CedLuJS: Deep Learning for JavaScript programs

Cedric Baumann
ETH Zürich 12-929-170
cbaumann@student.ethz.ch

Lukas Burkhalter
ETH Zürich 12-915-914
burlukas@student.ethz.ch

Abstract

In this work, we address the problem of predicting JavaScript API calls given a code snippet containing a missing method invocation. Statistical Language Models in combination with static analysis techniques have been shown to be a good approach to tackle such a problem [9].

We build a system that proceeds with this architecture and show that the selection of the language model plays an essential role. Our evaluation indicates that a simple n-gram model can outperform a complex RNN-model for our scenario. Furthermore, we show that our system can effectively predict JavaScript API's. The correct completions appear in 60-70% of the cases within the top 5 predictions.

1. Introduction

To this end, programmers rely on libraries and frameworks for realizing complex software projects. On one side, the integration of libraries drastically reduces the development time, but also requires the developers to learn the API's, which can be quite complex. In this work, we want to address this problem for JavaScript programmers by developing an assistant tool that predicts API calls. More precisely, given a JavaScript snippet with a missing method invocation on an object, our tool tries to predict possible completions for this hole.

We follow a similar approach as in the work of Raychev et al. [9]. They use Statistical Language Models combined with static analysis techniques for predicting Android API's. We continue this approach by applying the same techniques to JavaScript programs. We train a language model on a data set of object histories obtained from a big set of JavaScript programs (i.e. GitHub). For extracting the method invocation sequences, we use an inter-procedural points-to analysis for approximating the object histories. We use the trained language models for predicting possible continuations based on the observed method invocation sequence. Our main contributions in this work are:

- A Program Analysis for extracting object histories from JavaScript code.
- An implementation of the proposed system called *CedLuJS*.
- An experimental evaluation of our system, which demonstrates that an n-gram model outperforms an RNN statistical language model in our scenario. Furthermore, we show that our system effectively predicts JavaScript API calls given some code related assumptions. Our results indicate that the correct completion appears in 60-70% of the cases within the top 5 predictions from our system.

2. System Overview

In this Section, we shortly outline the architecture of the CedLuJS system depicted in Figure 1.

The main component is the *History Extraction Module* located in the center, which is responsible for extracting object histories from JavaScript code. To complete this task, the module first performs a points-to analysis on the code and in a second step uses the alias information for performing the approximate object history extraction. In the training phase, we use the module for extracting object histories from our training set of programs. The resulting data consist of method invocation sequences, which then are used to train a language model. In the API-Query phase, the user provides a partial program with a missing method invocation. The system extracts the method invocation sequence for the selected object by using the *History Extraction Module*.

Finally, the system uses the language model to predict possible continuations of the sequence and outputs the predictions with the highest scores to the user. In the following sections, we discuss each component in more detail.

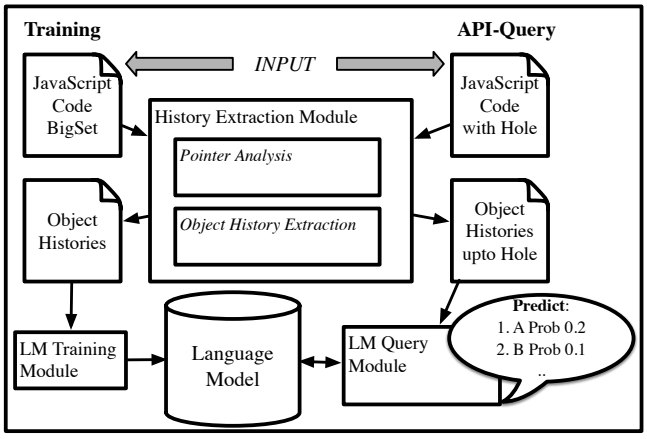

Figure 1: Architecture overview of the CedLuJS JavaScript-API prediction system.

3. Pointer Analysis

The points-to analysis is implemented in an Andersen style, i.e. each variable maps to a set of abstract objects that it may point to. We decided to go with a flow-insesitive, context-insensitive analysis because it was much easier to implement and would reach

a fixpoint in a shorter amount of time, having in mind that we will have to analyze a large number of programs.

