# Combining Code Embedding with Static Analysis for Function-Call Completion

Martin Weyssow DIRO, Université de Montréal DIRO, Université de Montréal Montreal, Canada martin.weyssow@umontreal.ca

Houari Sahraoui Montreal, Canada sahraouh@iro.umontreal.ca

Benoît Frénay University of Namur Namur, Belgium

Benoît Vanderose University of Namur Namur, Belgium benoit.frenay@unamur.be benoit.vanderose@unamur.be

Abstract—Code completion is an important feature of integrated development environments (IDEs). It allows developers to produce code faster, especially novice ones who are not fully familiar with APIs and others' code. Previous works on code completion have mainly exploited static type systems of programming languages or code history of the project under development or of other projects using common APIs. In this work, we present an approach for improving current functioncalls completion tools by learning from independent code repositories, using well-known natural language processing models that can learn vector representation of source code (code embeddings). Our models are not trained on historical data of specific projects. Instead, our approach allows to learn high-level concepts and their relationships present among thousands of projects. As a consequence, the resulting system is able to provide general suggestions that are not specific to particular projects or APIs. Additionally, by taking into account the context of the call to complete, our approach suggests function calls relevant to that context. We evaluated our approach on a set of open-source projects unseen during the training. The results show that the use of the trained model along with a code suggestion plug-in based on static type analysis improves significantly the correctness of the completion suggestions.

Index Terms—machine learning for software engineering, software maintenance tools, recommender systems

#### I. Introduction

Nowadays, developers rely on features provided by modern Integrated Development Environments (IDEs) to ease their cognitive load and increase their productivity. One purpose of these features is to avoid asking developers to provide information that can be inferred from the available data sources and the current development context [1]. Among these features, code completion is one of the most widely used by, among others, Java developers in Eclipse [2]. Code completion helps developers to write code faster by providing a list of suggestions of possible elements, such as function calls, pertinent to a given context.

There have been a lot of research contributions that attempt to improve code completion systems. Early learning-based approaches focused on completion, specifically for APIs by leverage historical or context data about the system under development [3], [4]. From another perspective, work has been done to exploit natural language modeling for, among other tasks, code completion, based on the idea of code naturalness [5]-[7]. More recently, other approaches have targeted AST representations of the code to perform the APIs calls completion [8]-[12]. In general, the above-mentioned works exploit historical data from the projects used during the evaluation of the system and/or evaluate their systems on specific APIs completion. In the first case, the approaches are not applicable to new projects or projects with short histories, whereas, in the second case, the objective is to predict the calls to APIs' methods. Although the obtained results are convincing, these approaches have shown to be efficient only for popular libraries.

In this paper, we propose an approach for improving function calls completion by learning models from independent code repositories. Our goal is to allow call completion not only with API functions, but also those of the project under development. More specifically, we consider each method as a natural text paragraph containing a sequence of function calls. Then, using a well-known word embedding model, we learn vector representation of variable-length sequences of these paragraphs. Our approach is based on the assumption that there exist recurring patterns of function-call sequences among the code repositories and that these patterns capture semantics about higher-level concepts. Our approach is intended to abstract these high-level concepts and use them to improve function-call completion by comparing the call site context with the huge amount of contexts learned from the repositories. We use the learned models for function-call completion by combining them with a static analysis performed on the project under development. A type-based static analysis allows us to retrieve the list of possible function calls given a completion site. The goal is to rank this list using the embedding model with the most likely calls at the top.

To evaluate the proposed approach, we used a corpus of more than 14,000 Java projects from which we extracted more than 10 millions function sequences to train our models. To test our completion strategies, we selected 10 projects, not considered for the training, and having more than 160.000 call sites to complete. The results of our evaluation show, on the one hand, that the ranking of a list of possible candidates retrieved by static analysis improves the completion precision of the static analysis tool, for 9 of the 10 projects, by up to 135% reaching 85% Recall@10. On the other hand, given the promising results obtained, we compared our approach with state-of-the-art language models used in code completion. We considered two variants of language models: (1) a ngram language model and (2) a n-gram model augmented with a cache component that has shown to outperform LSTM-based deep learning models in source code modeling [7]. In the former, we show that the n-gram model is not able to improve the ranking of the static analysis tool. While, in the latter, even though the n-gram model stores cache information about the system under development, the results show that our approach is much more efficient in term of Recall@10 and MRR. Finally, we found that it takes between 700 ms and 800 ms, on average, to produce completion suggestions for a call site. This makes our approach usable in a real programming setting.

The rest of the paper is structured as follows. In Section II, we introduce the word embedding and language models that we use in this work. Section III presents the general approach for building a code completion system based on embedding models and explains its integration within an existing typing-based tool. We present the evaluation setup in Section IV and report on the results in Section V. Later, we discuss the related work in Section VI. Finally, we draw conclusion and list future work directions in Section VII.

#### II. BACKGROUND

In this section, we review n-grams language models. Then, we describe word embedding models to learn vector representations of variable-length texts. In Section V-A, we use n-grams to quantify the predictability of function sequences in source codes. In Sections V-B and V-C, we compare n-gram-based and word embedding approaches for function-call completion.

# A. n-gram Language Models

Language models (*LMs*) assign probabilities to sequences of words. The main purpose of such model is to capture regularities in a large training corpus as leverage to some downstream task(s) (*e.g.*, speech recognition, spelling correction, text generation...).

