Learning APIs through Mining Code Snippet Examples

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
C M Khaled Saifullah

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Developers extensively use and reuse the Application Programming Interfaces (APIs) to faster the development time and effort. In order to do this, developers need to learn and remember APIs for effectively using them in their codebase. However, APIs are difficult to learn as they are large in numbers and are not properly documented and the documentation contains a lot of text to remember. To support developers learning and using those APIs, this thesis focuses three different studies that (1) enhances the code completion features of the modern integrated development environments (IDEs), (2) make the online forum code snippets compilable and (3) annotates the code elements of the dynamically typed programming language (e.g, JavaScript) by their types.

Towards this direction, we first explore the method name, argument and code completion techniques in the literature and find that none of them is suitable for completing a full method call sequence which consists of a name and a list of arguments. Thus we propose a Bi-LSTM based encoder-decoder model with attention mechanism and beam search, DAMCA that takes all three lexical, syntactic and semantic contexts of a method call and returns a list of method call sequences as the completion suggestions. Evaluation results show that the proposed technique outperforms the state-of-the-art method name, argument, code completion and program synthesis techniques for method call sequence completion. Next, we explore the techniques that are proposed for resolving the Fully Qualified Name (FQN) of the API element of the online forums code snippets. We find that the techniques restrict themselves by the locally specific code tokens only. We incorporate globally related tokens with the local tokens and use likelihood, context similarity, and name similarity to resolve the API element. Experimental results show that the proposed technique outperforms the state-of-the-art techniques with faster training. Finally, in our third study, we explore the techniques developed for statically typed programming languages (i.e, Java) for dynamically typed programming languages (i.e, JavaScript). The evaluation results show that the techniques performed very poorly for JavaScript. Next, we investigate the causes and built a technique that leverages Word2Vec, context similarity as the global models and previous outputs on the same project as a local model. The combination of models outperforms the technique developed for Java. We then compare the proposed technique with state-of-the-art deep learning based techniques developed for JavaScript. The experimental results suggest that the proposed technique has faster training time than the deep learning based technique without sacrificing accuracy. We believe that findings from this research and proposed techniques have the potential to help developers learning different aspects of APIs, thus ease software development and improve the productivity of developers.

Acknowledgements

At first, I like to praise Almighty Allah, the most gracious and most merciful, who gave me the ability to carrying out this work. Next, I would like to express my sincerest appreciation to my supervisor Dr. Chanchal K. Roy for his continuous guidance, help, motivation, and remarkable endurance during this thesis work. Without his guidance, this work would have been unthinkable.

I would like to thank Dr. Zadia Codabux, Dr. Roy Lee, Dr. Shahedul Khan, and Dr. Mark Keil for their willingness to take part in the advisement and evaluation of my thesis work. I would also like to thank them for their valuable time, suggestions and insights.

I would also wish to express my gratitude to Dr. Muhammad Asaduzzaman for extended discussions, valuable suggestions, passionate participation, and input, which have contributed greatly to the improvement of the thesis.

I would like to express my special appreciation and thanks to the anonymous reviewers for their valuable comments and suggestions on improving the papers produced from this thesis.

Thanks to all of the members of the Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Dr. Manishankar Mondal, Dr. Saidur Rahman, Dr. Masudur Rahman, Dr. Jeffrey Svajlenko, Farouq Al. Omari, Shamima Yeasmin, Judith Islam, Muhammad Mainul Hossain, Kawser Wazed Nafi, Amit Kumar Mondal, Tonny Kar, Golam Mostaeen, Rayhan Ferdous, Debasish Chakroborti, Hamid Khodabandehloo, Saikat Mondal, Md Nadim, and Avijit Bhattacharjee.

I am thankful to the Department of Computer Science of the University of Saskatchewan for their generous financial assistance through scholarships, awards, and bursaries that helped me to focus more deeply on my thesis work.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me to reach this stage. In particular, I would like to thank Gwen Lancaster, Greg Oster, Jeff Long, Cary Bernath, Smit Choksi, Sophie Findlay, Shakiba Jalal, Heather Webb, and James Ko.

I would like to convey my love and gratitude to my beloved wife, Afsana Sultana, who has brought a new meaning to my life. She came to my life the time I needed her the most and stayed with me in ease and hardship, inspired me constantly, and took away all stresses that made my work so easy.

I express my heartiest gratitude to my late mother Amina Akhter Parveen, and my father Shamsur Rahman Chowdhury who are the architects of my life. Their endless sacrifice, unconditional love, and constant good wishes have made me reach this stage of my life. I would also like to thank my late mother-in-law Mrs. Syeda Sharmin Sultana and father-in-law Md. Gias Uddin Chowdhury for their constant well wishes and inspirations in this thesis work. My brothers- Fahim, and Nayeem, and in-laws- Shohag, and Rumpa have always inspired me in completing my thesis work, and I thank all of them.

I dedicate this thesis to my mother Amina Akhter Parveen and my father Shamsur Rahman Chowdhury whose inspirations help me to accomplish every step of my life.

Contents

List of Tables

List of Figures

[1] https://github.com/eclipse/eclipse.jdt

List of Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
Bi-LSTM	Bidirectional Long Short Term Memory
BLUE	Bilingual Evaluation Understudy
CBOW	Continuous Bag of Word
COSTER	Context Sensitive Type Solver
CPU	Central Processing Unit
CSCC	Context Sensitive Code Completion
DAMCA	Deep API Method Call cum Arguments recommendation system
DeepAPI	Deep API Learning
DeepTyper	Deep Learning Type Inference
DNN4C	Deep Neural Network based Language Model for Source Code
Eclipse JDT	Eclipse Java Development Tool
EP	Extremly Popular APIs
FQN	Fully Qualified Name
GPU	Graphical Processing Unit
IDE	Integrated Development Environment
JDK	Java Development Kit
js	JavaScript
JSDoc	JavaScript Documentation
LSTM	Long Short term Memory
MRR	Mean Reciprocal Ranking
NL2Type	Natural Language 2 Type Inference
OLD	Occurrence Likelihood Dictionary
P	Popular APIs
PARC	Parameter Recommender
PPA	Partial Program Analysis
RAM	Random Access Memory
RNN	Recurrent Neural Network
RNNLM	Recurrent Neural Network based Language Model
SLAMC	Statistical Language Model for Source Code
SLP	Software Language Processing library
SMT	Statistical Machine Translation
StatType	Statistical Machine Translation Based Type Solver
ts	TypeScript
tsc	TypeScript Compiler
UP	Unpopular APIs
VRAM	Video Random Access Memory
VP	Very Popular APIs
VU	Very Unpopular APIs

1 Introduction

1.1 Motivation

Software Development in the real world is hugely dependent on the Application Programming Interface (API) usages. Developers use and reuse different APIs in their code to fasten the development as well as lessen the effort of writing those code from scratch [10, 20, 113, 144]. For example, when users want to integrate map or map related activities in their software, Google Map API[1] gives the support to all map and location related features. These APIs are more like a plug and play tool in the source code. Developers include the libraries/APIs with their project using either Maven[2] or Gradle[3] in java, Pip[4] or Conda[5] in Python, npm[6] in JavaScript and so on, and call their methods that get the ready-made solutions of particular functionalities.

However, such libraries/APIs need to be learned to get the best out of them. For example, a user is building a ride sharing application, where (s)he needs to collect the route information between the source and destination from each user's request. One of the ways to get the route information is shown at Listing 1.1.

```
1 GeoApiContext context = new GeoApiContext.Builder()
      .apiKey("AIza...").build();
2 DirectionsResult directions = DirectionsApi.getDirections(context, source
      , destination).await();
3 DirectionsRoute[] routesArr = directions.routes;
4 System.out.println(routesArr[0].summary);
```

Listing 1.1: An example[7] of Google MAP Direction API usages

In the code example at Listing 1.1, there are a number of API elements from Google MAP Direction API such as *GeoApiContext*, *DirectionsResult*, and so on and method calls such as *apiKey("AIza...")*, *getDirections(context, source, destination)* and so on. To understand how they work, the developer needs to learn the API elements along with the methods invoked on these elements. Developers, in general, use one of the three following approaches to learn the APIs.

[1] https://developers.google.com/maps/documentation
[2] https://maven.apache.org/
[3] https://gradle.org/
[4] https://pypi.org/
[5] https://docs.conda.io/
[6] https://www.npmjs.com/

First and the most trivial approach is to read the documentation. However, such a learning mechanism is impractical due to the large volume of APIs. Moreover, the documentation contains a lot of text and it takes a lot of time to understand the APIs. Furthurmore, Raemaekers et al. [99] found that 47% of libraries in the Maven central repository do not contain documentation. In summary, half of the APIs/libraries do not have any documentation and the other half contains long text as documentation which is time-consuming and impractical for learning.

The second approach to learn APIs is code completion tools. Almost all modern integrated development environments (IDEs) such as Eclipse[8], NetBeans[9], IntellIJ[10], Microsoft Visual Studio[11], Pycharm[12], WebStorm[13] and so on provide the code completion feature. The feature shows a list of completion suggestions in a popup window so that the developer can navigate and select the code element (s)he is looking for. For example, whenever the developer writes dot(.) after a receiver variable, Eclipse IDE provides a list of methods alphabetically defined for the receiver variable as method name completion suggestions. After the developer selects the desired suggestion, Eclipse IDE completes the method name. A study conducted by Omari et al. [91] found that code completion actions are repetitive in nature and Murphy et al. [81] found the code completion is one of the top ten commands used by the developers. Therefore according to the prior studies, the code completion feature is a very important medium for developers to learn the APIs. Thus, we are interested to leverage the code completion feature in IDEs to assist the developers to learn and use APIs.

Finally, developers use and explore different online Question-Answer sites such as Stack Overflow[14], Github Gists[15] and so on to get the code examples of API usage. A study conducted by Singer [118] shows that the developers prefer code examples than the documentation to learn the APIs. Whenever a developer wants to learn the API through the online forums, (s)he explores the API using google search, traverses the related discussions or code examples in the online forum and posts a question if no suitable discussion or code example is found. One important drawback of the code examples at the online forums is not being executable. Horton and Parnin [141] found that only 1% of the Java and C# code examples in the Stack Overflow posts are compilable. Yang et al. [49] further report that less than 25% of Python code snippets in GitHub Gist are runnable. One of the solutions in the literature to use and reuse the online forums code snippets is resolving the type of API elements present at code snippets. Therefore, we are interested to explore different aspects of type inference/resolving techniques.

In any data-driven technique such as machine learning and deep learning, the performance of the technique is highly dependent on how we represent the data. As the naturalness and localness of the source code is well explored in the literature [45, 128], several studies [9, 10, 42, 43, 84, 89, 96, 120] try to solve different software

[8] https://www.eclipse.org/ide/
[9] https://netbeans.org/
[10] https://www.jetbrains.com/idea/
[11] https://visualstudio.microsoft.com/vs/
[12] https://www.jetbrains.com/pycharm/
[13] https://www.jetbrains.com/webstorm/
[14] https://stackoverflow.com/
[15] https://gist.github.com/

engineering problems using the locally specified code tokens as the context. However, Tufano et al. [129] found that different representations of source code as context provide better performance for source code similarity. Nguyen et al. [86] also found the lexical, syntactic and semantic representations of code as context outperforms techniques that use only local code tokens as the context for code completion task. Therefore, in our studies, we would like to explore different types of contexts while building techniques that will assist the developers to learn APIs more effectively.

1.2 Research Problem

Documentation, code completion feature of IDEs and online forums remain the most used means to learn APIs nowadays. However, such methodologies also raise the following research problems.

- First, the IDEs mostly use the type of the receiver variable as the only information while suggesting the method name. However, in literature, source code snippets are found repetitive [45] and locally specific [10, 128]. A number of studies [9, 10, 43, 84–86, 89, 106] utilizes the repetitiveness and localness properties of the source code to improve the performance of the code completion specifically the method call completion. However, a method call completion consists of two items: method name completion and argument completion. The method name completion techniques leave the task of completing the arguments to the developers. On the other hand, argument completion techniques leave task of completing the method names to the developers. Finally, code completion techniques that suggest one code token at a time treat method name and argument as same as the other code elements. However, method name and argument carry a lot of type information that require more attention. Thus, we find a lack of study of completing both method names and arguments as a sequence with different levels of context information.

- Second, the code examples at online forums are not executable. Such characteristic hinders the learning of APIs. A number of studies [28, 96, 108, 120] attemted to resolve the type of the API element using either related documentations/discussions or locally specific code tokens. However, the techniques fail to meet the requirement of the developer due to the unavailability of documentation, informal nature of the discussion, being too strict to scope rule and being biased to APIs with higher examples. Furthermore, we find a lack of study that explores the relation of different types of contexts with the task of type resolving of API element of online forum code snippets.

- Third, the techniques described the previous problems are designed for the statically typed programming language, Java. However, almost all these techniques claim that they are adaptable for the dynamically typed programming language such as JavaScript. Unfortunately, there is a lack of study that validates such a claim. Moreover, only a few studies [42, 69, 105] are found in the literature that infer the types of the code elements of the dynamically typed programming language, JavaScript. However, state-of-the-

art techniques are being deep learning based, we find a lack of study whether a much simpler technique can be applied for such a problem.

1.3 Discovering Problems and their Solutions

Method call consists of two pieces: a name and argument(s). Modern IDEs complete each piece one by one where they consider the type of the receiver variable as the context for method name completion, and formal signature of the method and the related identifiers as the context for the argument completion. However, the code completion feature being very useful, IDEs fail to meet the requirement of the developers due to considering a very small amount of context and thus resulting in limited support. Method name completion techniques such as BMN [20], CSCC [10], APIRec [84] and so on are proposed in the literature that utilizes the localness property of the source code to suggest the method name to complete. However, they leave the argument completion task for the developers. On the other hand, a few argument completion techniques such as Precise [144], PARC [9], and LexSim [66] are proposed too. Similar to earlier techniques, argument completion techniques expect that developers completed the method name by themselves. Furthermore, the code completion techniques such as SLP [43], SLAMC [89], DNN4C [86] treat all code tokens including method name and argument equally. However, method name and argument contain a lot more syntactic and semantic information than the other code tokens such as *for, if, break* and so on. Therefore, we are interested to explore the method call completion as the sequence to sequence learning tasks where a list of method names and the arguments will be provided to the developers as the completion suggestion. Moreover, we are interested to consider the lexical, syntactic and semantic context for the method call completion.

While learning APIs from the social online forums, the developers find the code snippets are not ready-made solutions. These code snippets are incomplete, have no declaration statement (declaration ambiguity), have no import statement (external reference ambiguity), and API elements have more than one fully qualified name candidates (name ambiguity). One solution to make the online forums code compilable is resolving the type of the API element. Studies in the literature show that the type inference for the API elements is done in two ways. First, API elements are searched in the knowledge base where the documentations of APIs are stored [28] or API elements are searched within the discussion of the online forum posts [108]. However, documentations are rarely available [99] and discussions at online forums are very informal [120]. Second, the type of the API elements is resolved based on the localness property of the code. However, such techniques bind themselves within the scope where the target API element exits and fail to resolve the type properly [96]. Moreover, complicated approach such as statistical machine translation is used which is computationally expensive and not well for the APIs with the small number of examples in the training dataset. We are more interested to explore different types of contexts to find the fully qualified name of the API elements.

Finally, the techniques proposed for resolving the types of the API elements for statically typed programming language, Java claim that the techniques are adaptable for dynamically typed programming language. However, there is a marked lack of empirical studies to support the claim. Moreover, type inference is also important for dynamically typed programming languages. Studies [33, 41, 69, 104] show that type system in such languages helps to avoid bugs, understands undocumented code, makes code completion more effective and fixes type issues and semantic errors. Thus a number of techniques [42, 69, 105] have been proposed to infer the code element for dynamically typed programming language, JavaScript. However, the most recent two techniques [42, 69] are deep learning based where the study regarding the necessity of deep learning for such a problem is missing. We are interested to investigate the necessity by building more simpler technique without sacrificing the accuracy.

1.4 Addressing Research Problems

The previous section briefly describes the systematic investigation that helps to discover the research problems on learning APIs and provides hints of their solutions. The following three studies have been conducted in total to address the above-mentioned problems.

- **Study 1:** Deep API Method Argument Completion
- **Study 2:** Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets
- **Study 3:** Exploring Type Inference Techniques of Dynamically Typed Languages

The following subsections briefly describe each of these studies.

1.4.1 Study 1: Deep API Method Argument Completion

The study defines the method call completion problem as a sequence to sequence learning task. The lexical, syntactic and semantic contexts of a method call are used as the sequence of input to get a list of method names along with the arguments as sequences of output. By doing so, the study proposes a neural encoder decoder architecture based sequence to sequence technique, **DAMCA**, that uses attention mechanism based Bi-Directional Long Short Term Memory (LSTM) with beam search to recommend a list of method names with arguments as the completion suggestions. We evaluated the proposed technique with six different state-of-art method name, argument, and code completion, and program synthesis techniques where we use 10 medium to large subject systems and three different libraries as the dataset. The experimental results reveal that the proposed technique outperforms all the compared techniques by 5-25% in accuracy and 10-30% in Mean Reciprocal Ranking (MRR) for both intra and cross-project settings.

1.4.2 Study 2: Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets

This study resolves the type of API elements by finding the Fully Qualified Name (FQN) based on the context. While doing that, the study proposes a novel context-sensitive technique, called **COSTER**. The proposed technique collects source code elements within the top and bottom four lines as the local context as well as methods invoked on the receiver variable and methods use the receiver variable or the API element as the parameters as the global context. In training, the technique calculates the likelihood score and builds an occurrence likelihood dictionary (OLD). Given an API element as a query, COSTER captures the local and global contexts of the query API element, matches that with the FQNs of API elements stored in the OLD, and rank those matched FQNs leveraging three different scores: likelihood, context similarity, and name similarity scores. Evaluation with more than 600K code examples collected from GitHub and two different Stack Overflow datasets shows that our proposed technique improves precision by 4-6% and recall by 3-22% compared to state-of-the-art techniques. The proposed technique reduces the training time by a factor of ten in comparison with the existing state-of-the-art technique, StatType. Extensive analyses on results demonstrate the robustness of the proposed technique.