Flow-insensitivity means, that we only have one variable store for the whole program and basically don't care about the order in which statements are executed. Context-insentitivity means, that if we have multiple function calls to the same function, we assign the function arguments to the objects that were involved in any call to the function and only analyze the called function one time for all the calls instead of analyzing the function separately for each calling context.

At first, we were a bit worried about the impreciseness of the context-insensitive analysis, but after analyzing some test programs, it turned out that we didn't loose significantly much precision there because of the style that JavaScript applications are written.

3.1 Fixpoint and Function Calls

We used TAJS [5] as a framework for our analysis, this means that if TAJS fails to analyze a program, we also can't run our analysis on it. We first run the TAJS analysis on an input program to construct the flowgraph and callgraph. Then we use the generated flowgraph to follow the program flow and conduct the analysis.

The fixpoint in the pointer analysis is defined as follows: If we iterate through the whole flowgraph and no points-to set behind any variable, function argument, function return object or object property changed, we have reached a fixpoint.

For any function calls for which we don't have the source code, we will just return a new abstract object for every call, i.e. two calls to the same function at different points in the program will always return two different abstract objects. If we have a constructor call, we will generate a new abstract object for each call and put the newly created abstract object in the set of *this* objects of the constructor function. Because of the context-insensitivity of the pointer analysis, every property of the abstract objects that was generated in the same constructor function will be the same set of abstract objects after the constructor call.

3.2 Unreachable Code

We later learned, that there are numerous JavaScript programs in the dataset, that define many functions, but do not call them or use them in an event-based type of setting, so the static analysis can't know that they are called.

For the analysis of the test set of input programs, we also wanted to analyze this unreachable code because the programs mentioned above produced very few object histories due to the fact that many functions were not reachable. But after trying to analyze the unreachable functions, we learned that TAJS does not generate a callgraph for unreachable code, so we had to use another tool [8] for the call graph. As this tool turned out to be less precise than the call graph of TAJS, we used the call graph of both tools in a way that we use the TAJS call graph if it has information for the current function call and use the other imprecise call graph otherwise.

Examples of these impresicions is not providing the called function if the `call` or `apply` method is called on a function variable. As these unreachable functions might have arguments, we have to create a new abstract object for each argument because we also wanted to capture the object histories behind the objects of the function arguments. Furthermore, these abstract objects might also have properties but our analysis knows nothing about them, because we basically don't know anything about the data structure behind these objects.

3.3 Static Objects and Infinite Lattice

There are also global objects like the *document* object that have properties that were not assigned in our analyzed program. We, therefore, added the rule, that if there is a property access of an abstract object that doesn't have an abstract object assigned to the demanded property, we also assign a new abstract object to the wanted property. The motivation behind this rule was to increase the number of abstract object histories that we could get out of the JavaScript code.

One obvious drawback of this rule is, that the number of abstract objects in a program is not bounded anymore. This can be demonstrated in the JavaScript program shown in Listing 1. Both functions `a` and `c` are not called in the main program flow, so we put abstract objects behind the arguments and analyze both functions. As we access the property `e` of argument `b`, we put a new abstract object behind this property and add it to the set behind the function argument `d` of `c`. When we analyze function `c`, we will also put a new abstract object behind the property `f` of the just generated object and put it again in the set of function argument `b` of `a`. It is easy to see that this will never terminate and produce an infinite amount of abstract objects.

This was one of the reasons to introduce a maximum number of flowgraph iterations and just stop after that if we haven't reached a fixpoint yet. For the analysis of the test set programs, this number was set to 12. As for the static objects, we pre-defined the following objects before we conduct the analysis: *document, console, window, screen, location, process, Number, Math, jQuery, navigator, Date*.

```
function a(b){
    c(b.e);
}
function c(d){
    a(d.f);
}
```

Listing 1: A JavaScript program that produces an infinite amount of abstract objects.

3.4 Property Accesses

As JavaScript allows property accesses in the form of `object["property"] = "prop"`, we thought that it might make sense to also store the string value of abstract objects that represent a string object in the JavaScript program, because we couldn't handle this type of property access at all if we didn't do that. However, we don't support any string operations like string concatenation or the substring method. The same goes for number objects. If we directly assign a constant to a variable, we store this value in the abstract object in order to be capable of handling array accesses in the form of `array[10]` or `array[idx]` where idx is assigned to an abstract object with a number value.

