the parameter names of a function. The preprocessing, including removing stopwords and lemmatization, is implemented based on the Python NLTK library [21]. To convert natural language words into embeddings, we use gensim's Word2Vec module [22]. The neural network that predicts types from a sequence of embeddings is implemented on top of Keras, a high-level deep learning library, using TensorFlow as a backend [23].

Our implementation will be made publicly available when the paper gets accepted.

V. Evaluation

Our evaluation on real-world JavaScript code focuses on the following research questions:

RQ1: How effective is NL2Type at predicting function type signatures from natural language information?

RQ2: How does the approach compare to existing type prediction techniques [13], [14]?

RQ3: How useful is NL2Type for detecting inconsistencies in existing type annotations?

RQ4: What is the influence of configuration parameters on the results?

RQ5: Is NL2Type efficient enough to be applied in practice?

TABLE I: Precision, recall, and F1-score as percentages of NL2Type, with and without considering comments, and of a naive baseline.

Approach	Top-1			Top-3			Top-5		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
NL2Type	84.1	78.9	81.4	93.0	87.3	90.1	95.5	89.6	92.5
NL2Type w/o comments	72.3	68.3	70.3	86.6	81.8	84.1	91.4	86.3	88.8
Naive baseline	18.5	17.3	17.9	49.0	46.0	47.4	66.3	62.3	64.2

A. Experimental Setup

We evaluate NL2Type on a corpus of 162,673 JavaScript files that composed of a publicly available code corpus used in prior work [24] and popular JavaScript libraries downloaded from a content-delivery service [25]. We divide these files into disjoint sets of training files (80%) and testing files (20%). For all files, we extract data points as described in Section II, which gives a total of 618,990 data points. 31.1% and 68.9% of them are for function return types and parameter types, respectively. Not all data points contain all pieces of natural language information. In particular, 20.3% of all data points do not contain a comment c_f or c_p. Given the data extracted from the training files, we train the embeddings and our model, and then use the data extracted from the testing files to evaluate the trained model.

All experiments are run on a computer with an Intel Xeon E5-2650 processor with 48 cores, 64GB of memory, and an NVIDIA Tesla P100 GPU with 16GB of memory. The operating system is Ubuntu 16.04.04 LTS.

B. RQ1: Effectiveness at Predicting Types

1) Metrics: To evaluate the effectiveness of NL2Type in predicting types, we measure precision, recall, and F1-score. Intuitively, precision is the percentage of correct predictions among all predictions, and recall is the percentage of correct predictions among all data points. The F1-score is the harmonic mean of precision and recall. Similar to previous work [14], we report these evaluation metrics for the top-k predicted types, assuming that a user of NL2Type inspects up to k suggested types to find the right one. We also report the top-1 results, which mean that the user considers only the single most likely predicted type.

We define top-k precision as $precision = \frac{pred_{corr}}{pred_{all}}$ where $pred_{corr}$ is the number of predictions where the actual type is in the top-k and $pred_{all}$ is the number of data points for which the model makes a prediction at all. If the model suggests "other" as the most likely type, it indicates that it cannot make a good prediction, and we therefore count it neither in $pred_{all}$ nor in $pred_{corr}$. The top-k recall is defined as $recall = \frac{pred_{corr}}{dps}$ where dps is the number of all data points.

2) Results: Table I shows the precision, recall, and F1-score of the type predictions made by NL2Type. The first row shows the default approach, as described in Section II. When considering the first suggested type only, the approach achieves 84.1% precision with a recall of 78.9%. When considering the top-5 suggested types, the precision and recall

```
/**
  * Get the appropriate anchor and focus node/offset
  * pairs for IE.
  *
  * @param {DOMElement} node
  * @return {object}
  */
function getIEOffsets(node) {
  ...
}
```

Fig. 5: Function with correctly predicted type signature.