1.4.3 Study 3: Exploring Type Inference Techniques of Dynamically Typed Languages

From our previous study, we learn that the type inference techniques developed for statically typed programming languages such as Java can be adapted in the dynamically typed programming languages such as JavaScript. However, we are interested to investigate the above issue. Our investigation on a GitHub dataset of 25 million code tokens shows that the state-of-the-art type inference techniques for Java loose accuracy of more than 50% for JavaScript. While analyzing the causes of the poor performance of these techniques, we propose a technique that collects only the local context for each code element, calculates the semantic relatedness between contexts and the code tokens using Word2Vec and stores the contexts in a Lucene index file. While inferring types of a query code element, the technique extracts context, generates candidate types based on the semantic relatedness score from trained Word2Vec model, and sorts the candidates based on the context similarity and local model scores. The combination of local and global models provides 20-47% more accuracy than the statically typed language based techniques. Finally, our evaluation results of the proposed technique with state-of-the-art deep learning techniques developed for JavaScript reveals that our technique is 5-14 times faster than the compared techniques without sacrificing accuracy. Our analyses on sensitivity, overlapping of predicted types and the number of training examples justify the importance of the proposed technique.

1.5 Publications

Below is the list of publications and the works that are prepared for submission (with collaborator) from this thesis.

- **C M Khaled Saifullah**, Muhammad Asaduzzaman and Chanchal K. Roy. Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, pp. 243-254, 2019 (Acceptance rate: **77/377=20.40%**).

- **C M Khaled Saifullah**, Muhammad Asaduzzaman and Chanchal K. Roy. Exploring Type Inference Techniques of Dynamically Typed Languages. In Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, 2020 (Acceptance rate: **42/199=21.10%**).

- **C M Khaled Saifullah**, Muhammad Asaduzzaman, Banani Roy, Chanchal K. Roy and Kevin A. Schneider. Deep API Method Argument Completion. In IEEE Transaction on Software Engineering (to be submitted).

- **C M Khaled Saifullah**, Muhammad Asaduzzaman and Chanchal K. Roy. COSTER: A Tool for Finding Fully Qualified Names of API Elements in Online Code Snippets. In Proceedings of the 42nd International Conference on Software Engineering, 2020 (Demonstration Track) (under review).

- Saikat Mondal, **C M Khaled Saifullah**, Avijit Bhattacharjee, Mohammad Masudur Rahman and Chanchal K. Roy. Can Unanswered Questions of Stack Overflow Q&A Site be Automatically Predicted During their Submission Time?. In Proceedings of the 17th International Conference on Mining Software Repositories, 2020 (Under Review).

- **C M Khaled Saifullah**, Jeffrey Svajlenko and Chanchal K. Roy. BigCloneWE: Evaluating Clone Detection Tools using BigCloneBench in the Web. In Proceedings of the 36th IEEE International Conference on Software Maintenance and Evaluation, 2020 (Tool demos Track) (to be submitted).

- **C M Khaled Saifullah**, Muhammad Asaduzzaman, Banani Roy, Chanchal K. Roy and Kevin A. Schneider. Deep Method Argument Recommendation. In Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, 2020 (RENE Track) (to be submitted).

1.6 Thesis Outline

Chapter 2 discusses some background topics and related studies, we use in our studies. We present our Study 1 in Chapter 3. Chapter 4 describes our Study 2 and Chapter 5 discusses our Study 3. Finally, in Chapter 6, we conclude with an overall summary of the thesis and discussion of some future research directions.

2 Background and Related Studies

In this chapter, we provide a short discussion of the background, technical preliminaries, and related studies of the thesis. In Section 2.1 of the chapter, we discuss different language models applied in the software engineering tasks. We then present a general overview of string similarity functions used in the studies in Section 2.2 of the chapter. We discuss the background of the word embedding in Section 2.3. Finally, we present some related studies regarding the initiaves of learning APIs in Section 2.4.

2.1 Language Model

In natural language processing, language modeling defines as specifying a probability value to the sentences in a language [34]. The probability value signifies how likely the sentence would occur in the language. Besides specifying probability values to sequences of words (sentences), the model also specifies probability values to a given word or a sequence of words in a sentence to signify the likelihood of the word or the sequence of words to follow a sequence of words [34]. Source code being the natural product of human [45], different language models play a vital role to solve different software engineering problems. A few of them that are related to this thesis are described briefly.

2.1.1 Statistical Language Models

Statistical language models capture the regularities/patterns of the sentences/code statements by assigning them with the probability values based on previous occurrences [71]. The basic idea of this language model is, it dictates how likely the statement would occur in the language [39]. For instance, we have a code statement s in our language/project/dataset. Let us assume the code statement, s is $for(inti = 0; i < 10; i++)$;. Therefore the language model assigns the probability value for s using Eqn. 2.1.

$$Language\ Model(s) = Pr(s) = \frac{count(s)}{Total\ number\ of\ code\ statements\ in\ the\ language} \tag{2.1}$$

Thus, the language model learns how likely the code statement, s, appeared in the language/project/-dataset. However, such a modeling mechanism faces a challenge, data sparsity [71]. Data Sparsity is a phenomenon where enough data is not observed while modeling a language. [6]. Therefore, the model will get surprised frequently by seeing unknown data in testing [45]. One way to solve the data sparsity challenge is to model each word/code token of the statements based on the previous words. Therefore, for our code

statement, we first calculate the probability of each word and then use a joint probability function to model the full code statement. If we consider the previous one token while modeling each token, the model will be called unigram and it uses the following equations.

$$Language\ Model(s) = Pr(s) = Pr(for)Pr(()...Pr(;) \tag{2.2}$$

$$where,\ Pr(for) = \frac{count(for)}{Total\ number\ of\ code\ tokens\ in\ the\ document} \tag{2.3}$$

Therefore, the probability of each token depends on its probability and the value can vary on different documents. For example, in one document related to read and write files the probability of *for* can be 0.01 whereas the other document related to UI design, the probability can be *0.00001*. Thus, multiple documents based on different topics can be modeled using the unigram model and used to solve software engineering problems.

Whenever, the number of previous words while modeling each token becomes more than one, the model is regarded as n-gram, where n is the number of the previous word. The probability of each word depends on the previous n words in the n-gram model rather than its probability. Thus if the code statement, s has l number of tokens in it (for our example it is 15), then the n-gram model uses Eqn. 2.4 to calculate the probability for the tth code token of s.

$$Language\ Model(s) = Pr(s) \simeq \prod_{t=1}^{l} Pr(s_t|s_{t-(n-1)}, ..., s_{t-1}) \tag{2.4}$$

$$where,\ Pr(s_t|s_{t-(n-1)}, ..., s_{t-1}) = \frac{count(s_{t-(n-1)}, ..., s_{t-1}, s_t)}{count(s_{t-(n-1)}, ..., s_{t-1})} \tag{2.5}$$

In software engineering problems, the statistical language model such as the n-gram model suffers from limitations of data sparsity and curse of dimensionality [17, 45, 89].

2.1.2 Neural Language Model

The neural language model utilizes the continuous representation of word/code token which is also known as word embedding to model the code statement [17]. A neural language model can be derived as the model that uses an artificial neural network for calculating the likeliness of a token to appear in a code statement [135]. The neural network avoids the curse of dimensionality limitation of the statistical language model by representation each code token in a distributed way where the representation consists of a non-linear combination of weights in a neural network [16]. An example of a neural network that implements a neural language model is shown in Figure 2.1.

In general, the network for the neural language model is constructed and trained to predict a probability distribution over the tokens in the dictionary based on the context. The context can be a fixed size window of the previous word. Each token in the context is embedded into a word vector that can be represented

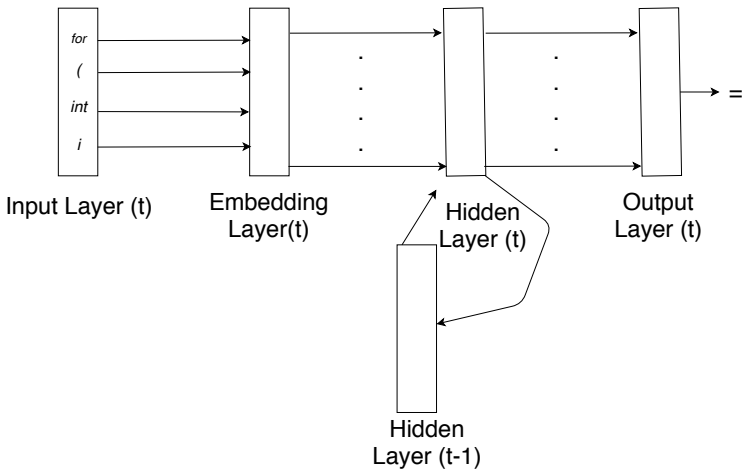


Figure 2.1: An example of neural network that implements a neural language model.

as a point in the higher dimensional vector space. It has the capability of capturing the semantic similarity between words [135]. For example, code token *int* and *double* being data types cluster together with lower distance whereas *equals* and *toString* will be away from *int* and *double* but remain close to each other.

A neural language model, in general, comprises four layers: input, embedding, hidden and output. The input layer contains the numerical vector of code tokens. Since the code tokens are of string type, we can represent them in numerical values by replacing the token by their index at the dictionary. Next the code tokens are embedded into the higher dimensional word vector ($s(\vec{t})$) using Equation 2.6. The embedded word vector, $s(\vec{t})$ and the values of the hidden layer from previous time step *h(t-1)* are then passed into the next hidden layer. Each node of the hidden layer is triggered by an activation function f that requires weights (w_{ij}) between the node of hidden and embedding layers. Based on the activation function values calculated by Equation 2.7, the output layer generates a probability distribution of all tokens of the dictionary using Equation 2.8.

$$s(\vec{t}) = embed(s(t)) \tag{2.6}$$

$$h_i(t) = f((\sum_{j=1}^{len(s(\vec{t}))} \vec{s_j}(t)w_{ij}), h(t-1)) \tag{2.7}$$

$$s_k(t) = g(\sum_{j=1}^{len(h(t)} h_j(t)v_{kj}) \tag{2.8}$$

Where, g represents the output activation function and v_{kj} represents the weight of edge between jth hidden node and $k$the output node. In hidden layers, any type of neural networks such as feedforward, Recurrent Neural Network (RNN), Long Short Term Memory (LSTM), Gated Recurrent Unit RNN/LSTM and Bi-direction RNN/LSTM can be used. Moreover, based on the problem definition, any number of the hidden layer can be stacked up in the model.

2.1.3 Neural Encoder-Decoder Model

A variant of the neural language model is the neural encoder-decoder model that deals with sequence input and output [22]. Such a model could be successfully implemented using the encoder-decoder architecture to solve machine translation [22], image captioning [15] and so on. The encoder-decoder model assumes that there are two languages: the source language, x and the translated language, s. Let us consider the example in Figure 2.2 where we are predicting the next code statement (target language) based on the context which is the current line of code (source language).

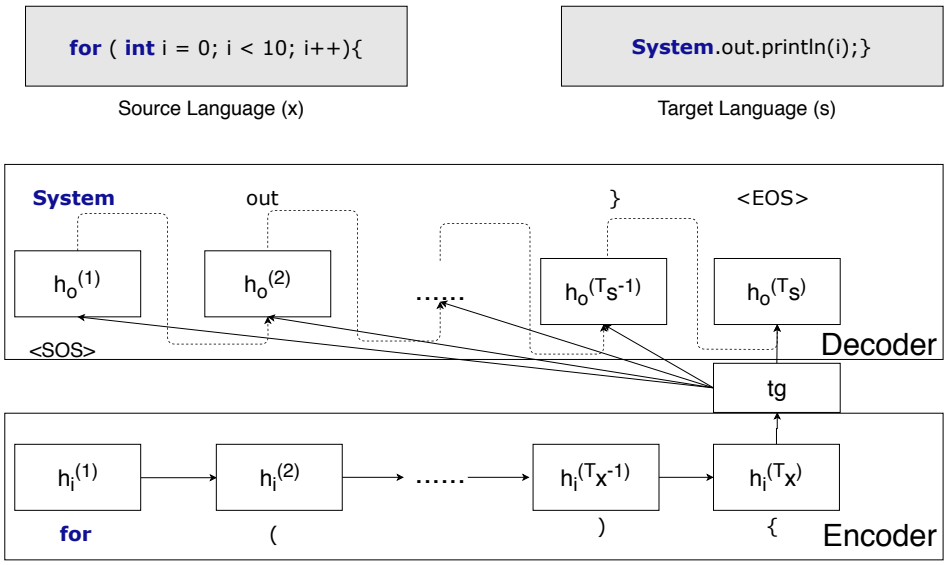


Figure 2.2: An example of a neural encoder decoder model where next code statement (s) is predicted based on the current line of code (s)

The source language, x is passed in the encoder phase of the model as of Figure 2.2. The encoder can be any type of neural network that computes the value of the hidden layers and summarizes in a fixed-length thought vector, tg using Equation 2.9. The thought vector, tg, saves the state of the hidden layer at last time step $h_i(T_x)$. The decoder can also be any type of neural network that accepts two inputs, one is the output of the previous time step and the other is the thought vector obtained from the encoder phase. Therefore, the output generated by the decoder follows the probability equation of n-gram(Eqn. 2.4) with the inclusion of the thought vector tg:

$$tg = h_i(T_x) = f((x(T_x), h_i(T_x - 1)) \tag{2.9}$$

$$Pr(s) = \prod_{t=1}^{T_s} Pr(s_t|s_1, ..., s_{t-1}, tg) \tag{2.10}$$

Recently, along with encoder-decoder architecture, attention mechanism is incorporated [39]. The attention mechanism has found to be useful in improving accuracy for the image captioning [139] and neural machine translation [13] problem.

Attention Mechanism

The basic idea for the attention mechanism is to assign attention weights with each hidden layer of the encoder while passing the context vector to every timestep in the decoder. The attention weights signify how much attention the decoder needs to put for any specific hidden layer of encoder while decoding into a word/token. For example, in Figure 2.3, the context vector tg_1 assists the decoder about the amount of attention it needs to give for each of the hidden layers of the encoder while decoding *System* token. In the case of the earlier architecture, the same context vector was repeated for every time step in the decoder. However, for long input sentence, remembering all tokens is not be useful [39].

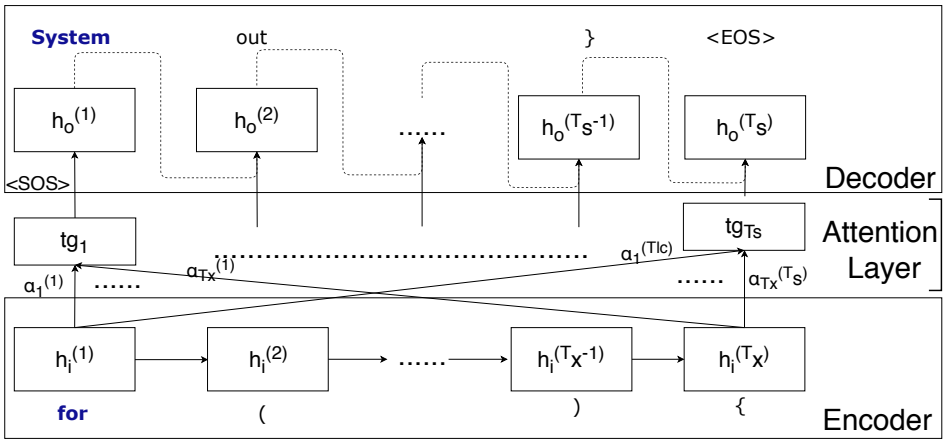


Figure 2.3: An example of a attention mechanism based neural encoder decoder model

To include the attention mechanism in the encoder-decoder model, the only change we need to do is putting attention weights while calculating the context vector. Therefore the equation 2.9 will be changed to:

$$tg_j = \sum_{t=1}^{T_x} \alpha_j(t) h_i(t) \tag{2.11}$$

That means for every jth time step of decoding, the context vector will be a summation of the product of hidden layers from the encoder at t time step and attention weights of t time step. The $\alpha_j(t)$ means how much attention $s(j)$ should pay to $h_i(t)$ while decoding. The attention weight can be calculated using equation 2.12.

$$\alpha_j(t) = \frac{exp(e_j(t))}{\sum_{i=1}^{T_x} exp(e_j(i))} \tag{2.12}$$

The term $e_j(t)$ is the attention factor for jth time step of the decoder and $t$the time step of the encoder. To calculate the value of attention factors, a neural network is used where the input is hidden layer values of the encoder for t timestep and hidden layer values of the decoder for $j-1$ timestep. One more addition in the neural encoder-decoder modeling is using beam search while calculating the probability of translated sentence.

Beam Search

While generating the output of the decoder, the token with the maximum probability value is considered. Eqn. 2.13 is used to generate the output in each timestep of the decoder.

$$s(j) = argmax(\prod_{i=1}^{dictionary\ size} Pr(s(i)|s(1), ..., s(i-1), tg_j)) \tag{2.13}$$

This is a greedy approach that can be stuck into the local optimum. To skip the local optima, researchers use beam search [57], a heuristic search algorithm in the decoder [39]. The principle of the beam search is to consider a $beam - width$ number of candidates rather than the best one after each token generation. An example of a beam search decoder with $beam - width$ value as 3 is shown in Figure 2.4.

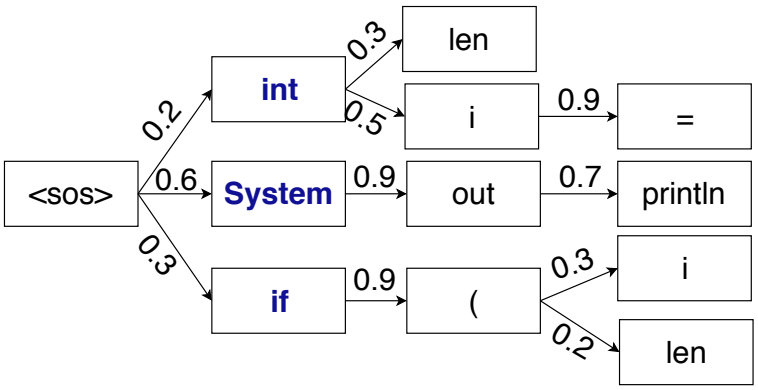


Figure 2.4: An example of Beam Search Decoder

After the first time step, the beam search decoder generates the top three tokens, int, $System$ and if with the probability values of 0.2, 0.6 and 0.3, respectively. Rather than continuing with the top one, the beam search decoder generates the top three tokens for each of the top three tokens of the first timestep in the second timestep. Thus we get nine tokens in the second time step and pick the top three tokens based on the probability value, multiply the values with the probability values from the first time step and continue until reaching the end of sequence ($< eos >$) token for all three decoded statements. Therefore the equation for calculating the probability of the statements described in equation 2.10 become:

$$Pr(s) = argmax(\prod_{t=1}^{T_s} Pr(s_t|s_1, ..., s_{t-1}, tg)) \tag{2.14}$$

Since the result is the multiplication of probability values, the result can be very small and many times become close to zero. To avoid such a problem, a length normalization function is used in the literature [137]. The normalization is calculated using Eqn. 2.15.

$$Pr(s) = \frac{1}{T_s^{\gamma}} \sum_{t=1}^{T_s} \log Pr(s_t|s_1, ..., s_{t-1}, tg)) \tag{2.15}$$

Here, γ represents the normalization factor. A Logarithm function is used because it is more stable, makes numerical rounding and maximizing log $Pr(s|x)$ maximizes $Pr(s|x)$ [137].