Similar to the string operations, we don't support any operations on the number values. This means that in the code snippet shown in Listing 2, we would only assign the String `"value"` to `array[0]` because we can't increment the value of `a`, nor can we tell anything about the range of `a`. Including this type of operations in the analysis would require the use of some numerical domain to represent values of number objects.

```
for(a=0; a < 10; a++){
    array[a] = "value"
}
```

Listing 2: A for loop containing an array property access with the loop variable

If we have a property access that includes a number or string value like the ones described above and we don't know anything about the value because of the use of unsupported operations, we

just ignore it instead of assigning every property of the abstract object because we thought that we could lose too much precision here.

4. Object History

For the object histories, we used a similar approach as described in [9]. For each basic block in the flowgraph, there is an in-mapping and out-mapping that maps abstract object to a set of object histories. When we visit a basic block, we start with the in-mapping and iterate through the statements of the basic block. When we are at the end of the block, we have created the out-mapping and merge it to the in-mapping of all successor blocks. If we have a call to a function for which we also have the source code, we will also add the function to the workset of functions that we have to visit, add the function name to the object histories of the abstract objects that it was called on and take the out-mapping of that function and add it to the current mapping.

As we take the out-mapping of the called function without having it possibly visited first, we will have to iterate multiple times through the flowgraph until we reach a fixpoint and have the final abstract object histories. For each new iteration, we start with empty mappings, but take the out-mappings of the last iteration if we have a function call with source code. The fixpoint is defined as follows: We have a fixpoint if all the in-mappings and out-mappings of two consecutive flowgraph iterations are the same.

4.1 Infinite Lattice

As the length of an object history might be infinitely long, for example if a function calls itself recursively, we had to introduce some limits to reach termination. For that matter, we said that the length of an object history is bounded by 20, and after that, we will ignore all the future function calls in that history.

Furthermore, with the presence of if statements and loops, the number of histories might grow rapidly, especially in the case of nested statements. For that reason, we bounded the number of object histories of each abstract object by 16 in the training set generation. At first, we thought about randomly deleting histories if we get over this bound like it's done in [9], but thought that the randomness will make it harder to reach a fixpoint as we might delete different histories in two consecutive flowgraph iterations. So we decided to just not add new histories after this bound is reached and in the case of a merge just keep the first 16 histories that are merged together.

As seen in the pointer analysis, the number of abstract objects might grow large, especially if we analyze a program that consists of thousands of lines of code. For that matter, we also bounded the number of flowgraph iterations by 12. The idea behind this choice was to get some histories out of very large programs where some histories still might be incomplete but reaching a fixpoint in these programs can't be done in a reasonable amount of time. We decided to visit each loop at most once, so we have one branch where we don't go into the loop and another where we go one time through the body. We decided to only track function calls on an abstract object in the histories. Tracking the function arguments like in [9] probably doesn't give us any advantage here as we are only interested in predicting the API call and not its arguments.

4.2 Known Limitations

During the evaluation of our object history generation, we observed two cases where our system couldn't produce the right output due to the flowgraph of TAJS. Both cases are contained in the code snippet shown in Listing 3.

The first problem was when we had a loop at the very end of the main function. It seems like TAJS inserts event dispatcher nodes at the end of the main function. These nodes have a loop in the flowgraph and also are visited before the body of the loop. The way we handle the number of loop iterations is, that we visit each basic block at most twice. So what happens in the code below is that we first visit the event dispatcher nodes after the loop until we have visited them twice because of the loop in these nodes. Then we visit the body of the while loop but can't propagate the object histories of the loop body to the successor nodes of the loop because we've visited them twice already.