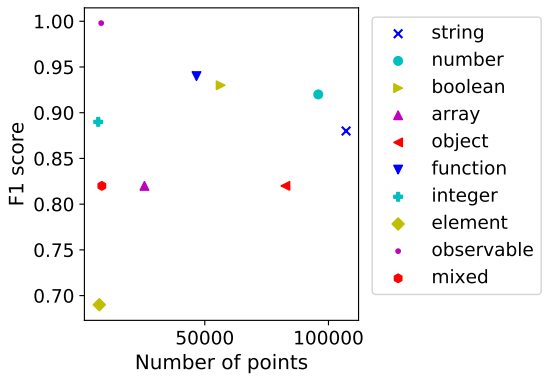

Fig. 6: Relation between F1-score and amount of data available for a type.

increase to 95.5% and 89.6%, respectively. The results for parameter types and for return types are similar to each other, showing that NL2Type is effective for both kinds of types. For example, Figure 5 shows a function for which NL2Type correctly predicts the parameter type and the return type. Note that the parameter type, `DOMElement`, is not a built-in JavaScript type, but nevertheless predicted correctly, presumably from the words "node" and "IE". Overall, these results show that the approach is highly effective at making accurate type suggestions for the majority of JavaScript functions.

To better understand to what extent the prediction depends on the amount of training data available for a specific type, Figure 6 shows for the ten most common types the F1-score along with the number of data points for the type. We find little correlation between the amount of available data and the prediction's F1-score. This result suggests that the amount of data we train NL2Type on is sufficient for commonly used types in JavaScript. Instead, the precision and recall for a specific type is likely to be influenced by the fact that some types are more likely than others to have a comment or name that reveals the type. For example, functions with return type `boolean` often have a function name that begins with "is" or "has". For other types, e.g., "object", it may not be as easy to infer the type from the natural language information.

Because not all functions come with comments, but all functions and their parameters have a name, we also evaluate a variant of NL2Type that does not consider any comments. Instead, the input given to the neural network consist only of the name of the function and the names of its parameters. The second row in Table I shows the results for this variant of the approach. As expected, the precision, recall, and F1-score are lower than for the full approach, because some valuable parts of the input are omitted. However, the approach still makes accurate suggestions that are likely to be useful in practice. We conclude from these results that using comments as part of the input considered by NL2Type is beneficial, but that comments are not essential to the effectiveness of the approach.

We also compare NL2Type to a naive baseline that simply predicts the k most common types every time it is queried. In particular, when asked for the top-1 type suggestion, the baseline always suggests `string` because this is the most common type. The third row in Table I shows the effectiveness of this baseline. NL2Type is clearly better than the baseline, e.g., improving the F1-score for the top-1 suggestion by a factor of 4.5x, which shows the need for a non-trivial type prediction approach.

C. RQ2: Comparison with Prior Work

The two closest existing approaches are JSNice [13] and DeepTyper [14]. Both use the implementation of a function to infer the function's type signature, whereas our approach ignores the function implementation and instead focuses on natural language information associated with the function. JSNice uses structured prediction on a graph of dependencies that express structural code properties, such as what kind of statement a variable occurs in. Similar to our work, they train their model with existing type-annotated JavaScript code. DeepTyper is similar to our work in the sense that they also use a neural network model. However, they train the model with an aligned code corpus, i.e., pairs of TypeScript and JavaScript programs, which are generated from existing TypeScript code.

1) Comparison with JSNice: To compare with JSNice, we download their publicly available artifact [26] and train a model with the same training data as for NL2Type, using the command line arguments given in the artifact's README file. We run the tool with a time limit of two minutes per file and remove any files that exceed that limit from the training corpus of both JSNice and NL2Type. In total, 7,025 files are removed for this reason. Once trained, we evaluate JSNice on our testing set. Because JSNice tries to predict types only for minified files, we minify the testing files using a script provided in the JSNice artifact. All results reported for JSNice are for the top-1 suggestion only, because the JSNice artifact reports only the most likely type suggestion. Beside types, JSNice also predicts other code properties, e.g., identifier names; we consider only the predicted parameter types and return types for our comparison.