Considering a word sequence  $w_1, w_2, ..., w_n$ , a LM assigns a probability  $P(w_1, w_2, ..., w_n)$  to the sequence. Such a probability is hard to compute because usually long sequences of words are not observed in a training corpus. Therefore, we use n-gram language models to approximate  $P(w_1, w_2, ..., w_n)$ .

n-gram language models assign a probability to a word w given an history of size n-1. n-gram LMs assume that the occurrence of a word depends only on the previous words. In other words, a n-gram model is a **Markovian approximation** of order n-1:

$$P(\mathbf{w}_1^n = w_1, w_2, ..., w_n) \approx \prod_{k=1}^n P(w_k | w_{k-n+1}^{k-1}).$$

The simplest approach to estimate these word probabilities is a maximum likelihood estimation (MLE) over the raw counts of words in the corpus. In practice, MLE is not used to avoid the model to assign zero probability to unseen sequences of words. Instead, smoothing techniques are used and assign part of the total probability mass to unseen *n*-grams. For

instance, **Kneser-Ney** is one of the most known smoothing techniques and is very efficient [13], [14].

1) Language model evaluation: A good language model predicts with a low-level of uncertainty the content of an unseen piece of text. The level of uncertainty of a language model can be measured by the **cross-entropy**. Given a n-gram language model L and a word sequence  $\mathbf{w}_1^n = w_1, w_2, ..., w_n$ , the cross-entropy is computed as:

$$H_L(\mathbf{w}_1^n) = -\frac{1}{n} \sum_{i=1}^n \log P(w_i | w_{i-n+1}^{i-1}).$$

For the case of a n-gram model, the cross-entropy is the average number of bits required to predict the  $n^{th}$  word given the n-1 previous words. Consequently, a model that has low entropy on a given piece of text has a low-level of uncertainty and predicts with confidence the content of the text.

# B. Distributed Representations of Words

Word embedding is a technique commonly used in natural language processing (*NLP*) to learn a mapping of words into an high-dimensional vector space. The notion of word embedding is highly related to **distributional semantics**. That is quantifying some semantic similarities between words or concepts that appear frequently in the same context in a large corpus of textual data. Two words that have close vector representation are meant to be semantically similar. For example, it is likely that *senate* and *politic* would be close.

One of the most-known framework for learning distributed representation of words is **Word2vec** [15], [16]. Nevertheless, there is no inherent scheme to the model to learn embedding of sequences of words. Such an approach would, for example, allow us to compute the similarity between two text documents (e.g., of variable-length). Paragraph vector model aims to tackle this problematic by learning vector representation of variable-length texts.

#### C. Paragraph Vector Embedding Model

The paragraph vector (PV) model, i.e., **Doc2vec**, is an extension of Word2vec proposed by Le and Mikolov [17]. PV models learn vector representations (paragraph vectors) of sequences of textual data of variable size (document, phrases, news article...). In this model, each input sequence has a unique corresponding paragraph vector that is learned along with the word vectors. Paragraph vectors are not just concatenation and average of word vectors contained within the paragraph. Instead, paragraph vectors are asked to contribute to a predictive task as for words in Word2vec.

The advantage of the PV model over Word2vec is that the model is able to learn representation of variable-length texts. As result of the learning phase, the paragraph vectors can capture semantic properties about whole sequences. This model has shown to be useful in topic modeling and several NLP tasks [18]–[20].

#### III. OUR APPROACH

To illustrate the rationale behind our approach, let us consider the situation in which, Ulwazi, a Java developer, is writing the method in Listing 1.

```
1 // ...
2 public long size() throws IOException {
3    if (!file.isFile()) {
4        throw new FileNotFoundException(
5        file.toString());
6    }
7    return file.? // prediction (ctrl+space)
8 }
```

Listing 1. Motivating example

Consider also that Ulwazi is coding in an IDE that incorporates, among other features, a code completion plug-in such as Eclipse content assist¹ that suggests function calls. In line 7, after she types "." the plug-in is invoked, and the latter will provide a suggestion list of possible items including function calls that could follow "file.". The plug-in exploits static environment information about the currently opened code artefact (e.g., imports, language typing...). The produced suggestion list is generally exhaustive, often long, and ordered alphabetically. Thus, it is more likely that the correct suggestion will not appear at the top of the list, and developers like Ulwazi will waste a valuable time browsing through the list.

Therefore, our objective is to alleviate the burden of developers by providing completion suggestion lists that are: (1) of limited size, and (2) ordered by pertinence so that the correct suggestion is likely to appear in the top positions.

Our approach is based on the hypothesis that there exist recurring function-call patterns in large corpus of source code. Those patterns embody some semantics about high-level concepts, which may appear in different programs with slight linguistic variations. For instance, coming back to the example of the method "size" that computes the size of a file, the first step consists in checking whether the input is a file, by calling, for example, a function ".isFile()". If it is not, one may want to raise an exception with a representation of the file by calling ".toString()". The final step is to call a function ".length()" that outputs the size of the file. Our approach makes the assumption that such sequences of function calls are totally or partially recurrent among a lot of projects and that they capture most of the semantics of some higher-level concepts (in our case, "get the size of a file"). By comparing the previous function call sequence (including the method name) "size, isFile, toString" with function call sequences abstracted by a word embedding model, it would be possible to determine that "length" is the most probable function call that comes after "file.".

In this work, we propose an approach to learn those highlevel concepts and their relationships by training an embeddings model (e.g., paragraph vector model) on a big corpus of code. Once the model trained, we can take advantage of it for the function completion task by ordering the list of possible function calls retrieved with a type-based static analysis tool. We describe the learning process in Section III-A and the function completion in Sections III-B. Figure 1 illustrates both processes.

# A. Learning Concepts from Code

As discussed in the previous section, sequences of function calls embody a great part of the semantics of code. Therefore, it might be a good way to use these sequences as textual representations of the source code. Conversely, syntactical tokens such as if, else, for or a parenthesis carry less domain-specific information and considering them could lead to introducing a lot of noise in the learning, especially in the context of function call completion. It is also important to define how do we cut the code in order to produce sequences of functions and learn the paragraph vectors. We propose to limit the scope of a sequence to a method declaration and its body as for the size method. A method can be seen as a paragraph that is designed to deal with a particular concern, as we would do in a text. And, it is more likely that functions sequences within a small scope are more recurring than in a broader scope, e.g., a whole class. Furthermore, limiting the sequences to a relatively small scope allows the model to learn specific and precise concepts.