The second problem occurs, if we have an if statement at the end of a loop. In this case, we will first visit the body of the if branch. The last block of the if branch will have the start block of the while loop as its successor. This is because the TAJS flowgraph doesn't have a join node after the multiple branches of an if statement. So we will now again visit the start of the loop because it has a lower order than the else branch, propagate the histories to the successor of the loop and will also visit the if statement for the second time. Like the first time, we will first visit the if branch, then not visit the start of the loop because it was already visited twice and not propagate the histories of the second iteration. The problem is that we will now visit the else branch of the if statement but can't propagate its histories to the successor of the while loop because this node won't be visited anymore.

```
while(some_condition){
    if(some_condition){
        // do something
    } else {
        // do something
    }
}
```

Listing 3: A while loop with and if statement at the end if the loop body

5. Language Models

To predict possible continuations of API function calls, we use statistical language models. Such models are designed for natural language processing, however, they are also suitable for method sequence prediction as presented in the work by Raychev et al. [9]. Similarly, we implement an n-gram and a recurrent neural network for suggesting possible API methods based on the previous invocations. Instead of sequences of words, we deal with sequences of method invocations called object histories $h = m_1, m_2, ..., m_l$. The goal is to compute the probability $P(s_i|h)$ of possible continuations s_i given the observed object history h. Language models allow us to compute these probabilities as they learn a probability distribution over sequences of words. In the following subsections, we focus on the two applied language models and their implementation and close the section with an explanation of our score computation algorithm based on approximated object histories.

5.1 N-Gram

The n-gram model is a simple and scalable language model [7], which acts as a baseline for our system. The probability of the current method invocations given the previous object history is approximated by looking only at the $(n-1)$ previous occurred method invocations.

$$P(m_l|m_1, ..., m_{l-1}) \approx P(m_l|m_{l-(n-1)}, ..., m_{l-1})$$

These probabilities are obtained by counting the number of n-grams, (n-1)-grams, ..., 1-grams in the training data. In our system, we use a simple 3-gram model, which counts up to 3-grams. Our n-gram implementation is entirely based on the SRILM toolkit for

language processing [1]. Since the n-gram model is sparse, we also need to use interpolations for avoiding zero probabilities. We use the Chen and Goodman's modified Kneser-Ney discounting implemented by the SRILM library (*-kndiscount* option). For querying the model in python, we additionally use the pysrilm python wrapper [2].

5.2 LSTM Recurrent Neural Network

With the rise of Deep Learning in recent years, neural networks have become popular for various problems including language modeling. Recurrent neural networks (RNN) specially LSTM-RNN have been shown to perform well for language models and code completion tasks [9, 11]. In this work, we compare the accuracy of recurrent neural networks to the n-gram model for our API completion task in JavaScript. In contrast to the n-gram model, the RNN captures long term relation between the methods in the histories. For our system, we implement an LSTM-RNN, which given the previous occurred context and the method m_i, computes a probability distribution over all possible methods for continuations m_{i+1}. Long Short Term Memory cells (LSTM) are capable of learning long-term dependencies and have been shown to suit well for such problems [3].

Our implementation is based on the RNN tutorial from tensorflow [6]. The RNN operates with a fixed size vocabulary, which contains all possible methods. Each method is represented by its index in the vocabulary because the RNN operates with integers. Before feeding the methods to the LSTM cell, the method ID's pass a word embedding layer, which embeds them into a dense vector representation. The LSTM processes one method vector at the time combined with the memory state of the network and outputs a vector that can be used to compute the probabilities with a softmax regression layer. The memory state updates after each processed method, and is responsible for tracking the long-term dependencies. The cost function that is minimized during training is the average negative log probability of the target method m_i.

$$cost = -\frac{1}{N}\sum_{i=1}^{N} ln(p_{m_i})$$

In literature, the term average per-word perplexity appears more often, which is simply e^{cost}. In order to avoid overfitting on the training data, we add an l2 regularization to the cost function and add a dropout layer after the LSTM layer. For querying the model for possible continuations, we process the observed history method by method and obtain the probabilities for all possible completions.

Vocab limitation: Since our RNN operates on a fixed vocab size, we limit the vocabulary to a certain size by replacing the less frequent occurring methods in the training data by a *unk* token. If the RNN observes an unknown method during the query phases, it simply replaces it by the *unk* token.