The precision achieved by JSNice is 62.5% with a recall of 45.0%, which gives an F1-score of 52.3%.[2] Comparing these results to those in Table I shows that NL2Type clearly outperforms the state-of-the-art approach. In particular, the F1-score of NL2Type is 29.1% higher than that of JSNice, which is a significant improvement. One reason why NL2Type outperforms JSNice is that it successfully predicts types for

[2]Note that our definition of recall is different from the one used in [13], which defines recall as the percentage of data points for which any prediction is made, either correct or incorrect.

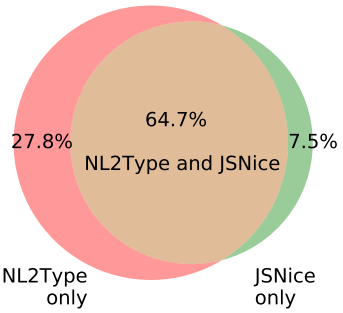

Fig. 7: Venn diagram showing the overlap of data points correctly predicted by NL2Type and JSNice.

```
/**
  * Utility function to ensure that object properties are
  * copied by value, and not by reference
  * @private
  * @param {Object} target Target object to copy
  *                        properties into
  * @param {Object} source Source object for the
  *                        properties to copy
  * @param {string} propertyObj Object containing
  *                             properties names we
  *                             want to loop over
  */
function deepCopyProperties(target, source, propertyObj) {
  ...
}
```

Fig. 8: Incorrect type annotation found by NL2Type: Our model correctly predicts the third parameter to be `object`.

functions independent of the amount of code in the function body, whereas JSNice relies on type hints provided by the function body. To evaluate to what extent NL2Type and JSNice complement each other, Figure 7 shows how many of the correctly predicted types overlap. The figure considers the top-1 predictions only. Of all data points that are predicted correctly by either NL2Type or JSNice, 27.8% are predicted only by NL2Type, while 7.5% are predicted only by JSNice. Overall, these results show that our approach of considering natural language information complements and improves upon prior work that focuses on the implementation of a function.

2) Comparison with DeepTyper: To compare with DeepTyper, we use their publicly available artifact [27] to reproduce their results and to compute our metrics based on their data. A more direct comparison based on the same training and testing data is difficult because DeepTyper requires a corpus of TypeScript code, whereas our implementation is for JavaScript. As we do for JSNice, we compare the top-1 predictions of DeepTyper. We focus the evaluation on their "GOLD" data set, which includes only type annotations that have been added to TypeScript code by the developer, ignoring type annotations inferred by the Typescript compiler.

We find that DeepTyper has a precision of 68.6% and a recall of 44.0%, which results in an F1-score of 53.6%. NL2Type outperforms the existing approach by almost 30%.

D. RQ3: Usefulness for Detecting Inconsistencies

An application of NL2Type that goes beyond predicting types in code without type annotations is as a tool to detect

```
/** Tests to see if a point (x, y) is within a range of
  * current Point
  * @param {Numeric} x - the x coordinate of tested point
  * @param {Numeric} y - the x coordinate of tested point
  * @param {Numeric} radius - the radius of the vicinity
  **/
near: function(x, y, radius) {
  var distance = Math.sqrt(Math.pow(this.x - x, 2)
      + Math.pow(this.y - y, 2));
  return (distance <= radius);
}
```

Fig. 9: Non-standard type annotation detected by NL2Type: Our model predicts the parameters to have type `number`, but the code annotates them as `Numeric`, which is not a legal JavaScript type.

```
/**
  * Calculate the average of two 3d points
  * @param {Point3d} a
  * @param {Point3d} b
  * @return {Point3d} The average, (a+b)/2
  */
Point3d.avg = function(a, b) {
  return new Point3d((a.x + b.x) / 2, (a.y + b.y) / 2,
      (a.z + b.z) / 2);
}
```

Fig. 10: Misclassification: NL2Type predicts a `number` return value, but the code indeed returns an object of type `Point3d`.

inconsistencies in existing type annotations. To evaluate the usefulness of NL2Type for this task, we get a ranked list of potential inconsistencies, as described in Section III-C, and manually inspect the top 50 of this list. We classify each potential inconsistency into one of three categories.