Our approach is also not limited to using sequences of functions. One alternative would be to consider subtokens of function names instead of the full functions names. Given a function name, the subtokens are words contained within it. For example, if we have a function that is called "convert-DateToString", we tokenize the camel case and the resulting subtokens are "convert, date, to, string". This approach has been used in previous works on source code modeling [12], [21]–[23]. It has shown to be useful to summarize code snippets and for suggesting out-of-vocabulary code tokens (names that does not appear during the training phase of a model).

The learning process is described in the upper part of Figure 1. The first step extracts function sequences from a corpus of code. Then, the sequences are used as input of a paragraph vector model (*Doc2vec*). Finally, the model learns high-dimensional vector representations of the function sequences (*paragraph vectors*).

#### B. Function Completion with Static Analysis

In most recent works on code completion, the authors proposed different approaches where a suggestion list of most-likely candidates is built for a specific context using a standalone model [7], [23]–[25]. The main drawback of such approaches is that they do not guarantee that all the suggestions are feasible in the current project. In this work, we propose an approach where, instead of building a list of suggestions with a standalone model, we rank the list of all possible function calls retrieved using a type-based static analysis tool with a learned model. The advantage of this approach is that the static analysis provides functions depending on the packages/libraries imported in the code file in which the method is implemented. Also, it filters the calls

<sup>&</sup>lt;sup>1</sup>https://www.eclipse.org/documentation/

#### Learning phase

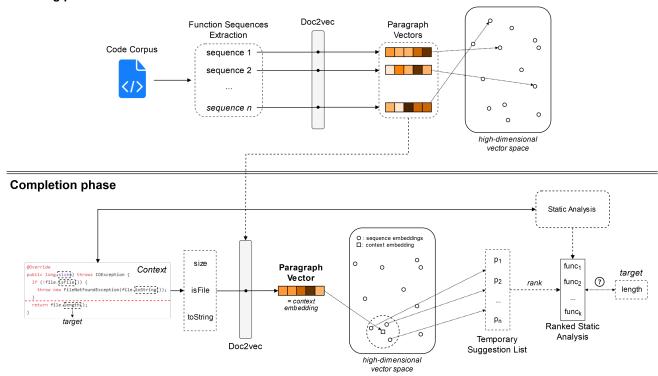


Fig. 1. Approach - General framework

that are not type compliant *w.r.t* the completion site. Therefore, this approach guarantees that the suggestion list will only contain tangible function calls.

The completion process is described in the lower part of Figure 1. Again, we consider the scenario of Section III in which Ulwazi is implementing a class method. The completion process is designed in five steps:

- Extraction of the context. The context is made of the name of the method under development followed by the sequence of calls, in the method body, preceding the call site that triggers the completion. If the call to complete is the first of the method, then the context is only made of the name of the method.
- 2) **Static analysis**. A static analysis is performed to retrieve all possible function calls given the context of the completion site.
- 3) **Inferring a paragraph vector**. Using the previously trained model, we infer a vector representation of the context (*context embedding*).
- 4) **Building a temporary suggestion list**. We use the embedding of the context to retrieve the closest paragraph vectors in the model. This can be done by finding the paragraph vectors that have the greatest cosine similarity to the context vector. Since paragraph vectors correspond to sequences of functions, this is like retrieving the most similar sequences of functions to the context. Then, we build a temporary suggestion list using the retrieved sequences. We add functions of the retrieved sequences

to the suggestion list and stop when the list has reached its maximum allowed size or a threshold of the similarity score.

5) Ranking the static analysis. Using the temporary suggestion list, we rank the possible function calls provided by the static analysis. We iterate over the functions in the temporary list. For each function, if it appears in the static analysis then we add it to the final suggestion list. We stop the process when the list has reached certain size or when the PV model is not able to find more similar sequences to the context.

As we will see in the evaluation, the steps 3, 4 and 5 can be generalized to any model that is able to provide a list of candidates for a completion site. This allows us to compare our approach using a paragraph vector model with state-of-the-art n-gram language models.

#### IV. EVALUATION SETUP

Previous works have shown that source code is (*locally*) repetitive and predictable using statistical language models [5], [6]. Recent works have found that variable and function identifiers are the main responsible for the high-level of entropy of code and that syntax tokens artificially increase the source code predictability [26]. Thus, one of the key challenges of learning high-level concepts from codes using sequences of functions lies in the high-level of unpredictability of those sequences. This leads us to address the following research questions:

TABLE I GITHUB JAVA CORPUS STATISTICS [27].

# Projects	LOC	Tokens			
14.785	352.312.696	1.501.614.836			

# RQ1 [Replication]: How repetitive and predictable are function sequences in source code?

We reproduce previous works on naturalness of software [5], [26]. We check whether our datasets satisfy the naturalness hypothesis introduced by *Hindle et al.* [5]. Then, we ensure that our datasets have a level of cross-entropy in the same order of magnitude than in *Rahman et al*'s experiments [26]. To estimate n-gram language models we use SLP, a toolkit that provides fast estimation and manipulation of n-gram models<sup>2</sup>. As a first step, we estimate n-gram language models using our training sets for  $n \in [2, 10]$ . Then, we compute the cross-entropy on our test set.

# RQ2: Can we use the paragraph vector model in order to rank accurately the function calls retrieved by static analysis using Eclipse JDT Core<sup>3</sup>?