5.3 Completion Score Computation

We use language models for predicting possible API continuations in JavaScript code. However, we do not have exact object histories as we use a static analysis approach. Therefore, it may occur that we have multiple object histories for the same object. Let us assume, we want to predict the next API call based on the following observed histories for one object:

```
1. [open] <?>
2. [open] [write] <?>
```

Our system provides three prediction modes: n-gram, rnn and combined. The modes handle the example in the following way.

ngram: In a first step, we use a bigram model for nominating possible candidates based on the observations in the training set. For each history, we obtain a set of candidates and by intersecting them we receive the final candidate set of possible methods. For our example the set is {close, write}. In a second step, the system queries the n-gram model for the probability for each candidate based on the observed history (i.e. $P(c_i|h_j)$). We obtain the following probabilities:

```
(1, write): 0.4, (1, close): 0.2
(2, write): 0.3, (2, close): 0.3
```

In a final step, the system averages the probabilities for each candidate and sorts them to obtain the final score.

```
1. write 0.35
2. close 0.25
```

rnn: Compared to the n-gram, the RNN model does not nominate candidates. The system processes each history directly with the RNN and obtains a large vector of word to probability tuples for each word in the vocabulary. As before, the values are averaged over all histories and sorted according to their score.

combined: The last mode combines the two other modes. By applying both of them, the system obtains two scores. The final score is computed by averaging the two scores.

6. Evaluation

In this section, we focus on the evaluation of CedLuJS. Our main objective is to measure the accuracy of our completion predictions for different JavaScript API's. Furthermore, we are interested in the difference in prediction accuracy dependent on the applied language model (i.e. n-gram vs. RNN). We first outline our setup and proceed with evaluation procedure.

6.1 Training set and Preprocessing

For training the models, we extracted object histories of 100000 JavaScript programs. The test set consists of 244MB data with 2.6 million histories, whereas the validation set consists of 62MB data with 0.65 million histories for tuning the parameters. The vocabulary contained 63198 JavaScript methods. Because of computational limits, we trimmed the vocabulary by replacing less frequent methods by the *unk* token for the RNN training to 20000 and for 3-grams to 60000 respectively.

6.2 Training Parameters

We train two different language models, (i) a 3-gram model with Chen and Goodman's modified Kneser-Ney discounting and (ii) an LSTM-RNN. The LSTM-RNN has various parameters that can be varied, which we outline here in more detail.

The parameters are inspired by previous work on language processing tasks [10]. We use 2 hidden layers with 400 LSTM units each. The word embedding matrix, therefore, has size (vocab size, 400). For avoiding overfitting, we set the dropout probability to 0.5 and the l2 regularization weight to 1e-4. The weights are initialized uniformly at random between -0.05 and 0.05. For the optimization, we apply Stochastic Gradient Descent with a learning rate of 1.0 and a learning decay of 0.8.

6.3 History Extraction and Model Training Phase

The 3-gram model was trained on a Mac Book Pro 2011 with a 2.2 GHz Intel Core i7 and 8 GB RAM. The computational intensive task of training the RNN was performed on Amazon Web Service (AWS) with a g2.2xlarge instance (Ubuntu 14.04 server, 1GPU with 1,536 CUDA cores, 8vCPU Intel Xeon E5-2670). We used