1) Inconsistency. We classify a warning as an *inconsistency* if the source code, the comments, and the type annotations are inconsistent with each other, because at least two of these three are contradictory. Developers should fix these inconsistencies by adapting either the type annotations, the comments, or the code. Figure 8 shows an example of an inconsistency due to an incorrect type annotation. Our model correctly predicts that the type of the `propertyObj` should be `object`, but the code instead annotates it as `string`.

2) Non-standard type annotation. We classify a warning as *non-standard type annotation* if the type annotation refers to a "type" that is not a legal JavaScript type, but may nevertheless convey the intended type to a human developer. For example, Figure 9 shows a function where the parameters are annotated as `Numeric`. However, this type is not a legal JavaScript type, and the developer intended the types to be `number`, which NL2Type correctly predicts. Because NL2Type learns conventions from a large corpus of code, it tends to predict the standard type instead of the non-standard type. To benefit from one of the type checkers built on top of JavaScript [2], [3] and from improved IDE support, developers should replace non-standard types with the corresponding standard type.

3) Misclassification. We call a warning a *misclassification* if the type predicted by NL2Type is incorrect and the code need not be changed in any way. For example, the function in Figure 10 returns an object that represents a point in the 3-dimensional space, as specified in the `@return` annotation.

TABLE II: Classification of potential inconsistencies reported by NL2Type.

Category	Total	Percentage
All inspected warnings	50	100%
Inconsistencies	25	50%
Non-standard type annotations	14	28%
Misclassifications	11	22%

TABLE III: Length limits for inputs processed by the neural network.

	Avg. in data set	Maximum considered	Fully covered data points
Words in function or parameter name	1.6	6	99.9%
Words in function comment	5.9	12	89.9%
Words in parameter or return comment	0.5	10	99.8%
Number of parameters	1.1	10	98.5%

However, the function name and the comment of the function mislead NL2Type to predict `number`. Misclassifications can result because NL2Type has not seen enough data similar to the given natural language information during training or because the code, comments, or identifier names are unusual w.r.t. the training corpus.

Table II shows how the 50 manually inspected warnings reported by NL2Type distribute across the above categories. Most warnings point to code that deserves action by the developer: fixing a type annotation, improving a comment, or changing the code. The percentage of actionable warnings is 78%. We conclude that NL2Type provides a useful tool for checking type annotations for inconsistencies. To the best of our knowledge, our work is the first to show probabilistic type inference to be effective for this task.

E. RQ4: Parameter Selection

1) Parameters for Input Representation: As discussed in Section II-D, each part of the input sequence has a fixed length, and data that are too short or too long are padded with zeros or truncated, respectively. Table III shows the length limits we use and how many of all data points these limits cover without any truncation. For example, we consider up to six words as part of a function or parameter name, which covers 99.9% of all names in our data set. The parameters are selected to cover the large majority of the available natural language data.

2) Parameters for Output Representation: The output of the neural network is a type vector of length $|T|$, which determines how many different types the model can predict. The set T_{all} of all types in our data set contains 11,454 types. Because classification problems become harder when the number of classes increases, and because the frequency of types follows a long-tail distribution, we focus on a subset $|T| \subseteq T_{all}$. Table IV shows how the size of $|T|$ influences the percentage of all data points covered by the considered types. For example, $|T| = 1,000$ covers 94.1% of all data points.

The trade-off in choosing $|T|$ is between precision and recall. Choosing a larger $|T|$ has the potential to increase recall because the model can predict the types of more data points. However, this potential increase of recall comes at the cost of lower precision because the model must choose from more possible types and because the amount of training data quickly decreases for less frequent types. To pick $|T|$, we train and evaluate models for $5 \leq |T| \leq 5,000$ and measure precision, recall, and F1-score for the top-1 prediction. The results in Figure 11 show the tradeoff between precision and recall. The approach reaches the maximum F1-score at $|T| = 1,000$, which is the value we select for the evaluation.