2.2 String Similarity functions

String similarity metrics or functions measure the distance between two strings based on approximation matching, fuzzy string searching and so on. Two of our conducted studies (Chapter 4, 5) make use of following string similarity functions to sort the candidate list.

2.2.1 Cosine Similarity

Cosine similarity[1] measures the orientation of two strings represented by two non-zero vectors. Each unique term in the strings is considered as a dimension and each string is then converted to a vector of such dimensions. Next, the function measures the cosine angle between the vectors [77]. Let us consider two string C_1 and C_2 representing two contexts for an API element. First, standard natural language preprocessing (e.g, token splitting, stop word and punctuation removal) are done to normalized the contexts and a vector $\vec{V}$ is built that contains all the unique terms from the normalized contexts. Next the cosine similarity *Cosine Similarity* between C_1 and C_2 is calculated using Eqn. 2.16.

$$Cosine\ Similarity(C_1, C_2) = \frac{\sum_{i=1}^{n} C_{1i} \times C_{2i}}{\sqrt{\sum_{i=1}^{n} C_{1i}^2 \times \sum_{i=1}^{n} C_{2i}^2}} \tag{2.16}$$

Here, C_{1i} and C_{2i} represents the weight of ith term in context C_1 and C_2, respectively, from $\vec{V}$. The function return values from zero that represents the complete dissimilarity to one that represents the complete similarity.

2.2.2 Levenshtein Distance

Vladimir Levenshtein [63] discovered this metric that calculates the minimum number of single-character edits (i.e, insertions, deletion, or substitution) between two words to obtain one from the other. For example, the Levenshtein distance between code token $BufferedReader$ and $InputStreamReader$ is eight which signifies we need eight edit operations to convert $BufferedReader$ token into $InputStreamReader$. Similar to the studies in the literature we use this metric in one of the conducted studies in this thesis (Chapter 4) where a smaller number of edit means the code tokens are more similar.

[1] `https://bit.ly/KTDKUy`

2.3 Word Embedding

Word embedding[2] is the combination of language modeling and feature learning techniques that map words or phrases from the dictionary into vectors of real numbers. In general, the word embedding techniques convert each word with a multidimensional space into a continuous vector having with lower dimensions. Studies [79, 102, 124, 143] in the literature utilize different word embedding techniques to extract the semantic relation between the word or code tokens. In one of our conducted studies (Chapter 5), we use Word2Vec [78], a well-known word embedding technique to get the type of code element based on the context.

2.3.1 Word2Vec

Mikolov et al. [78] propose Word2Vec, a shallow two-layered feed-forward neural network that takes a textual corpus and produces a high dimensional numeric vector for each unique word in the corpus. These vectors are called word vectors that are produced in such a way that if positioned in the vector space, words that share similar context become semantically similar words and remain close to each other in the space [111]. There are two types of architecture of Word2Vec: Continuous Bag of Word (CBOW) and Skip-gram. The neural network structure of CBOW and Skip-Gram with an example is shown in Figure 2.5.

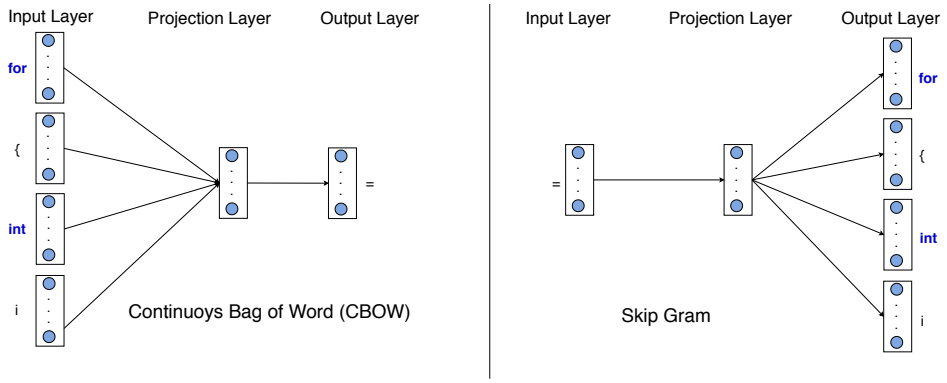


Figure 2.5: Neural network structure of CBOW and Skip-gram based Word2Vec

Continuous Bag of Word (CBOW) based Word2Vec embeds the words into the distributed representation given its contextual words. For example, in the example at Figure 2.5, the code token = is embedded based on the contextual words *for*, {, *int* and *i*. In the Skip-gram architecture, the model embeds the surrounding window of contextual words based on the current word. In our Study 3 (Chapter 5), we use the CBOW based Word2Vec to get the semantically similar types as the candidates of final output based on the context of the code element.

[2] `http://bit.ly/348vCcP`

2.4 Related Studies

A number of studies are found regarding the ways of learning APIs in the literature. Robillard [109] conducts a survey of 80 developers from Microsoft to find out the obstacle and ways to learn APIs. The survey results reveal that 78 percent of developers learn APIs through reading documentation, 55 percent use code examples, 34 percent use IDEs' code completion feature to learn APIs, 30 percent read articles, and 29 percent take peers' advice. On the follow-up survey by Robillard and Deline [110], the study finds the developers are having a difficult time understanding the APIs due to the lack of proper writing at the documentation. They also argue to have use case-based code examples at the documentation. Thus the developers are more comfortable with code examples rather than the documentation. Parmin and Treude [93, 95] find that social media content such as blog posts and tutorials cover over 87% of the APIs and people view those pages more than the documentation (over 70 millions). Finally, Singer [118] too finds that the developers prefer code example than the documentation in case of learning the APIs.

A few initiatives are found in the literature that investigates how the code examples for learning the APIs need to be presented in the online social forums and blogs. Nasehi et al. [82] investigate how a code example at StackOverflow can be a good resource to learn APIs. In case of questions, the code examples or the problems the developers can be beneficial to the future if the title and the tags are appropriate and complement the questions. On the other hand, an answer will be helpful if the answer contains concise code, question's context, step by step solutions, external links, multiple solutions, inline documentation, and the limitations of the solutions and APIs. Due to the large popularity of these online forums, Treude and Robillard [127] propose a technique, SISE, that augments the sentences at the StackOverflow posts to specify the API mentions. Their result shows that StackOverflow contains a lot of useful information about the APIs that are missed by the documentations. In addition to using online forums as the means of learning APIs, researchers also use the forums to find the obstacles of the APIs by mining the posts. Wang and Godfrey [132] investigate the obstacle of using Android and IOS APIs based on the StackOverflow posts and find over 3 thousand posts complaining about the different issues while using them.

Code search is very much related to learning APIs since in most cases, the first step to learn an API either through documentation or online forums is searching for it by a search engine. However, Rahman et al. [103] believe that natural language query-based search engines such as Google, Bing and so on are not the most appropriate to find the related code on the internet. Thus a number of code search techniques [19, 21, 39, 48, 55, 68, 75, 100, 101, 103] are proposed in the literature. Rahman et al. [101, 103] propose a couple of code search techniques based on the word embedding and exploiting keyword-API associations. Raghothaman et al. [100] propose a natural query into a code description of the usages of an API based code synthesis technique, SWIM. Gu et al. [39] propose a neural encoder decoder based natural language to code sequence based code search technique. Despite the techniques are useful, but due to a lack of comparison of the natural language-based search engine such as Google, the effectiveness of the techniques are not guaranteed.

3 Deep API Method Argument Completion

Method Call completion involves completing the method name and the arguments associated with the method. Traditional method name completion techniques leave the task of argument completion in the hand of developers whereas the argument completion techniques expect the method name completed by the developers. Code completion techniques treat method names and arguments similar to other code tokens. In this chapter, we discuss our first study that proposes a solution that can complete the method name along with the arguments in a sequence.

The rest of the chapter is organized as follows. Section 3.1 introduce the problem and discusses the proposed system for method call completion. We describe our proposed technique in Section 3.2 and summarize the evaluation procedure in Section 3.3. Section 3.4 presents the evaluation results including related discussions. Section 3.6 describes related work. Section 3.7 discusses threats to the validity of our work. Finally, Section 3.8 summarizes the chapter.

3.1 Introduction

These days, developers heavily reuse application programming interfaces (APIs) of frameworks and libraries for building applications instead of engineering those from the scratch [10, 20, 144]. To meet developers' requirements and to offer better flexibility, these frameworks and libraries are continuously growing in their sizes [70]. As a result, it often becomes difficult for a developer to remember every detail of the API specifications for use. To remove the burden of remembering every detail, a large number of code completion systems have been developed. When a developer starts typing code, a code completion system provides suggestions to complete the remaining code. Murphy et al. found that code completion is one of the top ten commands used by Java developers [81]. Among various code completion systems, the most popular is the method call completion systems [10,20,84,88,98]. However, these systems leave the task of completing method arguments to developers. Unfortunately, finding the correct argument is also a non-trivial task [20, 144] and need to be dealt with.

The declaration of each method contains a name and expected type of its parameters. Although method name and the static types of arguments are useful, they are not sufficient enough to effectively complete method arguments. Traditional code completion techniques (e.g., Eclipse JDT) suggest methods based on their types and thus find relevant method names in top-k infrequently. For example, in the example, Figure 3.1(a), Eclipse JDT[5] does not find the correct method in top-6 suggestions. It finds *BitSet* as the type of the

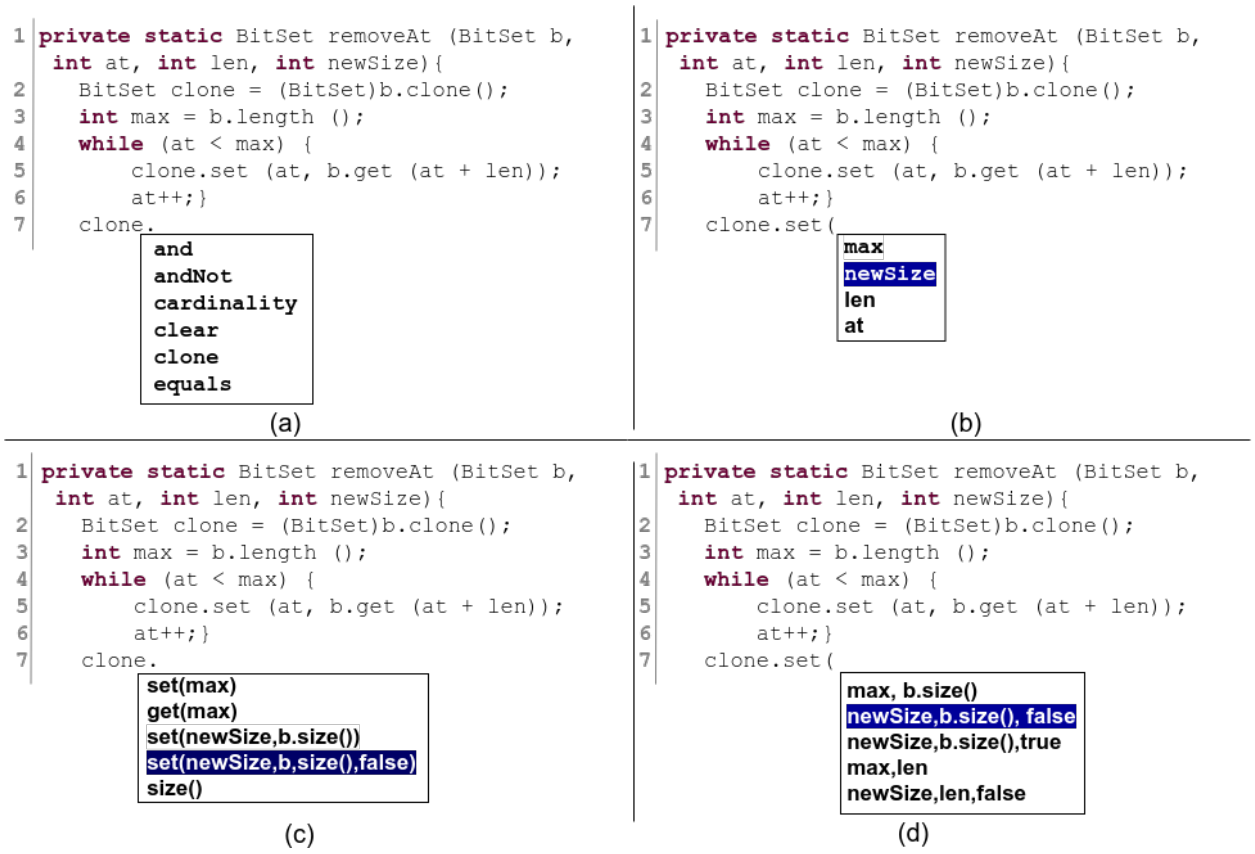


Figure 3.1: An example[1] of method call recommendation(1a), method argument recommendation(1b), method call cum argument recommendation by DAMCA (1c) and method argument recommendation by DAMCA (1d)

method and returns all method name in alphabetic order. Thus the required method call, *set*, remains very low in the list. Machine learning based approaches [10,98] will find the method name based on the number of instances of such occurrence. In case of argument completion, Eclipse JDT finds the first two arguments being *Integer* from the method definition. Therefore, it populates the recommendation list with all *Integer* type variables (Figure 3.1b). However, the actual argument for second position is a method invocation (*b.size*) that returns an integer value. Other code completion techniques [43, 45, 106, 135] treat arguments as well as method name same as all other code tokens. However, method call and arguments require a lot more semantic interpretation than other tokens such as if, for statement, variable initialization and so on. Thus they fail to recommend argument due to having no type information. Few techniques [89], consider type information but they limit themselves to fixed contexts. These lead to wrong or even no suggestion [85].

To the best of our knowledge, `Precise` [144] is the first technique that addresses the problem of completing arguments of method calls. It creates a parameter usage database and completes an argument query using the k-nearest neighbor algorithm between query context and argument usage contexts collected from previous code examples. Asaduzzaman et al. developed another technique, called `PARC` [9], that leverages source code localness property, static type analysis and string similarity measure to recommend argument. Liu

et al. leverage the similarity between the method parameter and the argument name to develop a name-based argument recommendation system [66]. However, there has been still a marked lack: for example, name-based argument recommendation system cannot recommend arguments of complex expression types (such as when the argument is a method call). PARC [9] collects argument usage context leveraging syntactic structure of source code and considers token similarity for recommending arguments. However, by considering embedding of tokens, one can capture both syntactic and semantic similarity between argument usage context in relation to the argument used in a method call. This can lead to better recommendations. Furthermore, the above systems assume that developers have selected or typed the correct method call. This assumption isolates method argument completion from method call completion. We argue that the isolation is not really necessary and incurs additional time to complete a method call. This is because developers need to request the code completion system at each method call and parameter positions in order to make selections. Thus, recommending method calls with arguments would reduce the burden of requesting and selecting individual arguments. There are a number of code completion and program synthesis systems that can recommend an arbitrary sequence of tokens which can also complete method calls and their arguments [43,84–86,89,106,135]. However, the performance of these systems has not been evaluated for completing method arguments.

In order to address the above-mentioned issues, we develop a technique, called DAMCA, that can recommend method calls with arguments. We formulate the problem of completing method calls with arguments as a machine translation problem. The technique leverages a Bi-directional Long Short Term Memory (LSTM) neural network [115] that encodes the input sequence to a vector of fixed dimension. Another LSTM is used to decode the target sequence from the vector. Instead of considering token similarity, the technique embeds prior tokens into a vector representation of context so that semantically related tokens can be identified. In order to mitigate the effect of the long sequence, our technique incorporates an Attention mechanism [13]. The choice of deep learning technique for our problem is due to its superior performance for sequence to sequence learning problem in natural language processing domain [14, 22, 23, 116, 122].

We evaluate the effectiveness of our proposed technique using ten large software systems and three popular software frameworks and libraries. We also compare our technique with method call completion, method argument completion, arbitrary sequence of token recommendation and program synthesis techniques to understand the effectiveness of our approach. For both intra and cross-project settings, our technique outperforms the compared techniques by 3-28% in prediction accuracy [58] and BLEU score [92], and 10-30% in Mean Reciprocal Ranking [25].

This study makes the following two contributions:

1. A technique that leverages a deep neural network based sequence to sequence model to complete method calls with arguments.

2. An extensive comparison of our proposed technique with existing state-of-art method call completion, method argument completion, arbitrary sequence of token recommendation and program synthesis techniques.

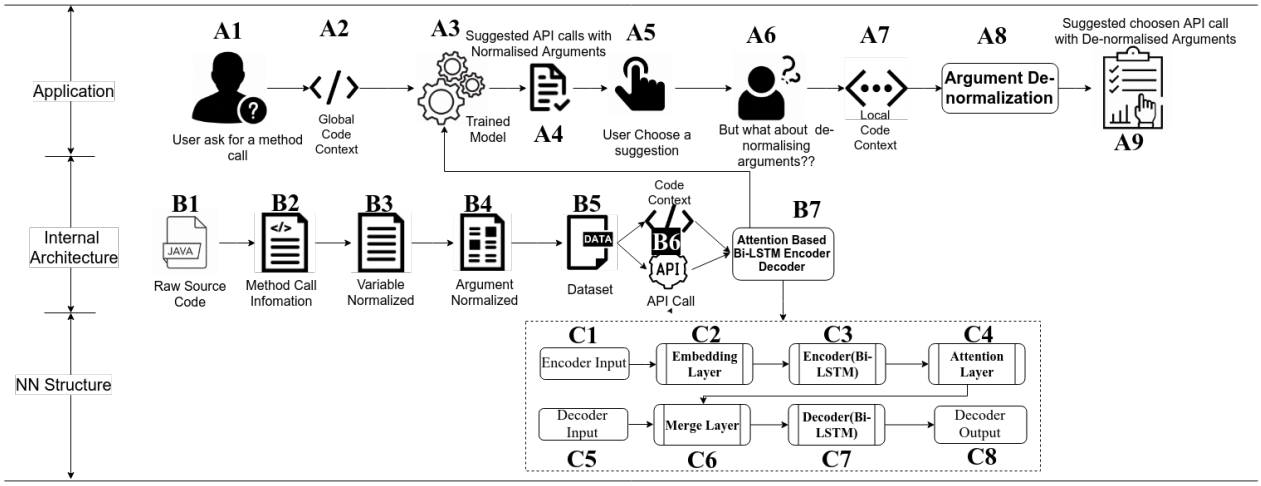


Figure 3.2: Application, Internal Architecture and Neural Network Structure of DAMCA

3.2 Proposed Approach

This section describes the proposed approach, **D**eep **A**PI **M**ethod **C**all cum **A**rguments recommendation system (DAMCA). The main functionality of DAMCA is to suggest relevant API method calls along with the list of argument(s) as a sequence. The application, internal architecture and neural network structure of the proposed approach are shown in Figure 3.2. The Application is the front end of our approach that gives a higher level abstraction of how the proposed technique works. The Internal Architecture and Neural Network Structure are the back end layers that describe how our proposed technique generates suggestions.