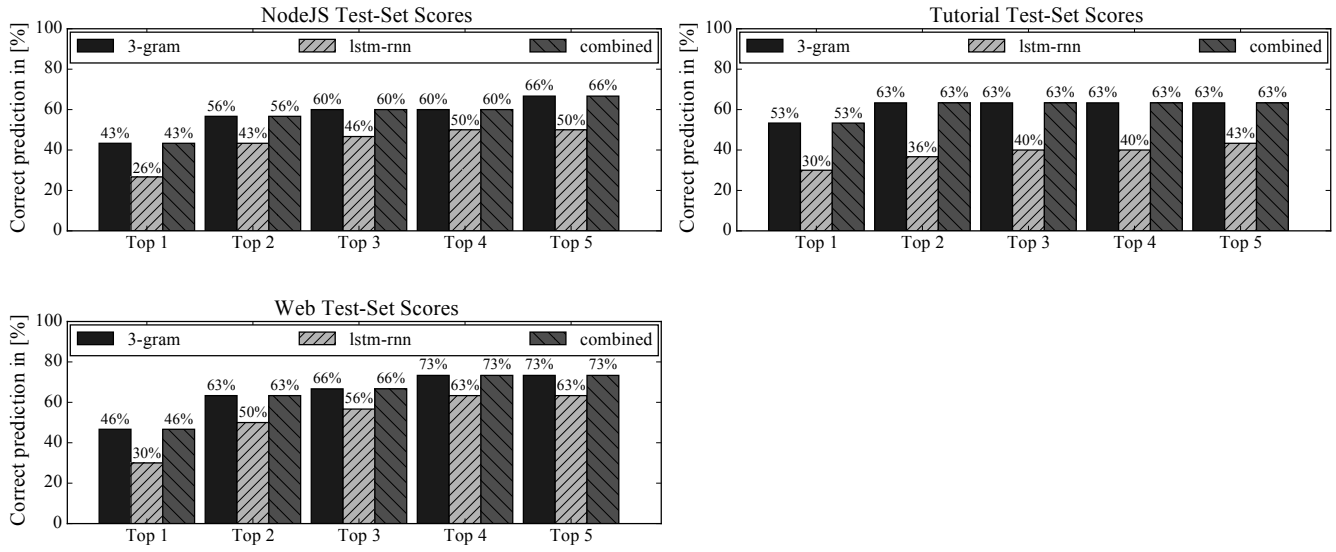

Figure 2: Evaluation of our prediction system on different test-sets. Each set consist of 30 prediction tasks for a different JavaScript domain. The graph plots indicate if the correct result was found within the top n predictions

Task	Execution Time	Output Size
Extract histories	roughly 10 hours	621MB
Train 3-gram	6.3 seconds	23.8MB
Train RNN	10.5 hours	77.4MB

Table 1: Execution time along with output size of the training and history extraction task.

tensorflow with Cuda GPU support. The history extraction was performed on a Lenovo T460s with an Intel Core i7 CPU clocked at 2.6GHz and 20GB RAM.

In table 1, the execution time and the data output size are depicted for each phase. In the first phase, we extracted the object histories from the JavaScript training set. We set a time limit on the execution time of the TAJS analysis and our analysis in order to analyze the test set in a reasonable amount of time. We set it to 8 seconds for the TAJS analysis and 16 seconds for our analysis for each program. Since our analysis is computationally expensive, the task took over 10 hours. There is a big difference in execution time between the 3-gram and the RNN model in the training phase. Despite having GPU and multi CPU for training the RNN, it took about 6000 times longer than for the 3-gram model. This is due to the fact that the 3-gram model simply counts the word contexts whereas the neural net solves a complex optimization problem, which computationally depends on the vocabulary size. In terms of the language model size, the 3-gram model requires about 3 times less space than the LSTM-RNN. Summarized, the RNN is a more complex model than the 3-gram and requires more resources.

6.4 Completion Accuracy Evaluation

For evaluating the completion accuracy of our system in the query phase, we provide three different test sets. Each of the program in the test sets contains one hole for one single object at the end of the program. That is, given object o with a missing method at the end our system computes a top 5 list of possible continuations for object o at this position. The top 5 are ordered according to their probability. This method is similar to the functionality an IDE provides. Imagine an IDE for JavaScript, where our system provides a list of possible API invocations for a given object the user currently uses.

Training Data The tests are divided in three groups of 30 programs, where each is associated to a JavaScript application domain. We divide them into the following three groups:

- **NodeJS**: Tests related to the NodeJS framework [4], which is mostly used for developing back-ends for web applications, but can also be used for regular applications. The framework provides several standard API's. The tests are picked from existing NodeJS applications.
- **Web**: JavaScript programs contained in websites for manipulating HTML documents. The samples are picked from different existing websites.
- **Tutorial**: Own written programs based on tutorials found on the web. The test set simulates a programmer that is new to JavaScript and uses our tool for assistance.

Unfortunately, we cannot test the popular JQuery framework, as our analysis cannot extract histories on it. This is due to the fact that the selectors allow complex queries. As we didn't implement JQuery functions in our analysis, they always just return a new abstract object which makes the produced object histories useless to analyze JQuery code. For the history extraction and predictions tests, our analysis does not analyze dead code because it was given that it will only be tested on the reachable code. Analyzing dead code anyways might produce less precise results which is why we chose not to do it. Therefore, we modified some tests in the program flow for avoiding dead code. Due to this limitations, we were not able to use random programs from the GitHub test data .