TABLE IV: Impact of the number of considered types on the number of covered unique types and data points.

Number of types	Unique types covered	Data points covered
5	0.04%	61.9%
50	0.44%	81.6%
500	4.37%	91.7%
1,000	8.73%	94.1%
5,000	43.65%	98.6%
10,000	87.30%	99.9%

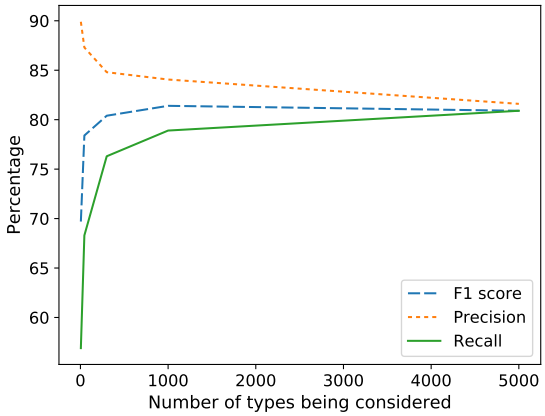

Fig. 11: Effectiveness of NL2Type depending on the number $|T|$ of types.

3) Parameters for Learning: Table V summarizes the values of parameters related to the learning parts of NL2Type. The hyperparameters of the neural networks are selected based on values suggested by previous work and by our initial experiments. We stop training after twelve epochs because it is sufficient to saturate the accuracy.

F. RQ5: Efficiency

The total time taken by NL2Type is the sum of the time for five subtasks. First, data extraction takes 44ms per function, on average, most of which is spent in the JSDoc tool while parsing JavaScript code. Second, data pre-processing takes 23ms per function, on average. Third, learning both the word embeddings takes about 2 minutes in total. Fourth, the one-time effort of training the model takes about 93 minutes. This time is relatively little, compared to some other neural networks, because of the small number of units in the hidden layer. Finally, predicting types for a new function takes the time to extract and pre-process data from the function plus 5ms per function, on average, to query the model. We conclude that

TABLE V: Parameters and their default values.

Parameter	Value
Neural network to predict word embeddings:	
Word embedding size	100
Context size	5
Minimum occurrences of a word	5
Neural network to predict types:	
Hidden layer size	256
Batch size	256
Number of epochs used for training	12
Dropout of model	20%
Loss function for model	Categorical cross entropy
Optimizer	Adam

NL2Type is efficient enough to apply to real-world JavaScript code and to quickly give feedback to developers.

VI. Related Work

a) Type Inference through Program Analysis: Static type inference addresses the lack of type annotations in dynamically typed languages [4]–[7]. Hackett et al. use type inference in a JIT compiler to type-specialize the emitted machine code [28]. TypeDevil observes types at runtime and reports type inconsistencies as potential bugs [29]. Because none of these approaches can guarantee to infer correct types for all values, lots of real-world JavaScript code still lacks type information. Our work addresses the problem by analyzing natural language elements instead of code.

b) Probabilistic Type Inference: Besides NL2Type, we are aware of two other probabilistic type inference approaches for JavaScript: JSNice [13] and DeepTyper [14]. Section V-C discusses and compares with both approaches, showing that NL2Type significantly outperforms both of them.

c) Analysis of Comments: Prior work on analyzing comments focuses on finding inconsistencies between comments and code [30]–[32], on inferring executable specifications for a method [33], on identifying comments that have textual references to identifier names [34], on finding semantically similar verbs that occur in method names and method-level comments [35], and on finding redundant comments [36]. To the best of our knowledge, NL2Type is the first to predict types from comments.

d) Natural Language Information and Code: Code search allows developers to find code snippets through natural language queries [37]–[40]. Similar to our work, these approaches use embeddings of natural language words. Huo et al. propose a neural network that predicts which file is buggy from a natural language bug report [41]. Other approaches predict natural language information from source code, e.g., by predicting function name-like summaries for code snippets [42], or by de-obfuscating minified JavaScript code [13], [43], [44]. In contrast, NL2Type uses the available identifier names, along with comments, to make predictions. Finally, DeepBugs uses natural language information in code, in particular identifier names, to detect code that is likely to be incorrect [45]. NL2Type differs from all the above by exploiting natural language information for predicting types.