Application of DAMCA from Figure 3.2 can be stated as follows: when a user seeks for a method call(A1), the proposed approach collects all contexts(A2) and passes them in to the trained model(A3). The Trained model is created using the sequence learning technique described in Section 3.2.2. A list of API method calls with normalized arguments in sequences are returned from the trained model(A4). When the user selects a suggestion(A5), DAMCA de-normalize the arguments(A6-A8) returns the list of arguments for the chosen method call as the final output of the system(A9).

Internal architecture involves data preprocessing(B1-B6), neural network creation(B7), train, test, and Top-k predictions.

3.2.1 Data Preprocessing

The input to our internal architecture is raw Java source code(B1). We extract information about the methods call from source code(B2), normalize variables names(B3), normalize arguments(B4) and process the data for creating contexts and labels(B5-B6). The sub-processes are discussed below.

Extracting a method call information

The source code is found meaningful for automated analysis when represented in all three forms: lexical, syntactic and semantic [86, 129]. On the other hand, naturalness, repetitiveness and localness properties of source code discovered by a number of studies [45,128] strengthen automated techniques [4,9,10,43,53,89,106]. We thus give more focus on nearby tokens with all possible representations while collecting method call information. The collected tokens are of two types: the code tokens in the line of method invocation and the code tokens of neighboring n lines. For example, the code tokens in the line for method invocation of $key.substring(idx, key.indexOf(`/'))$ in Listing 1 are *key* and $=$. We follow the order of the nearest to the farthest. Thus after rearranging, the line's code tokens are $=$ and *key*. Similarly, the code tokens from the neighboring n lines are the tokens that occur within the current scope. Among the collected tokens in line or neighbours, lexical tokens are names or keywords: for example, the qualified name of the receiver variable (e.g., *java.lang.String*), method name (e.g., *substring*), argument(s) (e.g., *idx* and $key.indexOf(`/'))$, list of the previous method called on the receiver variable (e.g., *[lastIndexOf]*), and list of the previous method called on the arguments. Syntactical tokens are the tokens extracted from Abstract Syntax Tree(AST) (e.g., *String, Task*). These tokens represent the syntax structure of the code. Semantic tokens are the normalized variables and arguments(discussed in succeeding sub-sections), method name associated with qualified name/type of receiver variable(e.g., Var:String.lastIndexof). Such tokens are more purposeful when they are associated with type information [86]. Last, embedding layers in the neural network create semantic graphs between all lexical, syntactical and semantic tokens.

```
1 private void parseURL(URL url){
2     String key = url.toString();
3     final int idx = key.lastIndexOf('#');
4     key = key.substring( idx, key.indexOf('/'))
5
6 Method Name: substring
7 Receiver Type: java.lang.String
8 Argument(s): SN:int, Var:String.indexOf
9 Line Code Tokens(Nearest to farthest): = Var:String
10 Neighbour Code Tokens(Nearest to farthest): Var:String.lastIndexof = Var:int
     final Var:Url.toString = Var:String Var:Url
11 Previous Method call on receiver: [lastIndexOf]
12 Previous Method call on arguments:[]
13 Abstract Syntax Tree:String Task
```

Listing 3.1: Intermediate representations of example code for DAMCA

Variable Normalization

Variable names are some nouns, and they differ from one code to another. So, keeping them in the original format will create a huge dictionary size with loosely connected tokens in it. We need to reduce them by normalizing into some common formats. Moreover, variables represented in their type have better performance in code completion previously [86,89]. In our example, variables occur as an argument, assigning expression, initialization expression and receiver of a method call. Apart from the arguments, we normalize the variables by their types. A common format of *Var:<data type>* for all of the variables except arguments is used. For example, we replace the variable name *key* as *Var:String*, *idx* as *Var:int*. Afterward, we pass the arguments in the argument normalization process.

Argument Normalization

There are 28 argument expression types in Java Development Tool[2]. Table 3.1 shows the eleven types that are supported by DAMCA with corresponding example and the normalized style. In our approach, we normalize the arguments according to the expression type. For example, the first argument of *substring* method is a simple name and its type is integer, so we normalize as, *SN:<Data Type>*. Therefore, the first argument will be *SN:int*. On the other hand, the second argument is a method invocation. There is a receiver variable in the method invocation. We normalize the receiver variable following variable normalization step. Therefore, the second argument will look like *Var:String.indexof*. A question will arise, what happens when the method invocation at the argument have parameter(s) in it? For example, our second argument $key.indexOf(`/')$ itself needs an argument completion. We recommend up to method name for such cases because whenever DAMCA finds another method invocation needs completion, it will start the whole procedure by itself. Now, we prepare the context and label sequence for training.

Creating context and label sequence

Our label sequence is a method and its argument(s). We create the method as *method name: qualified name of receiver* to represent its meaning semantically. Then the list of normalized arguments is followed by spaces. Therefore, for our example the label will look like *subString:java.lang .String SN:int Var:String.indexOf()*.

To create the context sequence, we follow the following order: 1) qualified name of receiver variable, 2) previous method call on receiver variable, 3) previous method call on arguments, 4) line code tokens (nearest to farthest), 5) neighbor code tokens (nearest to farthest), and 6) AST tokens. The motivation behind such ordering is to provide useful information earlier. For example, the first output token is a method name. It is very much related to the receiver type and the sequence of the method call in it. Although the Attention mechanism (described in Section 3.2.2) will take care of important tokens, we are still just assisting the network manually.

[2] `http://bit.ly/335ZS6v`

Table 3.1: Argument Expression type

Argument Expression Type	**Example**	**Normalized arguments**
Simple Name	int a = 1, b=5; String s = "Hello World"; s = s.subString(**a**,**b**)	**SN:Int SN:Int**
Qualified Name	s.equals(**System.out**);	**QN:System.out**
Method Invocation	s.indexOf(**s.length()**)	**VAR:String.indexOf**
Boolean Literal	System.exit(**True**)	**Boolean**
Character Literal	s.concat(s,**'a'**)	SN:String **Character**
Number Literal	s.substring(**0**,**4**)	**Number Number**
Null Literal	System.clearProperty(**null**)	**null**
String Literal	s.equals(**"ABC"**)	**String**
Class Instance Creation	new BufferedReader(**new FileReader()**)	**Class:FileReader**
Cast Expression	s.charAt(**(int)'a'**)	**(Int)Character**
This Expression	System.identifyHashCode(**Main.this**)	**this:Main**

Questions might arise that why we are preprocessing the raw data even though deep learning is capable of extracting features? Won't passing the raw source code be enough for deep learning? The answer is clearly demonstrated in a study by Tufano et al. [129], where they showed how different representations of source code can improve the accuracy of code similarity. Another study by Nguyen et al. [86] collects all three (lexical, syntactic, semantic) representations of a code fragment, passes into three deep networks and combines the results to generate the next code token. Deep learning is indeed a powerful tool that extracts important parts of the code as features. This is done by the non-linear activation function at the nodes of the hidden layers [116]. However, the main challenge is how we present our code to the deep neural network. In one of our experiments (Section:3.5.1), we demonstrate how the performance of DAMCA improves due to representing code tokens in all possible way.

3.2.2 Attention Based Encoder Decoder

We adopt the Attention based Bidirectional Long Short Term Memory (Bi-LSTM) Encoder Decoder approach for recommending method with normalized arguments. Encoder Decoder is one of the implementations of the deep sequence to sequence learning [22]. The implementation includes two neural networks: one for the encoder and other for the decoder. Encoder creates the hidden representation of input sequence. Hidden layer value of last time step of the encoder is saved in a vector called thought vector. Decoder starts operation after receiving the thought vector and generate/decode output tokens based on thought vector and output token of the previous time step. In practice, the vanilla encoder-decoder model cannot remember long sequence by thought vector [13]. Furthermore, remembering all tokens is not that efficient too since more irrelevant token would be included [39]. To resolve such issues, the Attention mechanism is proposed and

widely used [13, 39, 139]. The basic idea for the Attention mechanism is to assign attention weights with a corresponding hidden layer of encoder [13]. Therefore, rather than creating a single thought vector, the model creates thought vectors for each time step of the encoder. The functionality of attention layer is to provide the amount of attention the decoder should pay to hidden layers of the encoder while decoding at a particular time step [13]. Therefore, to calculate the thought vector tg at jth time step of decoder, one must follow equation 3.1.

$$tg_j = \sum_{t=1}^{T_x} \alpha_j(t) h_i(t) \tag{3.1}$$

$$\alpha_j(t) = \frac{exp(e_j(t))}{\sum_{i=1}^{T_x} exp(e_j(i))} \tag{3.2}$$

Here, tg_j denotes the thought vector at jth time step, T_x denotes the number of time steps of encoder, $\alpha_j(t)$ means the attention weights of tth hidden layer of encoder and $h_i(t)$ is the hidden layer of ith timestep of encoder for jth timestep of decoder. Attention weights $\alpha_j(t)$ is calculated by the Equation 3.2. In equation 3.2, $e_j(t)$ is the attention factor that can be calculated by operating a neural network.

Workflow of the attention based Bi-LSTM encoder-decoder along with train and test operations is shown in Algorithm 1.

Algorithm 1: Attention based Bi-LSTM Encoder Decoder

Input: Training Context c_{train}, Training Label l_{train}, Testing Context c_{test}, Testing Label l_{test}

Output: Top-K recommendation f_{num}

```
Create context dictionary dict_context based on c_train
Create label dictionary dict_label based on l_train
for each contextString in c_train and each labelString in l_train do
    Convert contextString and labelString into numerical vector numC_train and numC_test based on
     the dict_context and dict_label respectively
end
Configure Attenstion based Bi-LSTM Encoder Decoder
model = train(numC_train, numC_test)
for each contextString in c_train and each labelString in l_train do
    Convert contextString and labelString into numerical vector numC_train and numC_test based on
     the dict_context and dict_label respectively
end
recommendation_list = test(model, numC_train, numC_test)
f_num = De_Normalize_Arguments(recommendation_list)
return f_num
```

The algorithm starts from building two different dictionaries for context and label. Each dictionary is

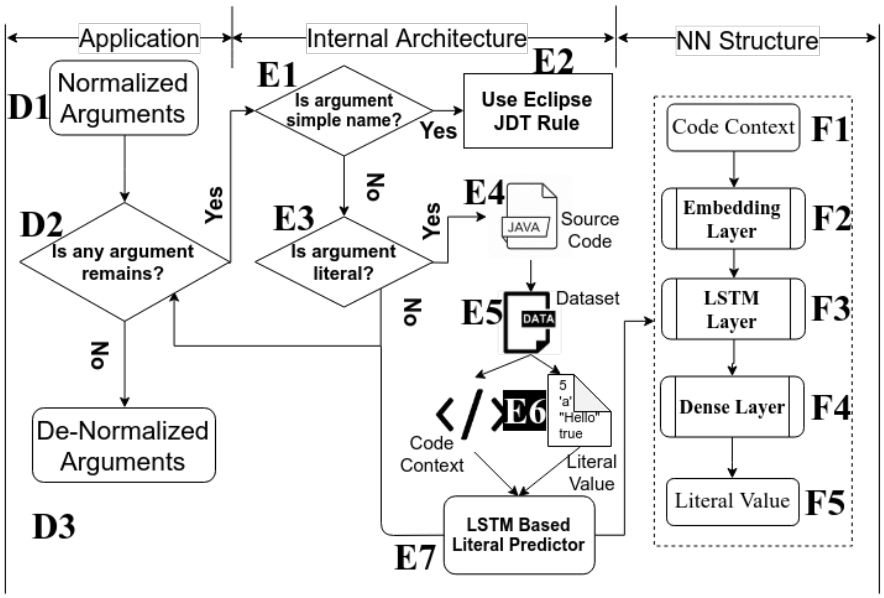


Figure 3.3: Unfolding Argument De-Normalization(A8)

initialized with four tokens, *padd* for padding sequence, *unk* for unknown token, *sos* for start of sequence and *eos* for end of sequence. Next, each unique token is assigned with an index value. Based on the index value of both dictionaries we converted the context and the label sequence into numerical vectors.

Context numerical vector(C1) is passed through an embedding layer(C2) in the neural network shown in the NN Structure section of Figure 3.2. The layer creates a higher dimensional embedding matrix for the encoder. Next, a Bi-directional LSTM layer(C3) encodes the context and the values of the hidden layer are passed into the attention layer(C4). The result of attention layer and the decoder input(C5) are merged together(C6) and passed into the decoder(C7). Another Bi-directional LSTM decodes the merge matrix and returns hidden layers values to output layer(C8). Output layer is a softmax based dense layer that produces probabilities of all tokens in the label dictionary. The tokens with the highest probabilities are considered as the output of the following time steps.

3.2.3 De-normalize arguments

Attention-based encoder-decoder returns a list of API calls with normalized arguments in a sequence. For example *SN:Int* or *Var:String.i-ndexOf*. But the user needs actual variable name, not the type. So we need to de-normalize the returned argument. Figure 3.3 shows the overall process that unfolds the Argument De-normalization phase(A8) of Figure 3.2.

In the application layer, a user/software developer will see a list of normalized arguments that are sent to the process (D1). All normalized arguments are processed through a loop (D2) and a list of de-normalized arguments (D3) is returned. For each argument, if simple name expression type is recognized (E1), we

follow the Eclipse JDT Core's [3] four rules of recommending argument to de-normalize (E2). The rules are: 1) local variable has the highest priority than class and inherited class variable, 2) variables having more lexical similarity with method parameter have more priority, 3) unused variable get more preference than used one, and 4) closely declared variables have more precedence. In the example, the first argument is a simple name type. Data type of the argument is *int*. Only integer variable in the scope is *idx* which is unused and close to the method call. Thus it gets the highest priority by the rules of 1, 3 and 4. When the argument is of literal(E3), we attempt to predict the value using an LSTM based neural network. In order to achieve that we pass the source code(E4) in the recommendation system and it generates dataset(E6) that contains code context as input and literal value as label/output. Next, an LSTM based literal predictor(E7) generate top-1 literal value for each argument. In the neural network code context(F1) is passed through an embedding layer(F2). The embedded matrix through a LSTM layer(F3) creates hidden representation that is passed through the Dense Layer(F4) with *softmax* activation function. Token with the highest probability is considered as the predicted literal value. So, after de-normalization, the predicted method call sequence for our example becomes *subString:java.lang.String idx key.indexOf()*.

3.2.4 Train, Test & Top-K Prediction

Our training process is straightforward. We pass the context numerical vector as the encoder input, the label numerical vector having *sos* token as the decoder input and the label numerical vector having *eos* token as the decoder output. Teacher forcing method is used at every time step in the decoder and the error generated by categorical cross entropy [46] is used for back-propagation through time.

Testing with Beam Search

Testing is slightly different since we do not have the predefined decoder input and output. Like training, we pass the encoder input and only *sos* token as the decoder input. At every time step in the decoder, we operate the heuristic algorithm Beam Search [107] and pick the top-k/top-*beam-width* tokens based on the returned probabilities for next timestep. The Beam Search is used to calculate the probability of the next token with respect to all previous tokens. Therefore, if any irrelevant token gets a higher probability in the first or second-time step due to high frequency in the training corpus, it loses its position while decoding later sequences. For example, *SN: int* appears as one of the Top-k suggestions in the first timestep. But we are expecting a method name. The greedy search will operate the rest of the program keeping *SN: INT* as the first token of one of the suggestions. In the Beam Search, the multiplicative value of the previous outputs and predicted tokens of current time steps are sorted. Thus the initial irrelevant tokens are eliminated afterward. Beam Search is found useful for machine translation [57], code search [39] and so on. Testing ends when the decoder finds *eos* token in every sequence.

[3] `https://www.eclipse.org/jdt/core/index.php`

Filtering incompatible methods

One last issue is filtering methods from Top-k with incompatible receiver type. We are expecting the method name as the first token in our sequence. It could happen that method name that is not compatible with the receiver type can get a high probability. For instance, we are expecting methods of *String* class at listing 14 such as *substring, indexOf, equals* and so on as the first token in the output sequence. Unfortunately, *toArray* or *getHeight* methods appear which are incompatible with *String* class. To filter out such irrelevancy, we generate recommendations n times more than actual *beam-width*/k. For example, if *beam-width* is 3, then we generate $3n$ sequences. Checking the type compatibility of Top-3, we filter out the irrelevant one and populate the next available sequence. Finally, *Top-k* method and arguments in a sequence are returned as the output of the proposed technique.

3.3 Evaluation

This section describes subject systems, performance metrics, evaluation procedure and results of the evaluation.

3.3.1 Subject Systems

We consider ten different subject systems of varying sizes. All these systems are active in the development and have been used by several previous studies [9, 86, 144]. Among them, NetBeans[4] is the largest subject system. Both Eclipse[5] and NetBeans[4] are popular open source IDE. Lucene[6] is an information retrieval library. POI[7] is a Java library for manipulating various file formats. Cassandra[8] and Db4o[9] are examples of databases. While JGit[10] is a Java library that implements the Git version control system, iText[11] enables to manipulate PDF files. Batik[12] is a toolkit that helps to work with Scalable Vector Graphics (SVG) format documents. Finally, Ant[13] automates Java build processes. We consider Swing/AWT, SWT, and JDK API method calls along with their arguments. Our selection of libraries is based on the fact that all these libraries are very popular and a large number of applications are actively being developed using these libraries.