Metrics For measuring the accuracy in our evaluation, we give the percentage of correct solutions that are contained in the top n predictions for $n = 1, .., n = 5$. Given the list of top 5 predictions of our system, we look if the correct solution is found in the top

n elements. If the correct solution is in the top 1 (i.e. $n = 1$), our system assigns the highest probability to the correct solution. If the correct solution is in the top 3, the correct method is found within the first three elements ordered by their probability in the list.

Discussion The results for each test group is summarized in Figure 2. The first surprising observation is that the 3-gram model outperforms the complex RNN in all cases. The system was always more accurate in the n-gram mode. The combination of RNN and n-gram does not increase the precision and performs similarly to the n-gram mode. We observed that the extracted histories in the training data are in most cases short and do not contain a lot of variation. A lot of histories contain only a repeated single method invocation. Therefore, the data is much simpler compared to natural language. We suspect that the simple 3-gram model fits better for this kind of data than our complex LSTM-RNN. For natural language the opposite is the case, LSTM-RNN's are shown to perform much better than n-gram [10].

In terms of the accuracy for different groups, we observe similar patterns. In each group, between 60 to 70 % of the cases, the correct solution was found within the top 5. This is not bad, however, these results should be treated with care. We observed that in JavaScript code it's the common case to use the callback and event pattern. As a result, we cannot extract histories for parts within these callback functions as they are treated as dead code. We avoid this kind of invocations in our tests, as we would have no results in the callback case. However, in a practical scenario, such callbacks and events are observed frequently. Furthermore, a lot of web pages use the JQuery framework for accessing HTML elements. Since they introduce complex query functions for accessing the elements, we have problems tracking them in the analysis. Therefore, we would expect lower prediction accuracy if the presented elements occur in the code.

Another observation is that in about 50 % of the tests our system assigns the highest probability to the correct solution, which is surprisingly high. We again suspect that this comes due to the fact that that there is not a lot of variation in the extracted histories for certain API calls.

Summarized, the n-gram model performs better than the LSTM-RNN in our scenario, both in training mode and in querying mode. Although using a simple model, the overall accuracy of n-gram is not bad with 60-70% correct predictions in the top 5, given the code related limitations. However, for practical use, the static Analysis should be improved to support more code constructs in JavaScript.

7. Conclusion

Although our tests produced predictions with reasonable accuracy, we don't think that JavaScript is as suitable as Java for this kind of analysis. One reason for this is the often event based nature of JavaScript programs. The second reason is, that there are not many long-lived objects that produce meaningful histories like in an Android application. This was also the reason why the RNN was outperformed by the n-gram model.

For our tests, we had to pick objects where we knew that they produced a history of a certain length. There were many objects that produced an object history containing two or even one API call, which makes it a bit hard to make an accurate prediction. However, we are satisfied with the accuracy of the predictions in our tests considering the difficulty of the language.

As for further work, one might try to include certain JQuery functions into the analysis in order to analyze this popular framework. Higher precision in the analysis might be achieved by including integer operations or using a flowgraph that is more suitable for this task.

References

[1] Srilm project. `http://www.speech.sri.com/projects/srilm/`, note = [Accessed: 2015-06-06], .

[2] Srilm python api. `https://github.com/njsmith/pysrilm`, note = [Accessed: 2015-06-06], .

[3] Lstm blog-post. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, note = [Accessed: 2015-06-06].

[4] Nodejs. `https://nodejs.org/`, note = [Accessed: 2015-06-06].

[5] Tajs project. `http://www.brics.dk/TAJS/`, note = [Accessed: 2015-06-06].

[6] Google tensorflow. `https://www.tensorflow.org/`. [Accessed: 2015-06-06].

[7] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, Dec. 1992.

[8] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.

[9] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014.

[10] M. Sundermeyer, H. Ney, and R. Schlüter. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 23(3):517–529, Mar. 2015.

[11] W. Zaremba, I. Sutskever, and O. Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.