e) Embeddings of Code: Related to our use of embeddings to represent natural language words embedded in code, recent work studies how to compute embeddings of code itself. Embeddings can be learned, e.g., from a graph representation of code [46], by randomly walking a control-flow graph [47], by walking an abstract syntax tree [48], or from program executions [49]. Future work could integrate code embeddings into NL2Type to reason about the code of a function, in addition to its comments and identifier names.

f) Statistical Modeling and Learning on Code: NL2Type contributes to a recent stream of research that applies statistical modeling and machine learning to programs. Hindle et al. show that code is "natural" and therefore amenable to statistical language modeling [8]. Dnn4C is a neural model of code that learns not only from tokens, but also from syntactic and type information [50]. Other work uses neural networks to predict parts of code [51], to detect vulnerabilities [52], to generate inputs for fuzz testing [53], [54], to detect and fix syntactic programming mistakes [55], and to predict whether a file is likely to contain a bug [56].

g) JavaScript Analysis: Program analyses for JavaScript include dynamic analyses to find violations of common coding rules [57] and JIT-unfriendly code [58], and to understand asynchronous behavior [59], as well as static analysis to extract call graphs [60]. A recent survey gives a comprehensive overview of JavaScript analyses [61]. Even though we implement NL2Type for JavaScript, we believe that the approach is applicable to other dynamic languages, because we make very few assumptions about the underlying language.

VII. Conclusion

This paper addresses the lack of type annotations in dynamically typed languages. In contrast to traditional techniques, which infer types from the program source code, we tackle the problem by analyzing natural language information embedded in the code. We present NL2Type, a new learning-based approach that feeds identifier names and comments into a recurrent neural network to predict function signatures. The approach yields a neural model that helps annotating not yet annotated JavaScript code by suggesting types to the developer. Our experiments show that NL2Type predicts types with an F1-score of 81.4% for the top-most prediction and of 92.5% for the top-5 predictions, which clearly outperforms existing work on learning to predict types. In addition to predicting missing types, we show how to use the model to identify inconsistencies in existing type annotations. By inspecting 50 warnings about such inconsistencies, we find 39 problems that require developer attention, e.g., because type annotations are incorrect or because they do not match the comments associated with a function. The broader impact of our work is to show that natural language information in code is a currently underused resource that is useful for predicting program properties.

REFERENCES

[1] The Linux Foundation, "2018 Node.js User Survey Report," May 2018.

[2] "Flow," https://github.com/facebook/flow, accessed: 2018-07-22.

[3] "Typescript," https://github.com/Microsoft/TypeScript, accessed: 2018-07-22.

[4] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi, "Tejas: Retrofitting type systems for javascript," in *Proceedings of the 9th Symposium on Dynamic Languages*, ser. DLS '13. New York, NY, USA: ACM, 2013, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/2508168.2508170

[5] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. 16th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 5673. Springer-Verlag, August 2009.

[6] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, "Static type inference for ruby," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1859–1866. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529700

[7] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks, "Dynamic inference of static types for ruby," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 459–472. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926437

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

[9] T. N. Sainath, O. Vinyals, A. Senior, and H. Sak, "Convolutional, long short-term memory, fully connected deep neural networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2015, pp. 4580–4584.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014. [Online]. Available: http://arxiv.org/abs/1409.0473

[11] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, pp. 2873–2879. [Online]. Available: http://www.ijcai.org/Abstract/16/408

[12] J. Y. Lee and F. Dernoncourt, "Sequential short-text classification with recurrent and convolutional neural networks," in *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, 2016, pp. 515–520. [Online]. Available: http://aclweb.org/anthology/N/N16/N16-1062.pdf

[13] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 111–124. [Online]. Available: http://doi.acm.org/10.1145/2676726.2677009

[14] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 12th Joint Meeting on Foundations of Software Engineering*. ACM, 2018.