The ranking is done using the process defined in Section III-B. We used Eclipse JDT Core as a static analysis tool to retrieve the possible function calls for each completion site that appears in the 10 bold projects in Table II. We evaluate our approach using metrics defined in Section IV-D.

# • RQ3: How does the paragraph vector model performs compared to state-of-the-art n-grams models on function completion?

We compare the results obtained in RQ2 with state-of-the-art n-gram models used by  $Hellendoorn\ and\ Devanbu$  [7]. The authors have shown that their implementations of n-gram models perform well for source code modeling and code prediction. Furthermore, we also compare our approach with n-gram model augmented with a cache component that allows to learn local information about the project under development at test time.

# A. Data Source

We use the GitHub Java Corpus [27] consisting of more than 14,000 open-source java projects collected from Github. The corpus' statistics are presented in Table I.

Before forming the training set, we removed 20 projects from the original corpus to build a test set. We select these projects based on their high popularity in Github and to cover a broad range of application domains. We also considered the diversity in size. Table **II** shows statistics for each test project, *i.e.*, the number of methods declared in each test project, the total number of call sites in these methods, and the percentage of function vocabulary that appear in the training dataset. We

TABLE II
TEST PROJECTS USED IN THE EXPERIMENTS ORDERED BY DECREASING NUMBER OF FUNCTION CALLS. COVERAGE IS THE PERCENTAGE OF FUNCTIONS THAT APPEAR IN THE TRAINING SET.

Name	# Method Decl	# Function Calls	Coverage	
aws-sdk-java	245.430	1.799.530	76%	
hadoop-common	46.449	347.093	88%	
spring-framework	44.433	332.121	88%	
hibernate-orm	30.867	278.124	86%	
neo4j	33.939	230.914	80%	
jclouds	24.746	196.070	85%	
cassandra	23.398	188.773	83%	
druid	15.674	123.341	87%	
gradle	26.913	120.123	84%	
spring-security	13.750	96.950	84%	
netty	14.326	72.754	82%	
mongo-java-driver	7573	35.836	84%	
twitter4j	2323	13.365	99%	
clojure	1966	13.020	94%	
antlr4	2222	11.053	84%	
junit	2522	8144	94%	
hystrix	1090	5790	78%	
facebook-android-sdk	1453	5689	80%	
android-async-http	198	675	5790 78% 5689 80%	
game-of-life	37	128	64%	

use the whole 20 projects to answer RQ1 and limit ourselves to the 10 projects in bold to answer the remaining questions. These 10 projects allow us to test the completion for more than 160.000 call sites.

For the training of the *n*-gram models and the paragraph vector model, we extract more than 10 millions function sequences from the filtered corpus. In Table **III**, we specify the number of tokens and types (*i.e.*, unique tokens) with and without a minimum count parameter. This parameter is used with both types of models to ignore functions that occur less than a specified threshold. The ignored functions are replaced by a common token <unk>. We can observe that when using this minimum count parameter, the number of types decreases drastically (around 7% of types are kept), but the total number of tokens does not decrease that much. This means that there is a significant amount of types that are not frequent among all projects and considering them in the learning phase could lead to learning a lot of noise.

In addition to this first training set, we also consider a variant of the data which consists of subtokens of the function names. We use this second training set for RQ1 to investigate the impact of representing functions as word subunits for source code modeling. Tokenizing the function names considerably reduces the size of the vocabulary. Furthermore, it is more likely that each subtoken of a function such as "convertDateToString" (i.e. "convert, date, to, string") appears

<sup>&</sup>lt;sup>2</sup>https://github.com/SLP-Team/SLP-Core

<sup>&</sup>lt;sup>3</sup>https://www.eclipse.org/jdt/core/

#### TABLE III

TRAINING SET 1 (full function names). STATISTICS WITH AND WITHOUT MINIMUM COUNT PARAMETER. TOKENS CORRESPONDS TO THE NUMBER OF METHOD DECLARATIONS AND FUNCTION CALLS IN THE DATASET.

TYPES IS THE NUMBER OF UNIQUE TOKENS.

	# Function sequences	Tokens	Types	
no min count	10.702.667	86.219.928	3.141.457	
min count (20)	10.702.667	74.820.025	222.730	

TABLE IV
TRAINING SET 2 (functions subtokens).

	# Function sequences	Tokens	Types	
no min count	10.702.667	183.334.996	165.110	
min count (5)	10.702.667	183.160.006	71.460	

frequently in a corpus than the whole function name. Table IV shows statistics of this second training set. We can observe that the number of types is significantly lower than in the first training set and that the minimum count parameter has almost no impact on the total number of tokens.

# B. Evaluating the Paragraph Vector Model

For this experiment (RQ2), we first retrieved the list of possible function calls for each call site of each test project using Eclipse JDT Core. Then, we evaluated each call site by extracting the context of the call  $f_c$  to be predicted. The context is made of the previous calls preceded by the method name  $(m, f_1, f_2, ..., f_{c-1})$ . Next, we followed the process described in Section III-B to rank Eclipse's static analysis. For the sake of evaluation, we fixed the maximum size of the suggestion lists to 10.

To tune the hyper-parameters of the PV model, we evaluated several configurations of the model on our test set with commonly used values of hyper-parameters. We found out that the following configuration works the best for our task: PV-DBOW with dimension of the embeddings of 300, a window size of 15, a threshold of 20 for minimum word counts and a hierarchical softmax as training algorithm.

## C. Evaluating the n-gram Language Model

For this research question (RQ3), we adapted the steps 3, 4 and 5 of our approach (see Section III-B). The temporary suggestion list is built using a n-gram model. Given a context, the model outputs the most-likely function calls that should follow that context. Then, we rank Eclipse's static analysis using the same call sites and the same process than for the PV model in RQ2. For a fair evaluation, the maximum size of the suggestion lists is also set to 10.