[4] https://netbeans.org/
[5] http://bit.ly/2QLSAmd
[6] https://github.com/apache/lucene-solr
[7] https://github.com/apache/poi
[8] https://github.com/apache/cassandra
[9] https://github.com/lytico/db4o
[10] https://github.com/eclipse/jgit
[11] https://github.com/itext/itext7
[12] https://github.com/apache/batik
[13] https://github.com/apache/ant

Table 3.2: Overview of the dataset.

Subject System	Version	Number of Files	LOC
NetBeans[4]	8.2.0	483,331	10.7M
Eclipse[5]	3.7.2	21,030	4.6M
Lucene[6]	4.6.0	7,863	1.7M
POI[7]	4.1.0	3,533	671K
Cassandra[8]	4.0.0	2,592	559K
Db4o[9]	7.8.0	5,556	498K
JGit[10]	5.3.0	1,512	392K
iText[14]11[]	7.1.6	1,709	366K
Batik[12]	1.11.0	1,650	360K
Ant[13]	1.9.6	1,310	282K

3.3.2 Experimental Settings

We use JavaParser[15] to parse source code and to identify locations of method calls that have arguments. We then use JavaSymbolSolver[16] to resolve bindings that find declarations connected to each element. This helps to determine the fully qualified name of arguments and method call receivers. We also collect tokens that appear prior method call locations. We call this the parameter usage context. Bi-LSTM encoder-decoder is implemented using Keras and Tensorflow Estimator at the backend having 128 embedding width, 264 hidden nodes, 0.2 dropouts, and *adam* weight optimizer with 0.01 learning rate. The hidden nodes are triggered with *tanh* activation function. *Softmax* activation with *categorical cross entropy* loss function are used at the output layer. An Intel(R) Xeon(R) CPU E5-2620 v2 with 2.10GHz with 16GB memory and one Nvidia K20 GPU is used for training that needs around 310 hours for 100 epochs.

We apply the 10-fold cross-validation technique [58] to measure the performance of the code completion systems. This is a popular way of measuring the performance of information retrieval systems and has been used by previous other studies. First, we divide the entire dataset into ten different folds, each containing an equal number of examples. Next, for each fold, we use code examples from the nine other folds to train code completion techniques. The remaining fold is used to test the performance of the technique. Thus, both our training and test data set consists of examples from different subject systems.

3.3.3 Performance Metrics

We use the BLEU score [92] to measure the performance. It is a popular metric for machine translation that calculates the similarity between machine translated sequence and actual sequence [39, 90]. In our case, the

[15]http://javaparser.org/
[16]https://github.com/javaparser/javasymbolsolver

actual sequence consists of a method call with arguments. We compare this with the sequence generated by code completion systems. BLEU score is calculated by considering the number of n-grams that are common between input and output sequences using Equation 3.3.

$$BLEU = BP \times \exp(\sum_{n=1}^{N} w_n log P_n) \tag{3.3}$$

$$P_n = (\frac{\#\ n-grams\ in\ actual\ output\ sequence + 1}{\#\ n-grams\ in\ recommended\ sequence + 1}) \tag{3.4}$$

N is the maximum number of grams. Each w_n acts as a weight to each P_n. And P_n means the precision of n-grams.We set $N = 4$ and $w_n = \frac{1}{N}$. The term BP states the brevity penalty. It works as the adjustment factors to penalize too short sequences since short sequences have the tendency to return higher precision value [92]. Equation for calculating BP is as follows:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-\frac{r}{c})} & otherwise \end{cases} \tag{3.5}$$

where r is the length of reference sequence and c is the length of candidate sequence. Let us see how BLEU score is calculated by an example. Consider the following method call: clone.set(newSize, b.size(), false). We generate the reference sequence by tokenizing and removing parenthesis, comma and dot (.) where the 1-grams are:{b,false, newSize, size, set}. Consider the generated method call:set(s, b.length(), false) and the 1-grams are {b, false, length, s,set}. Then, $P_1 = \frac{3+1}{5+1} = \frac{2}{3}$. Their 2-grams are: {(b,false), (false,newSize), (newSize,size), (size,set)} and (b,false), (false,length), (length,s), (s,set). Thus, $P_2 = \frac{1+1}{4+1} = \frac{2}{5}$. $P_3 = \frac{1}{3}$ and $P_4 = \frac{1}{2}$ as no 3-grams or 4grams is matched. Here, BP=1 as the length of recommended and actual sequence are same. Thus, the final BLEU score is calculated as follows: $1 \times exp(\frac{1}{5} \times \log \frac{2}{3} + \frac{1}{5} \times \log \frac{2}{5} + \frac{1}{5} \times \log \frac{1}{3} + \frac{1}{5} \times \log \frac{1}{2})$. In addition to that we calculate prediction accuracy(ACC) [58] and Mean Reciprocal Rank (MRR) [25].

3.3.4 Comparing with state-of-the-art techniques

We evaluate our proposed technique in three different ways. First, we evaluate the technique for completing method call with arguments. We can consider this problem as a sequence to sequence learning or recommending a sequence of tokens. Thus, we consider SLAMC [89] as a baseline technique. The technique incorporates semantic information into tokens by considering data type, scope, structural and data dependencies. The technique combines local context with global concerns using n-gram topic model and pair-wise association of program elements. Next, we develop a sequence to sequence learning using RNNLM [135]. The input and output of the technique are the same as ours. A recurrent neural network with 300 hidden units and 20 levels of contexts are used. SLP [43] is a hybrid technique that combines a deep neural network language model with a dynamic, scoped counting cache language model. A 6-gram cache language model is combined with an LSTM having 650 units at the hidden layer is used to construct the hybrid technique. DeepAPI [39] is a

deep learning based technique that given a natural language query can generate API usage sequences. While the objective of DeepAPI is different from ours, the technique also adopts a sequence to sequence learning approach. The subtle differences can be observed in data preprocessing, post-processing, types of RNN and hyperparameter values. We thus change the technique by replacing original input-output with that of ours. We also combine CSCC [10], a state-of-the-art method call completion technique with PARC [9], a state-of-the-art argument recommendation technique, to generate method calls with arguments. DNN4C [86] is another deep neural network language that can recommend the next code token. However, we cannot use the technique in our study because the technique is not publicly available. **Second,** we evaluate the performance of DAMCA for recommending method calls. For this purpose, we compare DAMCA with a method call completion technique, called CSCC. **Finally,** we evaluate DAMCA for recommending individual arguments and perform a comparison with PARC [9]. We adopt the same evaluation mechanism used by PARC. We consider that a developer has already selected the correct method call and now complete one argument at a time. At each argument position of target method calls, we asked both DAMCA and PARC to recommend arguments. We compare recommendations with reference values to calculate accuracy.

3.4 Evaluation Results

3.4.1 Evaluation Strategy-1: Recommending Method Calls With Arguments

In this experiment, we evaluate DAMCA for recommending method calls with arguments. We consider this as a sequence prediction task where the sequence consists of method calls and all required arguments. In this study, we evaluate DAMCA against six different techniques. These are SLAMC, RNNLM, SLP, DeepAPI, and CSCC+PARC. Here, the last technique consists of two different techniques. One focus on recommending method calls and the other focus on recommending arguments. We evaluate the performance of all compared techniques for both cross-project and intra-project settings.

DeepAPI shows the worst performance in our experiment. One major reason is that the technique is designed to map natural language queries and API usage sequences. Therefore, it fails to map between code tokens to the method call. API importance phase of DeepAPI [39] removes a lot of relevant arguments from the sequence and thus lessen the accuracy. RNNLM returns sequences with the accuracy between 5-28% for top-1 to top-10 positions. Deep neural network should capture pattern between data and perform better. However, passing lexical tokens to the neural network fails to represent the source code properly and underperforms in our data. While SLAMC can recommend token sequences, the technique fails to obtain accuracy over 38%. One major drawback is that the number of topics is very few compared to the diversity of data. Thus, semantically dissimilar tokens fall on the same topic and return unexpected results. For example, SLAMC put all class instance creation and array creation type arguments under the same topic. However, they exhibit very dissimilar characteristics. When we combine CSCC with PARC, the value reaches to 40%. This is because both CSCC and PARC collects method call contexts and parameter usage

Table 3.3: Comparison of sequence Accuracy and BLEU of DAMCA with related techniques (%) for cross-project settings

Subject System	Recc.	SLAMC		RNNLM		SLP		DeepAPI		CSCC+PARC		DAMCA	
		ACC	BLEU	ACC	BLEU	ACC	BLEU	ACC	BLEU	ACC	BLEU	ACC	BLEU
NetBeans	Top-1	9.34	11.24	5.34	8.28	18.69	20.34	3.71	5.47	11.81	13.75	**31.68**	**34.86**
	Top-3	18.25	21.49	16.35	18.62	31.25	36.73	11.74	13.50	21.72	24.91	**46.84**	**49.72**
	Top-10	30.34	33.73	25.62	28.40	43.76	46.49	19.32	21.01	34.67	36.76	**57.82**	**59.68**
Eclipse	Top-1	10.28	13.94	6.28	8.74	19.69	22.29	4.92	7.28	13.76	15.24	**32.43**	**35.46**
	Top-3	19.53	21.75	17.35	19.95	34.82	37.35	14.83	16.48	22.18	24.64	**47.24**	**49.82**
	Top-10	31.61	34.74	26.47	28.73	44.31	46.58	20.74	22.73	34.55	37.24	**58.36**	**61.40**
Lucene	Top-1	13.45	15.34	8.24	10.54	20.05	22.25	4.52	6.29	12.75	14.62	**31.76**	**34.18**
	Top-3	23.71	26.61	19.38	21.78	33.81	36.15	13.74	15.47	23.18	26.14	**46.92**	**49.85**
	Top-10	34.71	37.38	28.76	30.88	44.71	47.01	20.91	22.59	35.71	38.34	**58.06**	**60.76**
POI	Top-1	12.84	15.05	7.18	9.40	18.73	21.02	4.97	7.23	13.82	15.95	**32.43**	**34.92**
	Top-3	21.84	24.60	17.35	19.54	32.94	35.16	14.75	16.86	26.17	29.13	**47.24**	**50.44**
	Top-10	32.42	34.53	26.27	28.31	43.54	45.75	21.74	23.77	36.43	39.30	**57.35**	**60.19**
Cassandra	Top-1	11.27	13.53	6.41	9.12	19.76	22.09	3.92	6.07	12.95	15.80	**33.86**	**36.18**
	Top-3	19.51	21.75	14.26	17.22	31.72	34.09	10.29	12.14	25.76	28.05	**46.83**	**49.37**
	Top-10	32.81	34.86	23.41	26.16	44.05	47.15	18.54	20.21	35.94	38.40	**57.91**	**61.04**
Db4o	Top-1	14.26	16.54	4.31	6.76	20.91	23.73	4.18	5.72	13.71	16.34	**34.92**	**37.23**
	Top-3	23.76	25.88	13.25	15.29	33.84	36.25	11.95	14.05	26.29	29.08	**48.62**	**51.22**
	Top-10	35.81	38.32	21.75	24.36	44.86	47.45	19.26	21.61	36.17	38.99	**56.72**	**59.73**
JGit	Top-1	13.72	16.68	6.34	8.47	19.47	21.59	4.57	6.25	12.76	15.62	**33.49**	**36.53**
	Top-3	21.74	24.38	15.73	18.06	32.82	35.80	12.76	14.59	27.84	30.40	**47.24**	**49.73**
	Top-10	31.93	34.08	24.26	26.45	43.76	46.01	18.93	21.06	37.21	39.98	**57.13**	**59.44**
iText	Top-1	12.46	15.43	5.17	7.71	19.49	21.77	5.03	7.32	13.53	16.33	**32.82**	**35.60**
	Top-3	22.92	25.56	16.81	18.85	35.71	38.52	12.55	14.27	26.51	29.23	**46.18**	**48.72**
	Top-10	30.83	33.41	25.6	28.42	45.96	48.67	20.62	22.71	35.94	38.32	**57.63**	**60.57**
Batik	Top-1	14.86	17.77	6.08	8.61	17.61	20.59	4.87	7.25	12.95	15.69	**33.72**	**36.45**
	Top-3	23.43	26.14	15.75	18.47	33.72	36.21	11.83	13.55	26.84	29.56	**46.64**	**49.04**
	Top-10	34.82	37.69	23.81	26.60	44.72	46.87	20.05	22.26	36.08	39.01	**58.11**	**61.29**
Ant	Top-1	13.73	16.49	5.92	8.59	19.67	22.23	4.38	6.01	13.43	15.55	**32.94**	**35.69**
	Top-3	22.59	24.81	16.87	19.16	31.64	34.47	10.76	12.79	25.76	28.40	**47.05**	**49.68**
	Top-10	31.27	34.10	26.84	29.78	45.78	48.61	19.43	21.76	36.42	39.06	**58.27**	**60.69**

contexts separately and those contexts changes across examples. SLP becomes the second best technique for cross-project settings that contains accuracy between 17-45%. Although the technique considers an n-gram language model and an LSTM network, it passes lexical tokens to the model and thus fails to represent the data accurately. On the other hand, our proposed technique, DAMCA, outperforms every technique for all subject systems by 10-25%. Its because we explicitly focused on method call and arguments whereas the majority of the techniques focused on the general code completion. We provide all three lexical, syntax and semantic tokens and tried to catch the pattern globally. Therefore, it can skip the local biases of the code.

Second, we will compare all techniques along with hybridization of CSCC and PARC in inter-project settings. The average result of ten-fold cross-validation is reported at Table 3.4 where all three measures (ACC, BLEU and MRR) for top-1,top-3,top-10 of every technique is shown and best results are highlighted.

Table 3.4: Comparison of sequence Accuracy, BLEU and MRR of DAMCA with related techniques (%) for intra-project settings

Techniques	Recc.	ACC	BLEU	MRR
SLAMC	Top-1	21.64	24.96	0.21
	Top-3	34.18	37.82	0.33
	Top-10	49.88	51.87	0.42
RNNLM	Top-1	18.96	20.02	0.18
	Top-3	31.88	34.94	0.28
	Top-10	45.66	47.25	0.31
SLP	Top-1	35.52	37.24	0.35
	Top-3	50.37	53.19	0.43
	Top-10	61.83	64.73	0.54
DeepAPI	Top-1	21.28	24.25	0.21
	Top-3	36.74	39.25	0.44
	Top-10	52.98	55.84	**0.59**
CSCC+PARC	Top-1	33.49	35.49	0.33
	Top-3	49.16	51.92	0.31
	Top-10	60.58	62.53	0.47
DAMCA	Top-1	**52.27**	**55.63**	**0.55**
	Top-3	**63.41**	**65.84**	**0.48**
	Top-10	**71.94**	**74.08**	**0.59**

A common observation is that the performance of all techniques improves from cross-project settings by at least 10-20%. That proves the naturalness and localness property of our data. It suggests that, if a user builds a model using her code base along with the predefined model, (s)he can have more accurate recommendations with any of these techniques. Baseline machine learning(SLAMC), deep learning(RNNLM) and DeepAPI are the tier three approaches based on the performance. All of these techniques have an accuracy or BLEU score between 18% to 55%. Like cross-project settings, SLAMC fails to differentiate the functionality of code tokens having the insufficient topic number. RNNLM have poor data representation since they process in lexical format. DeeAPI is not suited in our problem as their technique is biased on mapping natural text and source code. SLP and CSS+PARC are the tier two techniques. This tier can also be called hybridization tier because SLP is the hybridization of machine learning and deep learning, and CSCC+PARC is the hybridization of a method and an argument recommendation techniques. Their accuracies fall between 33-65% which are 10-15% better than tier one. It directs us to an open research of hybridization of the algorithms to serve more accurate results. However, SLP bounds itself due to considering only lexical tokens. Like RNNLM, it fails to provide the neural network more perspicuous data. On the other hand, due to independent implementation,

CSCC and PARC fail to merge themselves for a better recommendation. For example, in many cases, PARC suggests accurate arguments since it has accurate method call information whereas CSCC fails to propose an accurate method call. Therefore, when we are merging them, the overall sequence fails to achieve an accurate recommendation. The vice-versa situation happens too and summed up to fall the accuracy value. DAMCA remains on tier one approach that has an accuracy difference of 20-30% from tier three and 10-20% from tier two. The proposed approach can suggest in top-10 almost three out of four times(70-75%) whereas tier one suggest a half time and tier two six out of ten. Along with different representations of data, Attention mechanism helps to point out the important encoded data while decoding. Moreover, filtering irrelevant method call helps the decoder to focus on less token. In the case of MRR comparison, Beam Search help our technique to uplift relevant sequences. For example, MRR of DAMCA for top-10 is 0.59, which suggest that on an average our suggestion can be found between first and second places. Beam Search helped DeepAPI and thus it has the same MRR value as DAMCA for top-10.

3.4.2 Evaluation Strategy-2: Recommending Method Calls

In our previous study, we evaluate the performance of DAMCA for recommending method calls with arguments. While DAMCA outperforms all compared techniques, we are interested in learning the effectiveness of DAMCA as a dedicated method call completion technique. For each test case, we ask DAMCA for recommending method calls with arguments. However, this time we determine the accuracy of method call recommendation only and compare the result with that of CSCC. Both intra-project and cross-project settings are considered where we calculate the average accuracy and MRR considering all subject systems. Table 3.5 summarizes results of the evaluation. BLEU score is irrelevant in this case since we only consider recommending a single token only.

Table 3.5: Comparison of DAMCA with CSCC for completing method calls only

Techniques	Recc.	Intra-Project		Cross Project	
		ACC	MRR	ACC	MRR
CSCC	Top-1	56.19	0.56	18.61	0.18
	Top-3	64.53	0.34	34.29	0.21
	Top-10	74.29	0.31	50.44	0.29
DAMCA	Top-1	**58.16**	**0.58**	**38.27**	**0.38**
	Top-3	**75.53**	**0.49**	**49.57**	**0.34**
	Top-10	**82.71**	**0.58**	**63.18**	**0.41**

DAMCA outperforms CSCC by 3-10% in accuracy and 3-45% in MRR for intra-project settings. While CSCC achieves 74.29% accuracy for top-10 recommendations, the value reaches over 82% for DAMCA.

The accuracy of both techniques drops in the cross-project setting. However, DAMCA managed to predict correctly in more cases compared to that of CSCC. For example, for the top-10 recommendations, DAMCA obtains around 63% accuracy. However, the value drops to around 50% for the CSCC.

3.4.3 Evaluation Strategy-3: Recommending Method Arguments

Recall that PARC generates a list of recommendations for each argument position. The first study evaluates the performance of DAMCA for recommending method calls with arguments and the second study evaluates DAMCA for recommending method calls only. However, we are interested in understanding the effectiveness of DAMCA for recommending arguments only. This type of recommendations is particularly useful when a developer cycles through different arguments of a method call. If the actual argument used in the source code does not match with any of the lists of recommendations, we consider this as an incorrect recommendation. Otherwise, we determine the rank of the correct recommendation and calculate the accuracy. Table 3.6 shows the accuracy of recommending arguments for both intra-project and cross-project settings.