[15] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2011, pp. 130–154.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[17] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, 2015, pp. 1422–1432. [Online]. Available: http://aclweb.org/anthology/D/D15/D15-1167.pdf

[18] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. International Convention Centre, Sydney, Australia: PMLR, 06–11 Aug 2017, pp. 1321–1330. [Online]. Available: http://proceedings.mlr.press/v70/guo17a.html

[19] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1050–1059. [Online]. Available: http://proceedings.mlr.press/v48/gal16.html

[20] "JSDoc tool," https://github.com/jsdoc3/jsdoc, accessed: 2018-07-22.

[21] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[22] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[23] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[24] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 761–774. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837671

[25] "CDNJS," https://cdnjs.com/, accessed: 2018-07-22.

[26] "JSNice Artifact," https://files.sri.inf.ethz.ch/jsniceartifact/index.html, accessed: 2018-07-22.

[27] "DeepTyper artifact," https://github.com/deeptyper/deeptyper, accessed: 2018-07-22.

[28] B. Hackett and S.-y. Guo, "Fast and precise hybrid type inference for javascript," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254094

[29] M. Pradel, P. Schuh, and K. Sen, "TypeDevil: Dynamic type inconsistency analysis for JavaScript," in *International Conference on Software Engineering (ICSE)*, 2015.

[30] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*iComment: bugs or bad comments?*/," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, 2007, pp. 145–158.

[31] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing javadoc comments to detect comment-code inconsistencies," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, 2012, pp. 260–269.

[32] E. W. Høst and B. M. Østvold, "Debugging method names," in *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2009, pp. 294–317.

[33] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating code comments to procedure specifications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 242–253.

[34] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 112–122.

[35] M. J. Howard, S. Gupta, L. L. Pollock, and K. Vijay-Shanker, "Automatically mining software-based, semantically-similar words from comment-code mappings," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 377–386.

[36] A. Louis, S. K. Dash, E. T. Barr, and C. Sutton, "Deep Learning to Detect Redundant Method Comments," *ArXiv e-prints*, Jun. 2018.

[37] X. Ye, H. Shen, X. Ma, R. C. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 404–415.

[38] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334

[39] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *ICSE*, 2018.

[40] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, 2018, pp. 31–41.

[41] X. Huo, M. Li, and Z. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, 2016, pp. 1606–1612.

[42] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 2091–2100.

[43] B. Vasilescu, C. Casalnuovo, and P. T. Devanbu, "Recovering clear, natural identifiers from obfuscated JS names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 683–693.

[44] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," TU Darmstadt, Tech. Rep. TUD-CS-2017-0296-1, Feb 2018.

[45] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," in *OOPSLA*, 2018.

[46] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 363–376.

[47] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," *CoRR*, vol. abs/1802.07779, 2018.

[48] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *PLDI*, 2018.

[49] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *CoRR*, vol. abs/1711.07163, 2017. [Online]. Available: http://arxiv.org/abs/1711.07163

[50] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *SANER*, 2018.

[51] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[52] Z. Li, S. X. Deqing Zou and, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018.

[53] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 50–59.

[54] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," TU Darmstadt, Tech. Rep. TUD-CS-2016-14664, 2016.

[55] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing Common Programming Errors by Deep Learning," 2015.

[56] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 297–308.

[57] L. Gong, M. Pradel, M. Sridharan, and K. Sen, "DLint: Dynamically checking bad coding practices in JavaScript," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 94–105.

[58] L. Gong, M. Pradel, and K. Sen, "JITProf: Pinpointing JIT-unfriendly JavaScript code," in *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 357–368.

[59] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack javascript," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 1169–1180.

[60] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript IDE services," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 752–761.

[61] E. Andreasen, L. Gong, A. Møller, M. Pradel, M. Selakovic, K. Sen, and C. alexandru Staicu, "A survey of dynamic analysis and test generation for javascript," *ACM Computing Surveys*, 2017.