We trained n-gram models used in *Hellendoorn and Devanbu*'s work [7]. We used Jelinek-Mercer smoothing that yields the best performances in their paper. Then, we tuned the n-gram model order and the vocabulary cut-off value. We found that a model order of 5 and a vocabulary cut-off of 20 gives the best performances at test time. We also considered

a variant of the n-gram model augmented with a cache component and used the same hyper-parameters configuration than for the n-gram model.

# D. Effectiveness Metrics

The evaluation aims to determine whether a learned model is able to efficiently provide good function call suggestion lists. To evaluate our systems, we consider that a set of suggestions is relevant if it reflects the user's need. That is, the suggestion list contains the correct function call that follows a given context.

To measure the relevance, we calculate two widely-used metrics, recall at k (R@k) and the mean reciprocal rank (MRR). As there is a unique valid suggestion for each call site, R@k for a test project is the number of times the expected function call appears in top-k of suggestion lists divided by the number of tested call sites.

The second metric we report is MRR. The reciprocal rank is given by the inverse of the rank of the first relevant suggestion in the result of a test sample. Mean reciprocal rank for a test set T is

$$MRR = \frac{1}{|T|} \sum_{i=1}^{|T|} \frac{1}{rank_i}$$

where  $rank_i$  is the rank of the first relevant suggestion in the *i*-th test sample. For example, if on average, the relevant function call appears at rank 2, the MRR is 0.5.

#### E. Replicability Package

To facilitate the replication of our experiments, we share a Github repository that includes the artifacts to train and evaluate both n-gram and paragraph vector models. The repository provides extensive description on how to replicate each RQ. It also contains links to download the training and test sets<sup>4</sup>.

#### V. EVALUATION RESULTS

In this section, we present the results of our experiments and answer the research questions. For the sake of clarity, we present the global results for questions RQ2 - 3 in Table V.

## A. Naturalness of Function Calls (RQ1)

Figure 2 shows the average cross-entropy on the 20 test projects including and excluding out-of-vocabulary (OOV) functions. The x axis represents the n-gram model order  $(n \in [2, 10])$ , which defines the size of the context considered by the model to produce a prediction. A high cross-entropy means that the next token is difficult to predict, while a low cross-entropy means that the code is easier to predict.

The cross-entropy for the full function names is much higher than in *Hindle et al.*'s work. But it decreases by excluding OOV functions and it gets closer to the cross-entropy they reported on a Java corpus of ten projects. Furthermore, we observe that function names' subtokens have a significantly lower cross-entropy and that excluding the OOV functions has no impact. The no decreasing of the cross-entropy when

<sup>4</sup>link may reveal authors' identities

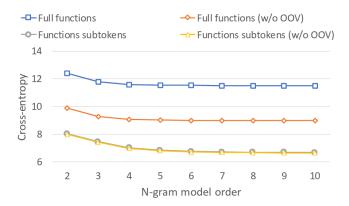


Fig. 2. Comparison of the average cross-entropy on the 20 test projects for full function names and names' subtokens with respect to the order of the n-gram model.

excluding OOV functions means that almost all subtokens in the test projects appear in the training set. This means that sequences of functions subtokens are more predictable than sequences of full function names. We conclude that the naturalness hypothesis is more prevalent using subtokens of function names. However, relaying on only subtokens may make us lose important information about the sharing of semantics across functions. For this reason, we will use the full names to answer RQ2 and RQ3.

In their work, *Hindle et al.* estimated *n*-gram models on a Java corpus that includes all tokens present in the code. *Rahman et al.* addressed the same replication work and conclude that syntax tokens are much more present than identifiers in programming languages and that they make the code artificially predictable. The levels of cross-entropy that we report are closer than those reported in *Rahman et al.*'s work. That is, including only functions as training data drastically decreases the predictability of the code.



Fig. 3. Comparison of the cross-entropy for 5 test projects with full function names and names' subtokens with respect to the order of the n-gram model.

To illustrate our words, we report, in Figure 3, the cross-entropy on 5 test projects using full function names and their subtokens. We observe that the subtokens approach yields a decreasing of the cross-entropy for all test projects. In addition

to that, we can observe that some projects such as *twitter4j* and *junit* have a very low cross-entropy, even when considering full names. This can be explained by the high vocabulary coverage of these two projects (*see Table II*). Therefore, we suspect that the paragraph vector and *n*-gram models will perform well on projects that have a high vocabulary coverage.

To answer RQ1, we have shown that the sequences of function calls included in our training set are difficult to predict by replicating *Rahman et al*'s work [26]. This also applies to most of our test projects indicating that the task of predicting the next function call with the full name is particularly difficult for these projects.

# B. Function-Call Completion with PV Model (RQ2)

We compare the performance of the ranking of Eclipse's static analysis candidates with the paragraph vector model and the unranked Eclipse's static analysis. Figure 4 summarizes the scores in term of Recall@10 and MRR for both systems on the 10 bold test projects of Table II. For 9 out of the 10 projects under test, the results highlight a common trend, that is big improvements of Eclipse static analysis when ranking it with the paragraph vector model. The only exception is *game-of-life* which can be explained by the small size of the project and the low percentage of vocabulary coverage (64%).

Two projects (twitter4j and clojure), each with more than 13.000 completion sites, stand out from the others with a MRR above 0.5 indicating that the relevant suggestion is on average between the first and the second position in the list. This can be explained by the high vocabulary coverage in these projects (respectively 99% and 94%). However, in Figure 3, clojure has also a very high cross-entropy meaning that the sequences of functions in the project are difficult to predict. Despite this, our model is able to find useful similar function sequences to perform accurate completions. Therefore, as we suspected in Section V-A, projects with the highest vocabulary coverage have the highest Recall@10 and MRR.