Table 3.6: Comparison with PARC for completing method arguments.

Techniques	Recc.	Intra-Project			Cross Project		
		ACC	BLEU	MRR	ACC	BLEU	MRR
PARC	Top-1	36.39	38.53	0.36	11.51	12.36	0.11
	Top-3	51.35	53.17	0.31	23.71	26.75	0.16
	Top-10	61.34	62.35	0.26	30.68	33.91	0.19
DAMCA	Top-1	**45.36**	**47.34**	**0.45**	**22.08**	**25.07**	**0.22**
	Top-3	**62.75**	**65.98**	**0.41**	**43.65**	**46.19**	**0.29**
	Top-10	**72.96**	**75.02**	**0.42**	**52.95**	**55.67**	**0.35**

As an argument completion technique, DAMCA outperforms PARC for both intra-project and cross-project settings. While for the top-10 recommendations and for the intra-project setting PARC achieves 62.35% accuracy, the value reaches to 75.02% for the DAMCA. For the cross-project setting and for the top-10 recommendations PARC obtains 30.68% accuracy. The value reaches to 52.95% for the DAMCA. Both DAMCA and PARC do not perform well for cross-project settings because many arguments are only project specific.

3.5 Analysis and Discussion

3.5.1 Sensitivity Analysis:Impact of decision

This section justifies our choices for developing the deep learning model. Towards this direction, we develop deep learning models using different algorithms and information sources for recommending method calls with

arguments. Let, C_0 denotes the argument usage context consists of tokens that appear within the previous four lines prior to calling a method with arguments. We build our first model (M_1) using the context with a recurrent neural network (RNN). Let, C_1 consists of method names that are called on receiver and argument variable types including those tokens that appear in C_0. Our second model (M_2) combines C_1 with another recurrent neural network. We also collect tokens by considering the hierarchical structure of the source code. Consider that a method call with arguments appears in a while loop. The loop appears inside an if block where the if the block is located inside a method declaration. Then, we collect tokens from the conditional part of the if block, expression part of the loop and the declaration part of the method call (i.e., method name and parameter types). The process continues until we reach a type declaration. Let, C_2 consists of tokens collected from traversing hierarchical structure of source code including those that appear in C_1. We build our third model (M_3) combining C_2 with a recurrent neural network. The next two models are built combining Attention and Beam Search approach with C_2 and RNN (M_4 and M_5 respectively). The last two models are built by replacing RNN in M_5 with LSTM and Bi-LSTM networks. We refer to these models as M_6 and M_7 respectively. BLEU score for top-1,3 and 10 in intra-project settings are shown at Table 3.7.

Adding every piece of context affects the BLEU score in the sensitivity analysis experiment. For example, when we consider tokens in C_2 group, the BLEU score reaches to 15.63% whereas tokens in C_0 manage to gain accuracy just 11.91% for top-10 recommendations. However, context alone is not helpful. The inclusion of Attention mechanism and Beam Search increases the score to 38.69%. The most notable difference is observed when LSTM and Bi-LSTM are chosen over RNN. Vanishing gradient problem is the main factor behind this [47].

3.5.2 Argument Expression Type based analysis

Both DAMCA and PARC support eleven argument expression types. In this section, we analyze the accuracy of DAMCA and PARC for each of the eleven argument expression types for intra-project settings. From the argument completion results, we categorize each argument along with the top-10 recommendation list of DAMCA and PARC based on the expression type. Then we calculate the accuracy and report at Table 3.8. The first column denotes the name of the expression type followed by the percentage of test cases for each argument expression type, the accuracy of PARC and DAMCA. The last column indicates the percentage of improvement of DAMCA over PARC.

Around 2-29% improvement is observed for ten out of eleven expression types. The fundamental difference lies in capturing right expression type since de-normaliztion of argument for both DAMCA and PARC is not very dissimilar. Furthermore, for both Boolean and Null literal DAMCA has accuracy over 90%. This is because the variation of value for these two types are limited and number of test cases are enough for DAMCA to converge to predict right value. DAMCA performs poorly for cast expression type because of not having enough test cases to train.

Table 3.7: Sensitivity Analysis

Model	Description	Suggestion	BLEU
M_1	C_0+RNN	Top-1	1.23
		Top-3	6.37
		Top-10	11.91
M_2	C_1+RNN	Top-1	3.93
		Top-3	8.51
		Top-10	13.55
M_3	C_2+RNN	Top-1	6.03
		Top-3	10.76
		Top-10	15.63
M_4	C_2+RNN+Attention	Top-1	13.81
		Top-3	20.07
		Top-10	31.14
M_5	C_2+RNN+Attention+Beam Search	Top-1	21.92
		Top-3	28.73
		Top-10	38.69
M_6	C_2+LSTM+Attention+Beam Search	Top-1	35.76
		Top-3	42.44
		Top-10	52.99
M_7	C_2+Bi-LSTM+Attention+Beam Search	Top-1	**55.63**
		Top-3	**65.84**
		Top-10	**74.08**

3.5.3 Manual and statistical analysis: Why does DAMCA perform well?

To order to further examine the superior performance of DAMCA, we conduct a manual analysis. For our motivational example at Figure 3.1, no techniques other than DAMCA return correct method call sequence. SLAMC returns *clone.clone()* since, it stores all methods of *Bitset* class in same topic and frequency of *clone()* method is the highest there. Moreover, in the dataset *clone()* function appears twice within 4-10 lines and SLAMC captures that pair association rule. RNNLM could not even predict a method because it captures receiver variable *clone* as the method name *BitSet.clone()*. Thus it returns *at, max*. CSCC+PARC finds *length()* as the final output. Individually, CSCC returns actual method call *set()* in its list but PARC has better recommendation score for method call *size()* with no parameter. SLP return current method call name but wrong arguments: *set(at,len)*. Due to cache model it finds right method but could not find the

Table 3.8: Argument Expression Type based analysis

Argument Expression Type	% of test cases	PARC	DAMCA	Improvement
Simple Name	37.17	80.78	**86.67**	5.89
Method Invocation	28.90	43.56	**59.10**	15.54
Numberr Literal	14.87	58.50	**77.11**	18.62
Qualified Name	8.38	26.28	**54.29**	28.01
Boolean Literal	4.59	92.50	**95.51**	3.01
String Literal	2.29	39.90	**45.62**	5.72
Character Literal	1.40	46.24	**53.16**	6.92
Null Literal	1.15	85.14	**91.68**	6.54
Class Instance Creation	0.66	46.25	**56.59**	10.34
This Expression	0.49	44.64	**46.37**	1.73
Cast Expression	0.10	**11.86**	10.17	-1.69

arguments. It even can't recognize that second argument can be a method call. But how DAMCA returns correct sequence? DAMCA normalizes the last four lines and found a *set()* there, all the parameters of the parent node *removeAt()* are stored while collecting AST and in its local context second argument of *set()* is a method call. Therefore, Bi-LSTM encoder decoder with Attention mechanism extracts the relations between code tokens by exploting them in higher dimensions to generates the most relevant sequence.

To validate our evaluation BLEU score for sequence prediction in both cross and intra-project settings, we operate paired t-test for dependent means. The value of t varies from 44-70 in cross-project and 21-41 in intra-project settings. For both settings, the p-score for all compared techniques are less than 0.00001 and thus reject the null hypothesis. This proves the results obtained by DAMCA is statistically significant from all others.

3.6 Related Works

3.6.1 Code Completion/Suggestion

Code completion systems that complete either method calls or their arguments are related to our study. A large number of method call completion systems have been proposed in the literature. For example, Bruch et al. propose an example based method completion systems, called Best Matching Neighbors (BMN) that uses the K-NN algorithm to recommend method calls [20]. Porksch et al. propose an extension of BMN algorithm that uses the pattern based Bayesian network for recommendation [98]. Hou and Pletcher develop a code completion technique, called BCC, that use sorting, filtering, and grouping of APIs [50, 51]. Asaduzzaman et al. develop a context-sensitive code completion technique, called CSCC, that leverages a locality sensitive hashing and string similarity measure for method call recommendations [10]. DroidAssist

supports method call completion and learns API usages from byte code of android mobile application [88]. APIRec is another method call completion technique that integrates fine-grained code change information with code context [84]. However, none of these techniques focus on completing method arguments. While a large number of code completion systems have been focusing on method call completion, only a few techniques focus on completing arguments of the methods. These are Precise, proposed by Zhang et al. [144], PARC developed by Asaduzzaman et al. [9] and LexSim proposed by Liu et al. [66]. Both Precise and PARC build argument usage databases by mining previous code examples and collecting different code contexts. LexSim leverages string similarity measure between the parameter and the argument names for its recommendations. While they report acceptable accuracy, they cannot provide the arguments unless a method call was selected by the developers. Method call completion is not supported in these techniques and thus they are of limited use. There are other forms of code completion systems. These include but not limited to API usage pattern completion [85,87], method body completion [44] and completion of next token or a sequence of code tokens [45, 86, 89, 106, 128]. A number of studies are also found in program synthesis domain that can generate code token/sequence. Some of them uses machine learning [39,80,117,119,138] and others user rule-based approaches [30, 100]. However, evaluation for code completion or program synthesis techniques in case of the method with argument completion is missing in the literature. Our approach shows superior performance when we compare with different state-of-art method calls, code and arguments completions, and program synthesis techniques.

3.6.2 Deep Learning in SE

Deep learning techniques have been applied to solve various software engineering research problems. This includes but not limited to neural language model based code completion [43, 86, 106, 135], bug localization [60, 131], code similarity/clone detection [64, 129, 134, 145], code search [38, 39] and deep learning model testing [125]. A few of these studies are also related to our study. Hellendoorn and Devanbu propose a cache enabled n-gram language model, called SLP, that can handle frequent changes in source code, large vocabularies and deeply nested scopes [43]. They found that the modified n-gram language model performs better than RNN or LSTM language models. Nguyen et al. propose a deep neural network based language model, called DNN4C [86]. Three different models are created from the same source code and three hidden layer representations are generated. A merge layer combines all three representations and generates the next code token accordingly. SLANG collects sequences of method calls to complete multiple statements [106]. While SLANG supports argument completion to some extent but the support is very limited. DeepAPI also uses the sequence to sequence learning approach to convert natural language query to a sequence of APIs. Comparison results of SLP and DeepAPI with our proposed technique suggests that passing the lexical token to the deep neural network cannot guarantee superior results. Moreover, DeepAPI is domain specific which cannot be fitted for our problem. Other two (DNN4C and SLANG) could not be included in our study due to unavailability of implementation.

3.7 Threats To Validity

We have identified the following threats to the validity of our research.

First, we evaluate code completion systems using ten software systems written in the Java language. One can argue that the result may not generalize for other systems and for different programming languages. However, we would like to point to the fact that those subject systems are quite large in size and have been actively developed by a large base of users. We thus believe that our proposed model would work well with other Java systems.

Second, we collect `JDK`, `Swing/AWT`, and `SWT` method calls for evaluating code completion systems. Thus, one can claim that we cannot generalize our findings for APIs of other frameworks and libraries. Given that `JDK, Swing/AWT, and SWT` consist of a large number of APIs to support different domains of applications, we believe that such cases would be highly unlikely and the result we obtain should largely carry over.

Third, we focus on eleven argument expression types in this study and ignore others. This is because examples of other expression types are very small which make it impossible to train the model. Furthermore, previous studies also follow the same strategy [9].

Fourth, we re-implement SLAMC because the technique is not publicly available. Although we cannot guarantee that our implementation of the technique does not contain any errors, we spent considerable time in replicating and testing the technique to ensure its correctness.

3.8 Summary

This chapter proposes a novel deep learning based code completion technique, called DAMCA, that recommends both method name and its argument(s) as a sequence. Previous studies focus on either suggesting method call names or their arguments. Few code completion techniques can suggest both but due to lack of type information, they fail to recommend appropriate method call with the argument. Our proposed technique solves the problem using a Bi-directional LSTM based Encoder Decoder with Attention Mechanism and Beam Search. Comparison of our proposed technique for ten large subject systems with other neural network models and with existing state-of-art code completion systems shows that our proposed technique is more robust and efficient. For both intra and cross-project settings, DAMCA outperforms every technique by 5-25% in accuracy and 10-30% in MRR. Our sensitivity, argument expression type, manual and statistical analysis strengthen the superiority of the proposed approach over all others. The proposed technique can help the developers learning the API usages within the IDE. Furthermore, the technique is viable when the code is compilable. However, developers also found to learn the APIs and their usages from social online forums. Those code snippets are not always compilable which leads to our next study. In Chapter 4, we thus explore why the code snippets on the social online forums hinder the learning process and propose a technique for finding the fully qualified name of the APIs in the online forum code snippets.

4 Learning from Examples to Find Fully Qualified Names of API Elements in Code Snippets

The first study in Chapter 3 explores the API learning mechanism when the code snippets are written with the IDE and are compilable. The developers can learn API usage by choosing the suggestion proposals made by DAMCA. However, developers also explore different online forums such as StackOverflow, GitHub to learn the APIs and their usages. More often those code snippets are found to be not compilable. One primary reason is the API elements within the online forum code snippets have no Fully Qualified Names. Thus learning from these code snippets or reusing the code snippets remain challenging. In this chapter, we discuss our second study that proposes a context sensitive type solver technique, COSTER. It collects a variety of contexts and finds the Fully Qualified Name of the API elements.

The rest of the chapter is organized as follows. Section 4.1 introduces the problem along with the proposed technique for solving the problem. Section 4.2 presents a motivating example and explains the challenges in resolving FQNs of API elements. We describe our proposed technique in Section 4.3. Section 4.4 introduces datasets and explains the evaluation procedure. Section 4.5 evaluates COSTER against two other state-of-the-art techniques and Section 4.6 provides further insights on the performance of our proposed technique. We discuss threats to the validity of our work in section 4.7 and Section 4.8 presents prior studies related to our work. Finally, Section 4.9 summarizes the chapter.

4.1 Introduction

Developers extensively reuse Application Programming Interfaces (APIs) of software frameworks and libraries to save both development time and effort. This requires learning new APIs during software development. However, inadequate and outdated documentation of APIs hinder the learning process [27, 28]. As a result, developers favor code examples over documentation [118]. To understand APIs with code examples, developers explore online forums, such as Stack Overflow (SO)[1], GitHub Issues, GitHub Gists[2] and so on. These online forums provide a good amount of resources regarding API usages [94]. However, such usage examples can suffer from external reference and declaration ambiguities when one attempts to compile them [28, 120]. External reference ambiguity occurs due to missing external references, whereas declaration ambiguity is

[1] `https://stackoverflow.com`
[2] `https://gist.github.com/discover`

caused by missing declaration statements. As a result of these ambiguities, code snippets from online forums are difficult to compile and run. According to Horton and Parnin [141] only 1% of the Java and C# code examples included in the Stack Overflow posts are compilable. Yang et al. [49] also report that less than 25% of Python code snippets in GitHub Gist are runnable. Resolving FQNs of API elements can help to identify missing external references or declaration statements.

Prior studies link API elements in forum discussions to their documentation using Partial Program Analysis (PPA) [26], text analysis [12, 28], and iterative deductive analysis [120]. All these techniques except Baker [120], need adequate documentation or discussion in online forums. However, 47% of the APIs do not have any documentation [99] and such APIs cannot be resolved by those techniques. Baker [120] depends on scope rules and relationship analysis to deduce FQNs of API elements. However, the technique fails to leverage the code context and cannot infer 15-31% of code snippets due to inadequate information within the scope [96]. Recently, Statistical Machine Translation (SMT) is used to determine FQNs of APIs in StatType [96]. However, the technique requires a large number of code examples to train and it performs poorly for APIs having fewer examples. The training time of StatType is also considerably higher than other techniques.

In this study, we propose a context-sensitive type solver, called COSTER. The proposed technique collects locally specific source code elements as well as globally related tokens as the context of FQNs of API elements. We calculate the likelihood of appearing context tokens and the FQN of each API element. The collected usage contexts and likelihood scores are indexed based on the FQNs of API elements in the occurrence likelihood dictionary (OLD). Given an API element as a query, COSTER first collects the context of the query API element. It then matches the query context with that of the FQNs of API elements stored in the OLD, and then rank those FQNs leveraging three different scores: likelihood, context similarity, and name similarity scores.

We compare COSTER against two state-of-the-art techniques for resolving FQNs of API elements, called Baker [120] and StatType [96], using more than 600K code snippets from GitHub [73] and two different Stack Overflow (SO) datasets. We not only reuse the SO dataset prepared by Phan et al. [96] but also build another dataset of 500 SO posts. Results from our evaluation show that COSTER improves precision by 4-6% and recall by 3-22% compared to state-of-the-art techniques. COSTER also needs ten times less training time and one third less memory than StatType that is considered as a state-of-the-art technique for this problem. We also investigate why the proposed technique outperforms others through extensive analyses on i) sensitivity, ii) number of libraries, iii) API popularity, iv) receiver expression types, and v) multiple mapping cardinality. Thus, the contributions of this study are as follows:

1. A technique that leverages a context-sensitive approach to resolve the FQN of an API element.

2. Evaluation of the proposed technique against two state-of-the-art techniques.

3. Extensive analyses on the results of all competing techniques.

4.2 Motivating Example

Let us consider a code snippet collected from a SO post[3] as shown in Fig. 1. The post describes a situation where a developer wants to use the *Element* and *Document* classes, but (s)he does not know which libraries or APIs need to be imported.

```
1 private void writeFile(){
2   dFact =DocumentBuilderFactory.newInstance();
3   build = dFact.newDocumentBuilder();
4   doc = build.newDocument();
5
6   Element root = doc.createElement("outPutResult");
7   doc.appendChild(root);
8
9   for(Result r:resultList){
10    Element|title = doc.createElement("Title");
11    title.appendChild(doc.createTextNode(r.getTitle));
12    root.appendChild(title);
13
14    Element add = doc.createElement("Address");
15    add.appendChild(doc.createTextNode(r.getAddress));
16    root.appendChild(address);
17  }
18 }//End of Write function
```

Figure 4.1: A Stack Overflow post[3] regarding how to use the *Element* class

What are the challenges in resolving the FQNs of this code snippet? First, the code in online forums is not always compilable or runnable. For example, in Fig. 1, the code snippet is incomplete, having not been enclosed by a class. Thus, we cannot compile or run the code directly as more changes are required.