Finally, another aspect that we evaluated is the time to produce completion suggestions for a call site. This time is on average between 700 ms and 800 ms, which makes our approach usable in a real programming setting.

To answer RQ2, based on the large number of tested call sites, we can state that using the paragraph vector model to rank potential call candidates, obtained by static analysis, improved dramatically the correctness of the static analysis tool without a negative impact on the response time.

# C. Comparison of PV Model and n-gram Language Models for Function-Call Completion (RQ3)

We compare the performance of our model with state-of-theart n-gram language models. Figure 5 summarizes the scores

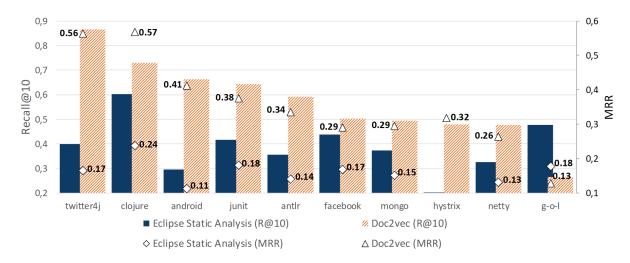


Fig. 4. Comparison of Recall@10 and MRR on 10 test projects with Eclipse Static Analysis and Doc2vec.

and the Table V shows the overall results for comparison with RQ2.

As we can observe, the *n*-gram model is the worst performing model. It improves Eclipse's static analysis for only 4 test projects and the performance is far from being of the same order of magnitude as the PV model. Even for projects that have low-level of cross-entropy (*twitter4j and junit, see Figure 3*), the *n*-gram model is not able to provide accurate recommendations. This is particularly reflected in the low MRRs for these projects, which indicates that the correct recommendation does not appear at the top of the list, on average.

Given the poor performances of the n-gram model, we investigate the use of a n-gram model augmented with a cache component. The cache allows to improve considerably the performance of the n-gram model. Nevertheless, although the model stores cache information about the project under test during the evaluation, the n-gram model is less efficient than the PV model for a great majority of the test projects, as depicted in Figure 5. In addition to that, the n-gram model still produces much lower MRRs than the PV model in all the cases but one. For a fairer evaluation, our model should also have included a cache component. Nonetheless, even though we did not implement that mechanism for the PV model, it largely outperforms a n-gram cache model that has shown to outperform some LSTM-based deep neural network in Hellendoorn and Devanbu's previous work on source code modeling [7].

In term of memory footprint and time for completion, the n-gram model has a lower computational cost and is faster than the paragraph vector to produce a suggestion. Therefore, despite the promising results, there is room for improvement to produce suggestions faster for the PV model.

To answer RQ3, we can state that the paragraph vector model performs much more effectively than n-

gram models for the function-call completion task. The PV model is better at abstracting the recurring function sequences in the training set in order to provide insightful recommendations. Therefore, we believe that the usage of models, such as a PV model, that are able to learn more complex relationships between tokens in a sequence (e.g., non-sequential relationships) should be favoured for such a task.

# D. Threats to Validity

We identified some threats to the validity of our evaluation and attempted to address them during its design. The first threat relates to the mono-operation bias as we experimented only with Java projects. We conjecture that our approach can be used for call completion in other languages as we do not rely on Java language constructs, but on identifiers. To prevent the mono-method bias, we evaluated our approach with two metrics commonly used to measure the effectiveness of ranking systems, R@k and MRR. Another threat concerns the interaction of setting and treatment. Indeed, we reused and compared our results with the completion in Eclipse. It has been shown that Eclipse's static analysis tools are commonly used by Java developers [2], and we do believe that it is representative enough. Similarly, we compared our paragraph vector model with only one other type of model, i.e., ngram model. Nevertheless, we considered two variants of this model and believe that it provides a good basis for comparison between two kind of models that are completely different in nature and widely used in the literature. Another important aspect that we considered is the representativeness of the dataset. We made sure to train our models on a large dataset of open-source projects from different domains of application and of variable sizes. For the evaluation, we choose a variety of test projects as well. Finally, an important threat to the validity of our results arises from the choice of hyper-parameters of the

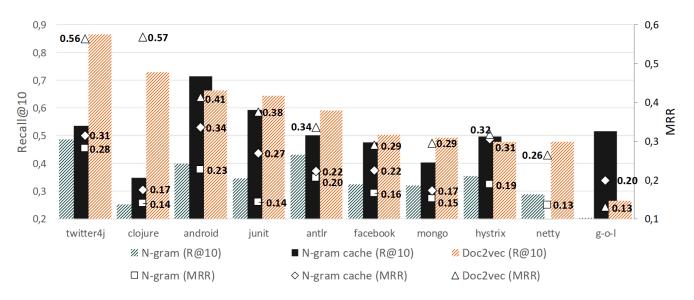


Fig. 5. Comparison of Recall@10 and MRR on 10 test projects with n-gram, n-gram cache and Doc2vec models.

 $TABLE\ V$  Global results of RQ2 and RQ3 (bold: best, underlined: second best, R@10 in percentage).