Second, the code snippets often contain identifiers without declarations. In Fig. 1, identifiers *dFact*, *build*, and *doc* at line 2, 3, and 4, respectively are not declared within the code snippet. While completing the declaration statements of these identifiers, declaration ambiguity will occur because of missing type information.

Third, API elements used in a code snippet require specific external references. For example, the classes *DocumentBuilderFactory*, *Result* and *Element* at lines 2, 6 and 9 require external references.

Last but not least, API elements can have name ambiguity. For example, there are five *Element* classes in JDK 8[4] and it is not clear which Element class we should import to compile the code.

To tackle the above challenges, existing techniques either use rules or heuristics [28,108,120], or statistical machine translation [96]. Rule-based systems (such as RecoDoc [28] and ACE [108]) search documentation or discussion to resolve the types. However, they have three limitations: documentation is rarely available [99],

[3] https://stackoverflow.com/questions/20157996/
[4] https://docs.oracle.com/javase/8/

discussions are usually informal [120] and using Partial Program Analysis [26] results in partially qualified names [96]. Baker [120] resolves a type by deducing the candidate FQNs based on the tokens within the scope of that type. The declaration of an API element can be located outside the current scope and Baker fails to resolve the FQN of that API element. For example, the undeclared variables *build* and *dFact* at lines 2 and 3 caused insufficient information for Baker [120]. Moreover, increasing the number of libraries also increases the likelihood of mapping the same token name to multiple APIs with a similar name in the oracle. That creates name ambiguities and Baker has too little information to tackle such ambiguities. To overcome the limitations of rule-based systems, StatType [96] used locally specific resolved code elements to find the regularity of co-occurring tokens. However, StatType requires a large number of training examples to perform well. Moreover, the technique also requires a long training time. These motivate us to investigate the problem further.

Key Idea:

Instead of relying only on the locally specific code elements (i.e., local context), COSTER also considers globally related token (i.e., global context) of an API element. Such combination is found effective in other research areas [5, 10, 84]. The definitions of the local and global contexts are as follows:

Definition I Local Context: The local context of an API element consists of method calls, type names, Java keywords and operators that appear within the top four and bottom four lines including the line in which the API element appears. For example, local context of *root.appendChild(title)* at line 12 of Fig. 1 is {*for*, *Result*, *Element*, =, *createElement*, *appendChild*, *createTextNode*, *getTitle*, *Element*, =, *createElement*, *appendChild*, *createTextNode*, *getAddress*, *appendChild*}.

Definition II Global Context: The global context of an API element consists of any methods that are called on the receiver variable, or use either the receiver variable or the API element as a method parameter, and located outside the top and bottom four lines of that API element. The global context of *root.appendChild(title)* at line 12 of Fig. 1 is {*appenChild*} since the *appendChild* method at line 7 uses the receiver variable *root* of the API element as a parameter.

Local context captures the naturalness [45] and localness [128] properties of the code. On the other hand, global context tries to capture the long term dependency of the API element. The motivation behind choosing global context is mainly because they enrich the context of an API element by adding the tokens that are related to the element but do not closely located. For example, in Fig. 1, the local context of *doc*, *root*, *title* and *add* have the method name *appendChild*. Therefore, co-occurrence based on local context will suggest that all four are the object of the *Element* class. However, when we add the global context, *doc* will have other methods, such as *createTextNode*, and the other three will have the *appendChild* only. Thus, the global context differentiates *doc* from the other three and helps to infer more accurate FQN.

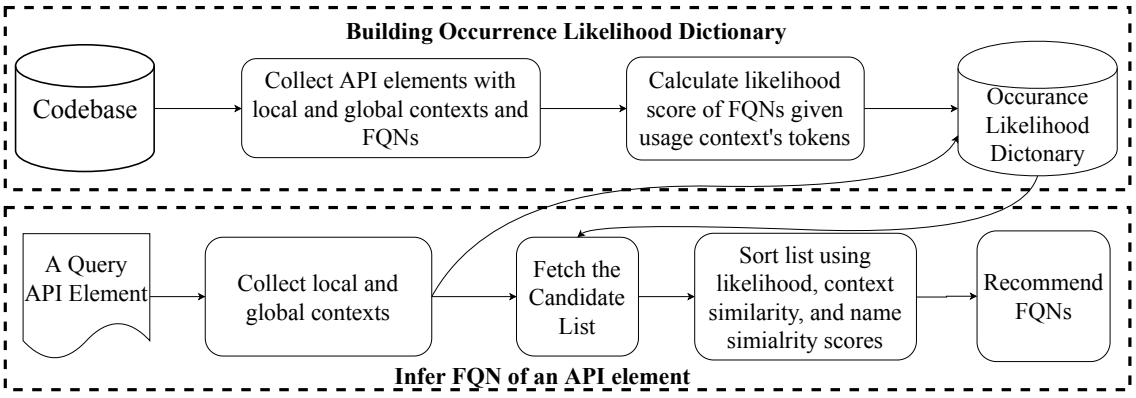


Figure 4.2: Overview of COSTER's entire process of building OLD and recommending FQN of a query API element

4.3 Proposed Technique

This section describes our proposed technique for finding FQNs of API elements, called COSTER (**Co**ntext **S**ensitive **T**ype Solv**er**). Fig. 4.2 shows an overview of the proposed technique. Our example-based context-sensitive technique works in two steps as follows:

- **Build Occurrence Likelihood Dictionary (OLD)**. We collect two different forms of contexts: local context as per Definition I and global context as per Definition II (see Section 4.2, Key Idea) for each API element; i.e., a method call, a field call or a type variable. Next, we combine them based on the position in the source code to form the usage context. Finally, we calculate the likelihood of appearing usages context tokens and the FQN of each API element. Collected usage contexts and likelihood scores are indexed based on the FQNs of API elements in the OLD.

- **Infer FQN of an API element**. This involves searching for any FQN in OLD whose usage context matches with that of the target API element. COSTER collects only those FQNs whose usage contexts share a minimum number of tokens with the target API element. We called this the candidate list. Next, we synthesize FQNs from the candidate list leveraging a) likelihood scores of contexts in the candidate list, b) cosine similarity score between the usage contexts in the candidate list and the usage context of the target API element, and c) name similarity score between the candidate FQNs and the name of the target API element using the Levenshtein distance. A combined similarity score is calculated and the technique sorts FQNs in the candidate list in descending order of their combined similarity score. We recommend the top-k FQNs after removing any duplicates.

We describe each of these steps in detail as follows.

Table 4.1: Example code snippet [113] with context, likelihood, context similarity, name similarity scores and possible FQN candidates

```
1 private static int countFlips(String stack) {
2     Set<String> visited = new HashSet<>();
3     Queue<State> bfsQueue = new LinkedList<>();
4     visited.add(stack);
5     bfsQueue.add(new State(0, stack));
6     while (!bfsQueue.isEmpty() && !isSolved(bfsQueue.peek().pancakes)) {
7         State state = bfsQueue.poll();
8         for (int i = 1; i <= state.pancakes.length(); ++i) {
9             String flipped = flip(state.pancakes, i);
10            if (!visited.contains(flipped)) {
11                bfsQueue.add(new State(state.flips + 1, flipped));
12                visited.add(flipped);
13            }
14        }
15    }
16    return bfsQueue.poll().flips;
17 }
```

API Element: *bfsQueue.add*

Local Context:{*State, =, poll, for, int, =, <=, length, ++, String, =, if, !, contains, State, +, add*}

Global Context: {*add, isEmpty, peek, poll*}

Combined Context: {*add, isEmpty, peek, State, =, poll, for, int, =, <=, length, ++, String, =, if, !, contains, State, +, add, poll*}

Candidate	Candidate Context	Likelihood Score	Context Similarity	Name Similarity	Candidate FQN
c_1	*..., add, isEmpty,..., peek,....., for, poll*	0.51	0.47	0.33	*java.util.Queue*
c_2	*for, int, ..., =, String, ..., if, ...*	0.31	0.27	0.00	*java.lang.String*
c_3	*....., if, contains, ..., isEmpty,....., String*	0.26	0.24	0.07	*java.util.List*
c_4	*poll,..., for, ..., peek, add,..., isEmpty*	0.21	0.36	0.33	*java.util.Queue*
c_5	*.., add, ..., poll, ..., for, .., peek*	0.17	0.21	0.10	*java.util.LinkedList*
...	...	...	...	...	...

4.3.1 Building Occurrence Likelihood Dictionary (OLD)

In this step, we build a dictionary of usage context of API elements that will be used to infer the FQN of query API element. To do that, COSTER uses Eclipse JDT[5] to parse source code examples and collects usage context of API elements (e.g., method calls, class names, and field calls) including their FQNs. The usage context of an API element consists of two different contexts: local and global contexts defined previously.

The FQN of each API element and the corresponding usage context together constitute a transaction. We then calculate the likelihood of appearing a context token and the FQN of the corresponding API element leveraging the trigger pair concept of Rosenfeld [61]. If a token t is significantly correlated with the FQN f_p of an API element p, then t can be considered as a trigger for f_p. However, instead of using maximum entropy as was used by Rosenfeld [61], we estimate the likelihood of the FQN f_p given the token t appeared

[5] https://www.eclipse.org/jdt/

in the usages context by considering the ratio of transactions that contain both t and f_p ($N(t,f_p)$) over the number of transactions that contain the t ($N(t)$) as shown below. Thus, the ratio represents the likelihood score ($ls\ (f_p|t)$) between a token and the FQN of the corresponding API element.

$$ls\ (f_p|t) = \frac{N(t, f_p) + 1}{N(t) + 1} \tag{4.1}$$

To include the distance into consideration between p and t (i.e., the more p and t are closely located, the higher will be the likelihood score between them), we update the likelihood score ($ls(p,t)$) calculation as follows:

$$ls\ (p, t) = ls\ (f_p|t) \times \frac{w_{weight}}{distance(p, t) + k} \tag{4.2}$$

Here, w_{weight} represents the weight of the token and k is a small positive number. We set the value of k to 0.0001 for our experiments to avoid division by zero. If the token is located in the local context, we set the weight value to 1; otherwise, the weight value is set to 0.5. The distance between the token and the API element, referred to $distance(p, t)$, is calculated by considering the number of tokens between p and t. The closer the token t to the API element p, the smaller would be the distance. Given a set of tokens (i.e., $T = \{t_1, t_2, t_3, ..., t_n\}$) as the usages context, we calculate the likelihood score of the FQN f_p of the corresponding API element p by summing the scores for all pairs of $\{p,\ t_i\}$, as shown in Eq. 4.3.

$$\begin{aligned} \log(ls(f_p|T)) &\simeq \log(ls\ (p, T)) \\ &\simeq \log(ls(p, t_1)) + \log(ls(p, t_2)) + ... + \log(ls(p, t_n)) \end{aligned} \tag{4.3}$$

We note that to avoid the underflow, we use the logarithmic form. The collected usage contexts and likelihood scores are indexed based on the FQNs of API elements in the OLD.

4.3.2 Inferring FQN of an API element

This section discusses the steps we follow to determine the FQN of an API element.

Context Collection

Given an API element for which FQN needs to be determined, COSTER collects both local and global contexts of the API element. Let us consider the API element *bfsQueue.add* as shown in Table 4.1. We follow the same approach as described in the previous subsection to collect both local and global contexts of API elements. The global context for the above-mentioned example consists of the following four method calls: *add*, *isEmpty*, *peek*, and *poll*. Next, we combine tokens of local and global contexts to form a combined context that preserves the order of tokens. Combined Context at Table 4.1 shows the context for our example. From now on we refer to the combined context as the query context, API element of the query context as the query API element, and the FQN of the query API element as the query FQN.

Candidate list generation

Our next step is to select FQNs from OLD along with their contexts and likelihood scores where each context matches with the query context. We select only those FQNs whose combined context shares at least 25% of the tokens with that of the query context. The choice of the threshold value of 0.25 (25%) is made by running the inference step for different values and getting the most stable performance for 0.25. Our query context in Table 4.1 has 17 unique tokens in it. Therefore, if any contexts in the OLD has a minimum of $17 \times 0.25 \approx 4$ shared tokens, we include that in the candidate list.

Context similarity calculation

We now have a list of candidate contexts along with their FQNs, and we need to calculate how similar are they to the query context. The goal of this step is to find similar contexts that not only contain similar tokens but also those tokens that appear in the same order. Thus, we calculate the cosine similarity [77] score and multiply that with the fraction of matched tokens that are in the same order of query context to obtain the context similarity score as follows:

$$Sim_{context}(T_q, T_{ci}) = \frac{N_{order}}{N_{matched}} \times \frac{T_q \cdot T_{ci}}{||T_q||\,||T_{ci}||} \tag{4.4}$$

In Eq. 4.4, T_q and T_{ci} are the numerical vector representations of the set of tokens of the query context and each candidate context, respectively, N_{order} is the number of tokens in order and $N_{matched}$ is the number of tokens matched. In the case of our example at Table 4.1, the column *Context similarity* shows the similarity score between the query context and each candidate context.

Name similarity calculation

During our manual investigation of FQNs of API elements, we observe that the names of the API elements that share similarity with the FQN are most likely the desired output. To leverage such similarity in our ranking, we calculate the Levenshtein distance [63] between the name of the query API element (p_q) and the candidate FQNs (f_{ci}). The distance simply calculates the number of edits required to attain a particular FQN from the query API element. The smaller the number of required edits, the higher would be the similarity between the name of the query API element and a candidate FQN. Thus, we calculate the name similarity score using the following equation:

$$Sim_{name}(p_q, f_{ci}) = \begin{cases} 1 - \frac{lev(p_q, f_{ci})}{len(f_{ci})} & \text{if len>lev} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

In Eq. 4.5, $lev(p_q, f_{ci})$ and $len(f_{ci})$ refer to the Levenshtein distance between the query API element and each candidate FQN, and the length of the candidate FQN, respectively. Column name similarity in Table 4.1 represents the name similarity scores between the API element *bfsQueue* and the candidate FQNs.

We found that *java.util.Queue* has the highest name similarity score having the number of edits required as 10. Therefore, the name similarity score becomes: $(1 - \frac{10}{15}) = 0.33$. *java.lang.String* requires 16 edits which is the same as the length of FQN. Thus, the name similarity score becomes zero.

Recommending top-k FQNs

The candidate FQNs are sorted in descending order of the similarity scores calculated using Eq. 4.6.

$$\begin{aligned} candidate\ score(f_{ci}) = \alpha \times ls\ (f_{ci}|T_{ci}) \\ + \beta \times Sim_{context}(T_q, T_{ci}) \\ + \gamma \times Sim_{name}(p_q, f_{ci}) \end{aligned} \tag{4.6}$$

Here, $ls\ (f_{ci}|T_{ci}))$ is the likelihood score of the candidate FQN f_{ci} given the set of tokens in the candidate context T_{ci}. Moreover, α, β, and γ are the coefficients of likelihood, context similarity, and name similarity scores, respectively. The values of these variables are determined using Hill Climbing Adaptive Learning algorithm [121] over the training data. For our example in Table 4.1, the final score for *java.util.Queue*, *java.lang.String*, *java.util.List* and *java.util.LinkedList* based on Eq. 4.6 are 0.68, 0.38, 0.34 and 0.27, respectively. Our technique recommends the top-k FQNs of API elements after removing any duplicates and the value of k can be adjusted.

4.4 Evaluation

This section compares COSTER with two state-of-the-art techniques, Baker [120] and StatType [96]. To evaluate COSTER, we answer the following three research questions:

- **RQ1: Intrinsic Accuracy.** How accurate is COSTER in identifying FQNs of API elements in Java source code snippets collected from Github dataset [73]?
- **RQ2: Extrinsic Accuracy.** How accurate is COSTER in identifying FQNs of API elements in Java code snippets collected from Stack Overflow posts?
- **RQ3: Timing and memory performance.** Does COSTER improve the timing and memory performance compared with Baker and StatType?

All experiments were performed on a machine with an Intel Xeon processor having a processing speed of 2.10 GHz, 16 GB of memory, and running on Ubuntu 16.04 LTS operating system.

4.4.1 Dataset Overview

We collected datasets from two different sources for evaluating our technique and for comparing with the state-of-the-art techniques. A brief overview of the datasets is shown in Table 4.2.

Table 4.2: Dataset Overview

GitHub Dataset		**Stack Overflow Dataset**	
Info	Number	Info	Number
No. of Projects	50,000	No of Posts	500
No. Of Files	602,173	LOC	3,182
No. of Libraries	100	No. of Libraries	11
No. of Classes	19,259	No. of Classes	203
No. of Methods	99,473	No. of Methods	1,375
No. of Fields	21,739	No. of Fields	624

GitHub Dataset: We consider a collection of 50K Java projects collected from GitHub, called 50K-C projects [73]. We use the term GitHub Dataset to refer to the dataset as shown in Table 4.2. The dataset consists of 19K unique classes/types, 99K unique methods, and 21K fields. Our selection of this dataset is based on the fact that all these projects are compilable and include all required dependencies in the form of jars to resolve FQNs of all APIs. We select the top frequent 100 libraries used by these projects. Then we use Eclipse JDT[5] to parse the source code and to resolve FQNs of all API elements for those libraries.

Stack Overflow Datasets: We leverage two different Stack Overflow (SO) datasets to conduct the extrinsic experiment. First, we consider the SO dataset used in the study of StatType [96]. We use the term StatType-SO to refer to this dataset. We also built another dataset by collecting code snippets from SO posts considering eleven popular libraries, referred to as COSTER-SO. Out of eleven libraries, ten are selected as the top frequent libraries of GitHub dataset and the remaining one is JDK8. We downloaded the latest SO data dump to collect code snippets. For each selected library, we searched the class, method and field names in the code snippet to identify library posts. Similar to StatType [96], we collected code snippets from both questions and answers. We then randomly collected 500 code snippets with an equal number of code snippets selected for each library of interest. Code snippets in SO often do not contain required import statements, variable declarations, class names or method bodies. To resolve FQNs, we need to convert those code snippets to compilable Java source files by incorporating those missing information. Five annotators, all are computer science graduate students, made those code snippets to compilable code snippets by manually incorporating the missing information. The dataset consists of API elements from 203 unique classes, 1,375 unique methods, and 624 unique fields, as shown in table 4.2.