PROJECT		Eclipse		Doc2vec		n-gram		n-gram cache	
	Size	R@10	MRR	R@10	MRR	R@10	MRR	R@10	MRR
game-of-life	128	<u>47.66</u>	0.18	26.59	0.13	20.31	0.06	51.56	0.20
ındroid-async-http	675	29.66	0.11	<u>66.37</u>	0.41	39.85	0.23	71.56	0.34
facebook-android-sdk	5689	43.79	0.17	50.40	0.29	32.41	0.16	<u>47.65</u>	0.22
nystrix	5790	20.38	0.08	<u>47.89</u>	0.32	35.37	0.19	49.67	0.31
unit	8144	41.69	0.18	64.44	0.38	34.55	0.14	<u>59.23</u>	0.27
ntlr	11053	35.61	0.14	59.16	0.34	43.04	0.21	<u>50.17</u>	0.22
clojure	13020	60.28	0.24	72.90	0.57	25.16	0.14	34.88	0.17
witter4j	13365	39.96	0.17	86.65	0.56	48.71	0.28	<u>53.47</u>	0.31
nongo-java-driver	35836	37.34	0.15	49.37	0.30	32.00	0.15	<u>40.31</u>	<u>0.17</u>
netty	72754	32.58	0.13	47.82	0.26	28.95	0.13	-	-

paragraph vector models. To address the issue, we followed guidelines from the literature. We tuned the hyper-parameters that influence the most the quality of embeddings<sup>5</sup> and chose commonly used values for the other hyper-parameters, following Lau and Baldwin's recommendations [20]. We reported the chosen values in Section **IV-B**.

# VI. RELATED WORK

Neural approaches, n-gram and embedding-based language models have been widely used for automating tasks of the software development lifecycle. However, we focus on code completion by contrasting previous works with our approach. Then, we discuss about source code modeling and the broader usage of embedding-based approaches on source code (see the

literature study by Chen and Monperrus [28] and the survey by Allamnis et al. [29] for more references on these topics).

Code Completion: Code completion has been an active field of research in software engineering. In one of the ealier learning-based approaches, Bruch et al. [3] used k-nearest-neighbors to find relevant code suggestions using features extracted from the call site. Later, Proksch et al. [4] improved their work by using Bayesian networks and gathering more context information. The main limitation of these techniques is that they are designed to predict calls of particular APIs and require training data specific to these APIs. In our embedding-based approach, the model learns distributed representations of the source code from a large training corpus and does not require to extract manual information from the project under development. With the hypothesis of naturalness of software, Hindle et al. [5] outlined the possibility to use n-

<sup>&</sup>lt;sup>5</sup>https://code.google.com/archive/p/word2vec

gram language models for code completion by predicting a call given the previous code tokens. Tu et al. [6] used cache n-gram language models for code completion by capturing local patterns in the code. Hellendoorn and Devanbu [7] extended this approach by improving the cache component with information about the scope of the call site. Nguyen et al. [30] proposed an extension of n-gram language models by incorporating semantic information about the completion context. Similarly, Nguyen et al. [8] used AST-based language models to learn higher-level patterns than n-gram language models to improve API code suggestion. Raychev et al. [31] compared the performance of n-gram and neural language models for Android API code suggestion. In this work, we compared our approach with two configurations of n-gram models and show that a paragraph vector model is able to outperform both n-gram models on function-call completion. Moreover, the chosen n-gram baselines are strong since they have shown to be efficient and sometimes a better choice than RNN/LSTM-based neural networks for source code modeling [7]. We also show that such models can be integrated with a type-based completion tool to suggest only tangible function calls.

Recent approaches using deep learning have mainly focused on learning representations of ASTs with attentionbased neural networks. Bhoopchand et al. [9] used pointer networks to learn long-range dependencies in Python ASTs for identifiers completion. Li et al. [10] used the same approach with a focus on out-of-vocabulary identifiers. Liu et al. [24] leveraged LSTM neural networks trained on partial ASTs to predict nodes in a target AST. Karampatsis et al. [23] proposed a LSTM neural networks that is able to suggest out-of-vocabulary identifiers by learning the internal structure of code tokens. In the same vein, Svyatkovskiy et al. [11] compared several neural network architectures for method and API recommendations in Python. They learn AST-based representations of code snippets to perform the completion by comparing a call site context with the representations learned by their model. In a subsequent paper, Svyatkovskiy et al. [12] defined a framework using the same approach combined with an existing code completion tool to produce ranked lists of suggestions. Alon et al. [32] proposed an approach where a transformer model learns to predict an AST node given all possible AST paths leading to this node. Finally, Kim et al. [25] designed the same kind of approach but compared several ways to feed AST trees into a transformer model and focused the evaluation of their model on predicting specific types of tokens. These deep learning based works have shown to be efficient for code completion and especially to predict common API calls. In future works, we plan to compare our approach with deep neural network approaches for the completion of function-calls in general and not specifically for particular

**Source Code Modeling:** Recent works on source code modeling have focused on learning probabilistic models of code. Source code modeling is usually an upstream task of predictive tasks such as code completion. Approaches based

on n-gram language models have shown to be useful to find regularities in code [5]–[7], [30]. More recent approaches are based on distributed representations of source code [15]–[17], [33] that learn more complex semantic relationships between code tokens. Both kinds of approaches can be useful for some downstream tasks. Allamanis et al. [22] used embedding-based language model to predict method names. Nguyen et al. [34] learned embeddings of API elements and try to map them across programming languages. Gu et al. [35] proposed an embedding-based approach to find relevant API sequences given a search query. From another paerspective, White et al. [36] and Chen and Monperrus [37] used embeddings to find similarities in code for automatic program repair. Finally, Büch and Andrzeiak [38] learned embedding of ASTs of methods for clone detection. These previous works show a broad range of applications in which our embedding-based approach could be used with small adaptations. These include method names prediction [22], [39], clone detection [40], [41], API pattern detection [42] or code search [43]–[46].

# VII. CONCLUSION

In this paper, we presented an approach for function-call completion that can be integrated with a static analysis tool based on a language typing system. Our approach starts from the assumption that it is possible to abstract application-independent high-level concepts in the form patterns of call sequences contained in code repositories. To this end, we build on document-embedding algorithms to train models that can be exploited for function-call completion. Our experiments highlights promising results for most of the tested projects and indicate that our trained model captures useful high-level concepts that can be used for completion. This shows that our approach can be useful for helping developers writing their software even for new projects and with limited knowledge about the used APIs.

Although the obtained results are satisfactory, there is room for improvement. One of the limitations of our approach is that it is less efficient with projects having very specific function names, not frequent in existing code repositories. We plan to improve the natural-language processing pipeline to cope with this situation. We also plan to explore other embedding-based language models to improve the completion. Finally, instead of capturing high-level concepts inside a method scope, we plan to learn similar concepts in wider scopes and thus learning recurring long-range dependencies that could be useful for program summarizing, for instance.

From another perspective, the fact that our approach does not rely on language constructs, but rather on sequences of identifiers used in method names opens the door for many other possibilities to explore. Indeed, we conjecture that the learned models can be reused cross-programming languages. They can also be used, with some adaptation, to assist developers for other tasks such as program documentation by providing summaries, construct naming for automated generation, clone detection, and code search. Finally, an approach

similar to ours can be employed to assist in building design diagrams such those of UML.

#### REFERENCES

- [1] G. Murphy, "Beyond integrated development environments: Adding context to software development," 05 2019, pp. 73–76.
- [2] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, pp. 76–83, 07 2006.
- [3] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 213–222. [Online]. Available: https://doi.org/10.1145/1595696.1595728
- [4] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, Dec. 2015. [Online]. Available: https://doi.org/10.1145/2744200
- [5] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, p. 837–847.
- [6] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–280. [Online]. Available: https://doi.org/10.1145/2635868.2635875
- [7] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 763–773. [Online]. Available: https://doi.org/10.1145/3106237.3106290
- [8] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference* on Software Engineering - Volume 1, ser. ICSE '15. IEEE Press, 2015, p. 858–868.
- [9] A. Bhoopchand, T. Rocktäschel, E. T. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," *CoRR*, vol. abs/1611.08307, 2016. [Online]. Available: http://arxiv.org/abs/1611. 08307
- [10] J. Li, Y. Wang, I. King, and M. R. Lyu, "Code completion with neural attention and pointer networks," *CoRR*, vol. abs/1711.09573, 2017. [Online]. Available: http://arxiv.org/abs/1711.09573
- [11] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2019. [Online]. Available: http://dx.doi.org/10.1145/3292500.3330699
- [12] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," 2020.
- [13] R. Kneser and H. Ney, "Improved backing-off for m-gram language modeling," 1995 International Conference on Acoustics, Speech, and Signal Processing, vol. 1, pp. 181–184 vol.1, 1995.
- [14] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in 34th Annual Meeting of the Association for Computational Linguistics. Santa Cruz, California, USA: Association for Computational Linguistics, Jun. 1996, pp. 310– 318. [Online]. Available: https://www.aclweb.org/anthology/P96-1041
- [15] 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://dblp.uni-trier.de/db/journals/corr/corr1301. html#abs-1301-3781
- [16] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310. 4546
- [17] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014. [Online]. Available: http://arxiv.org/abs/1405.4053
- [18] A. M. Dai, C. Olah, and Q. V. Le, "Document embedding with paragraph vectors," *CoRR*, vol. abs/1507.07998, 2015. [Online]. Available: http://arxiv.org/abs/1507.07998

- [19] K. Hashimoto, G. Kontonatsios, M. Miwa, and S. Ananiadou, "Topic detection using paragraph vectors to support active learning in systematic reviews," *J. of Biomedical Informatics*, vol. 62, no. C, p. 59–65, Aug. 2016. [Online]. Available: https://doi.org/10.1016/j.jbi.2016.06.001
- [20] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," *CoRR*, vol. abs/1607.05368, 2016. [Online]. Available: http://arxiv.org/abs/1607. 05368
- [21] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602. 03001
- [22] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2786805.2786849
- [23] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code! = big vocabulary: Open-vocabulary models for source code," 2020.
- [24] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2017.
- [25] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," 2020.
- [26] M. Rahman, D. Palani, and P. Rigby, "Natural software revisited," 05 2019.
- [27] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 207–216.
- [28] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," *CoRR*, vol. abs/1904.03061, 2019. [Online]. Available: http://arxiv.org/abs/1904.03061
- [29] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017. [Online]. Available: http://arxiv.org/abs/1709. 06182
- [30] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 532–542. [Online]. Available: https://doi.org/10.1145/2491411.2491458
- [31] 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*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 419–428. [Online]. Available: https://doi.org/10. 1145/2594291.2594321
- [32] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," 2019.
- [33] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart, *Distributed Representations*. Cambridge, MA, USA: MIT Press, 1986, p. 77–109.
- [34] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen, "Mapping api elements for code migration with vector representations," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 756–758. [Online]. Available: https://doi.org/10.1145/2889160.2892661
- [35] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," CoRR, vol. abs/1605.08535, 2016. [Online]. Available: http://arxiv.org/abs/1605.08535
- [36] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," *CoRR*, vol. abs/1707.04742, 2017. [Online]. Available: http://arxiv.org/abs/1707.04742
- [37] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *CoRR*, vol. abs/1811.05703, 2018. [Online]. Available: http://arxiv.org/abs/1811.05703
- [38] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, pp. 95–104.
- [39] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018. [Online]. Available: http://arxiv.org/abs/1808.01400

- [40] 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. [Online]. Available: http://arxiv.org/abs/ 1802.07779
- [41] L. Buch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," 02 2019, pp. 95–104.
- [42] T. Nguyen, A. Nguyen, and T. Nguyen, "Mapping api elements for code migration with vector representations," 05 2016, pp. 756–758.
- [43] S. Luan, D. Yang, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *CoRR*, vol. abs/1812.01158, 2018. [Online]. Available: http://arxiv.org/abs/1812.01158
- [44] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1050–1060.
- [45] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," *CoRR*, vol. abs/1905.03813, 2019. [Online]. Available: http://arxiv.org/abs/1905.03813
- [46] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.