4.4.2 Evaluation Procedure

In the case of intrinsic evaluation, we apply the 10-fold cross-validation technique to measure the performance of each technique for resolving FQNs of API elements. We divide the dataset into ten different folds, each containing an equal number of API elements. Nine of those folds are used to train and the remaining fold is

used to test the performance of the competing techniques. We use precision, recall, and F_1 score to measure the performance of each of the techniques. For each API element in the test dataset, we present the code example to each technique to recommend the FQN of the selected API element. If the target FQN is within the top-k positions in the list of recommendations, we consider it relevant. The precision, recall, and F_1 are defined as follows.

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \tag{4.7}$$

$$Recall = \frac{recommendations\ made \cap relevant}{recommendations\ requested} \tag{4.8}$$

$$F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{4.9}$$

Here, *recommendations requested* is the number of API elements in the test set. The *recommendation made* is the number of cases the technique can recommend FQNs. We use two-tailed Wilcoxon signed-rank test [136] for our study. For each evaluation metric (i.e., precision, recall and F_1 score), we collect the result of COSTER for each fold as one data point and compare ten data points we obtain from ten different folds with that of Baker and StatType. Since we are performing two comparisons (i.e., comparing COSTER with StatType and Baker), our result can be affected by the type I error in null hypothesis testing. To minimize the error, we restrict the false discovery rate (FDR) by adjusting the p-values using Bonferroni correction [2]. If adjusted p-values are less than the significance level then we reject the null hypothesis (i.e., statistically, the results of COSTER are significantly different than Baker and StatType).

For the extrinsic evaluation, we evaluate the effectiveness of all the competing techniques in recommending FQNs of API elements in SO code snippets. We train each technique using code examples of GitHub dataset and then test the technique using two different SO datasets. We then compute precision, recall and F_1 scores for each dataset separately. For both of the state-of-the-art techniques (i.e., Baker and StatType), we use the settings used in their prior studies.

4.5 Experimental Result and Analysis

This section presents the evaluation results and answers research questions described in Section 4.4.

4.5.1 RQ1: Intrinsic Accuracy. How accurate is COSTER in identifying FQNs of API elements in Java source code snippets collected from Github dataset [73]?

Table 4.3 shows the evaluation results for all three candidate techniques on the GitHub dataset. We determine the performance of candidate techniques for top-1, top-3 and top-5 recommendations. Table 4.3 only shows the top-1 recommendation for Baker since the technique only recommends the best match.

Results from our evaluation show that Baker gives comparatively lower precision and recall. While the precision for the top-1 recommendation is 83.63%, the recall drops to 68.19%. Compared with Baker,

Table 4.3: Precision (Prec.), Recall(Rec.) and F_1 score (F_1) of all competing for GitHub dataset [73]

Techniques	Recc.	Prec.	Rec.	F_1
Baker	Top-1	83.63	68.19	75.12
	Top-3	-	-	-
	Top-5	-	-	-
StatType	Top-1	85.91	86.74	86.32
	Top-3	89.34	90.76	90.04
	Top-5	91.74	92.47	92.10
COSTER	Top-1	**89.48**	**90.04**	**89.76**
	Top-3	**92.11**	**93.26**	**92.68**
	Top-5	**95.43**	**95.84**	**94.63**

StatType improves both precision and recall by 2.28% and 18.55%, respectively. Among the three compared techniques, COSTER obtains the best precision and recall. While the precision is 89.48%, the recall reaches to 90.04% for the top-1 recommendation. Thus, COSTER achieves 5.85% higher precision and 21.85% higher recall in comparison with Baker and 3.57% higher precision and 3.30% higher recall than StatType. These indicate the effectiveness of the context that COSTER collects to capture the usages of FQNs of API elements. Performance improves as we increase the number of recommendations. For example, the precision and recall for the top-5 recommendations are 95.43% and 95.84%, respectively for COSTER. Statistically, the precision, recall and F_1 scores of COSTER are significantly different than the compared techniques for top-1, top-3, and top-5 recommendations.

4.5.2 RQ2: Extrinsic Accuracy. How accurate is COSTER in identifying FQNs of API elements in Java code snippets collected from Stack Overflow posts?

Table 4.4 shows the evaluation results for the StatType-SO dataset considering the topmost recommendation. The dataset consists of API elements from six different libraries. Since StatType performed better than Baker for this dataset in their experiment [96], we only report results for StatType and COSTER.

Interestingly, as we see from Table 4.4 for the Hibernate library, COSTER obtains 3.9% and 8.9% higher precision and recall compared to that of StatType. For the remaining five libraries, the differences between the evaluation results of StatType and COSTER are very small. StatType has marginally better precision for four libraries but COSTER obtains a slightly better recall. While the Precision of COSTER ranges between 93.7% and 98.4%, the recall ranges between 95.2% and 99.6%.

Next, we compare all three techniques for COSTER-SO dataset and the results are shown in Table 4.5. Similar to the intrinsic experiment, Baker recommends only top-1 with comparatively poor performance. The

Table 4.4: Precision (Prec.), Recall(Rec.) and F_1 score (F_1) comparison between StatType and COSTER for StatType-SO

Libraries	Total APIs	StatType			COSTER		
		Prec.	Rec.	F_1	Prec.	Rec.	F_1
Android	1,022	**98.7**	97.9	**98.3**	98.4	**98.1**	98.2
Joda Time	652	**98.3**	98.0	**98.1**	97.6	**98.4**	98.0
XStream	463	**99.8**	**99.6**	**99.7**	99.3	**99.6**	99.4
GWT	1,243	96.6	95.9	96.2	**97.1**	**96.5**	**96.8**
Hibernate	840	89.8	86.3	88.0	**93.7**	**95.2**	**94.4**
JDK	2,934	**98.9**	**99.1**	**99.0**	97.6	98.8	98.2

technique obtains 87.34% and 75.92% precision and recall, respectively. StatType obtains 3.31% and 15.74% higher precision and recall compared to that of Baker. COSTER outperforms both Baker and StatType for top-1 recommendation by obtaining 92.17% precision and 93.27% recall. For top-3 and 5 recommendations, COSTER achieves 1-3% more precision and recall than StatType. Similar to intrinsic experiment, statistical test after adjusting p-values shows that the results of COSTER are significantly different than Baker and StatType.

4.5.3 RQ3: Timing and memory performance. Does COSTER improve the timing and memory performance compared with Baker and StatType?

This section compares the time and memory performances that include training and testing times, and the sizes of vocabulary, language model and dictionary. The sum of times required to parse source code to identify API elements and to determine their FQNs is reported as the code extraction time in Table 4.6. Training time includes the creation of OLD, training any machine learning model and so on. Inference time refers to the time needed to detect the FQN of a query API element. Vocabulary, language model, and dictionary sizes refer to the number of words in the vocabulary, size of the language model (if any), and the size of the dictionary (if any), respectively. To have a fair comparison, all these techniques were run on the same machine for GitHub dataset.

Baker requires the least amount of time for parsing source code whereas COSTER takes 30 more minutes to collect the usage context of all API elements in the Github dataset. StatType, on the other hand, requires more time, possibly because of generating source and target languages, and to check the alignment between them. Baker does not require any training time since it simply stores the APIs in the dictionary without calculating any scores. COSTER calculates the likelihood of the FQN of each API element given usage context tokens in the training code examples (i.e., likelihood scores) and builds the OLD. It takes around 11 hours to complete these operations. StatType requires significantly higher training time. It takes more than

Table 4.5: Precision (Prec.), Recall(Rec.) and F_1 score (F_1) comparison between all competing techniques for COSTER-SO dataset

Techniques	Recc.	Prec	Rec.	F_1
Baker	Top-1	87.34	75.92	81.23
	Top-3	-	-	-
	Top-5	-	-	-
StatType	Top-1	90.65	91.66	91.15
	Top-3	93.76	94.86	94.31
	Top-5	95.73	97.05	96.39
COSTER	Top-1	**92.17**	**93.27**	**92.72**
	Top-3	**96.65**	**97.09**	**96.87**
	Top-5	**98.27**	**98.95**	**98.61**

Table 4.6: Timing and memory performance for all three competing techniques

	Baker	StatType	COSTER
Code Extraction Time (hrs)	**7.9**	9.1	8.2
Training Time (hrs)	-	109	**11**
Inference Time (ms)	6.2	**4.3**	5.2
Vocabulary Size (n words)	**1.7M**	7.9M	2.8M
Language Model Size (GB)	-	6.9	-
Dictionary Size (GB)	**1.63**	-	2.3

100 hours to train. One can argue that training is a one-time operation. However, we would like to point to the fact that supporting a new library would require training the technique. Such a long training time can increase the cost significantly if a user leverages any web services for model training. For example, on Amazon EC2[6], StatType will cost more than 200 USD to train the technique only once whereas COSTER will cost between 18-20 USD. In the case of inference, COSTER requires 0.9 milliseconds more than StatType to determine FQN of a query API element. The difference is negligible and can be ignored.

Baker has the least memory requirement, having 1.7 million tokens in the vocabulary that requires 1.63 gigabytes of memory. Having just 0.9 million more tokens and 700 megabytes more memory, COSTER performs significantly better than Baker. StatType requires about three times the number of tokens and memory required by COSTER. In short, the results in Table 4.6 show that our proposed technique is capable

[6]`https://aws.amazon.com/ec2/pricing/on-demand/`

to exhibit the best performance (reported in Table 4.3), requiring one-tenth training time and one-third memory than StatType. Thus, our proposed technique can be considered as efficient, not only in terms of accuracy but also in terms of timing and memory requirements.

4.6 Discussion

The evaluation results in the previous section provide a ranking of competing techniques in terms of their performance. However, it does not answer why COSTER performs better than other techniques. We hypothesize that this is because of COSTER's ability to capture the fuller context of the FQNs of API elements. To provide further insights into this hypothesis, we conduct a set of studies and present their results in this section. All the experiments and analyses in this section are performed on GitHub Dataset.

4.6.1 Sensitivity Analysis: Impact of decision

This section validates our design decisions for building the model. Our local context consists of the top and the bottom four lines, including the line in which the API element is located. We select four lines because we observe that the precision becomes steady after considering more than four lines whereas the execution time increases exponentially. We conduct a set of studies to understand how the selection of tokens, different contexts, and similarity scores affect the performance of the technique. Our initial context C_0 contains tokens from the top four lines only. Next, we add the tokens of the bottom four lines with C_0 to create the context C_1 (i.e., local context). To understand the importance of using the global context, first, we incorporate those methods of the global context that are called on the receiver variable to create C_2, and then add the methods that use either the receiver variable or the API element as a method parameter to create C_3. Therefore, the context-wise categories are:

C_0: Context containing tokens from the top four lines.

C_1: C_0 + Tokens from the bottom four lines.

C_2: C_1 + Methods of global context that are called on the receiver variable.

C_3: C_2 + Methods of global context that use either the receiver variable or the API element as a method parameter.

COSTER considers three different similarity scores: likelihood score, context similarity score, and name similarity score. To understand the effect of those similarity scores, we train and test COSTER using different context settings (i.e., C_0, C_1, C_2, and C_3) using only the likelihood score. Next, we train and test COSTER by including the context similarity score and the name similarity score, one at a time. We record the precision, recall, and F_1 score after each run, as shown in Table 4.7.

Considering only top four lines of the local context, precision and recall values reach to 45.72% and 46.27% for the top-1 recommendation. Adding the bottom four lines of the local context also helps to improve the

Table 4.7: Precision (Prec.), Recall (Rec.) and F_1 score (F_1) of COSTER for considering different contexts and similarity scores

Models	**Description**	**Recc.**	**Prec.**	**Rec.**	$\boldsymbol{F_1}$
M_0	C_0+ Likelihood Score	Top-1	45.72	46.27	45.99
		Top-3	52.94	51.73	52.33
		Top-5	54.28	53.61	53.94
M_1	C_1+ Likelihood Score	Top-1	51.67	48.21	49.88
		Top-3	54.34	53.71	54.02
		Top-5	55.07	54.83	54.95
M_2	C_2+ Likelihood Score	Top-1	62.76	65.17	63.94
		Top-3	71.83	74.33	73.06
		Top-5	75.28	77.92	76.58
M_3	C_3+ Likelihood Score	Top-1	71.38	72.94	72.15
		Top-3	79.17	82.94	81.01
		Top-5	83.77	85.67	84.71
M_4	M_3+ Context Similarity	Top-1	85.67	86.19	85.93
		Top-3	90.82	92.08	91.45
		Top-5	94.33	95.17	94.75
M_5	M_4+ Name Similarity	Top-1	**89.48**	**90.04**	**89.76**
		Top-3	**92.11**	**93.26**	**92.68**
		Top-5	**95.43**	**95.84**	**95.63**

result, precision and recall values are increased by 5.95% and 1.94%, respectively. We also observe that the inclusion of the global context also has a positive impact on the performance. The precision and recall values reach to 71.38% and 72.94%, respectively for the top-1 recommendation. Context similarity score plays a more important role than the name similarity score. Adding the context similarity score increases precision and recall values to 85.67% and 86.19%, respectively. Finally, when we consider all the contexts and similarity scores we obtain the best result. The precision and recall values reach to 89.48% and 90.04% for the top-1 recommendation. We also observe similar effects when we consider top-3 and top-5 recommendations.

4.6.2 Effect of increasing the number of libraries

Increasing the number of libraries can have the following two effects. First, with the increase of libraries, the number of infrequent APIs also increases. Second, the likelihood of mapping the same API name to multiple FQNs in the training examples also increases. We were interested in examining how these affect the performance. Baker and COSTER can easily be adapted to an iterative experiment setting where we increase the number of libraries by adding one library at a time and record the performance at each step. However, we could not do so for StatType because the technique takes a considerable amount of time for training. Thus, we conduct the experiment by considering seven different number of libraries and record the performance of all three competing techniques for the top-1 recommendation at each number. Note that we apply the same 10-fold cross-validation technique to measure the performance.

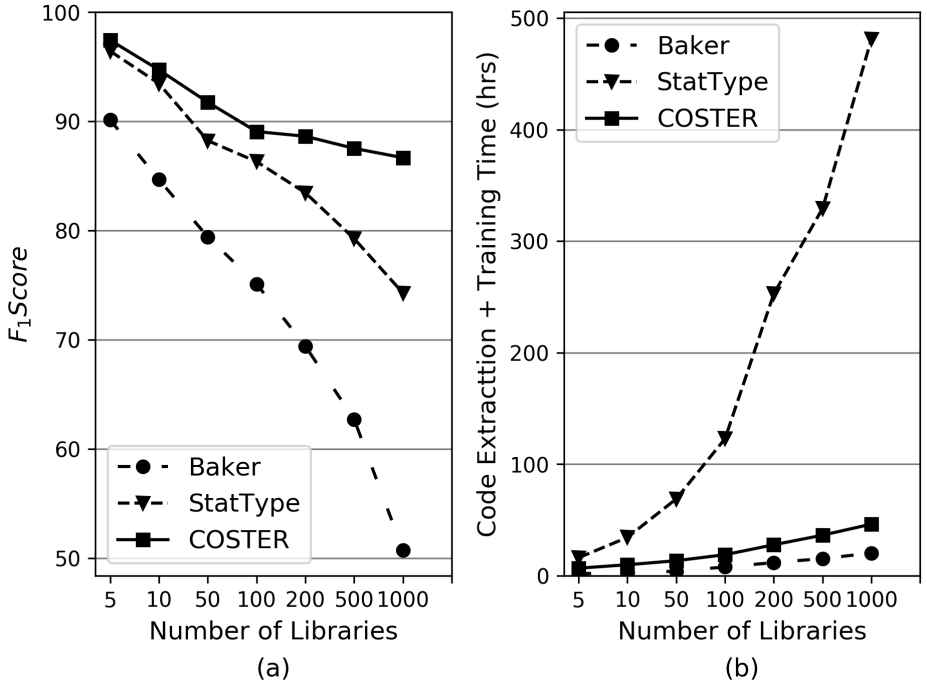


Figure 4.3: The effect of increasing the number of libraries on the (a) performance (i.e., F_1 score) and (b) Code extraction + Training time of Baker, StatType and COSTER

Fig. 4.3(a) shows the F_1 score of Baker, StatType and COSTER for different number of libraries. Among the three competing techniques, Baker performs relatively poorly where it has around 90% F_1 score for five libraries and the performance drops as we increase the number of libraries. The primary reason for such declination is that the more we increase the number of libraries, the more the API names are mapped to multiple FQNs. Thus, Baker fails to reduce the size of the candidate set into one for those multiple mapping cases. StatType and COSTER have similar F_1 score when the number of libraries is five. However, increasing the number of libraries affects the performance of StatType more than that of COSTER. Increasing the number of libraries also increases the number of APIs and many of those APIs lack a large number of

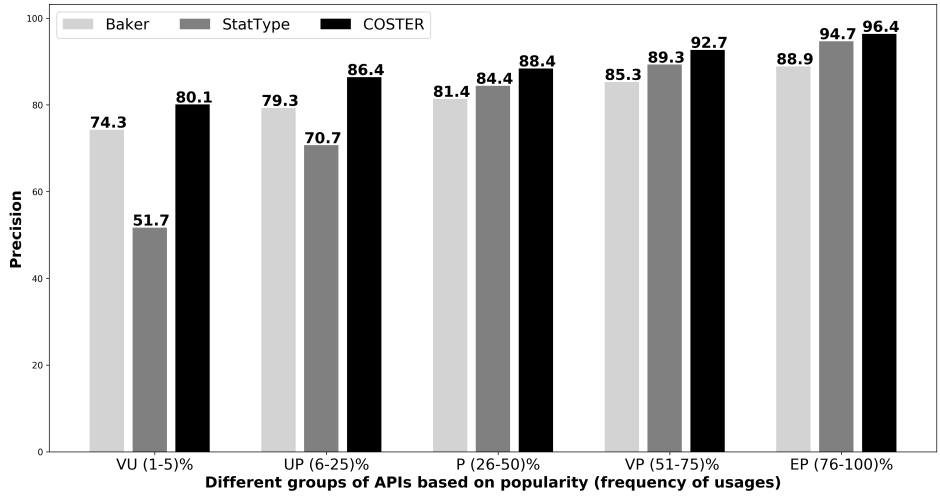
examples. This affects the performance of StatType. However, the performance of COSTER affects the least. This is possible because the technique considers different information sources to recommend FQNs of APIs and does not require large training examples to capture their usage patterns (discussed more in the next analysis).

With respect to timing, shown in Fig. 4.3(b), StatType has the worst outcome. With the increase of libraries, the number of examples also increases and the training time grows exponentially for StatType. Baker has the best performance since it does not require any training time. On the other hand, COSTER consumes twice more time than Baker and ten times less than StatType, and manages to maintain the highest F_1 score.

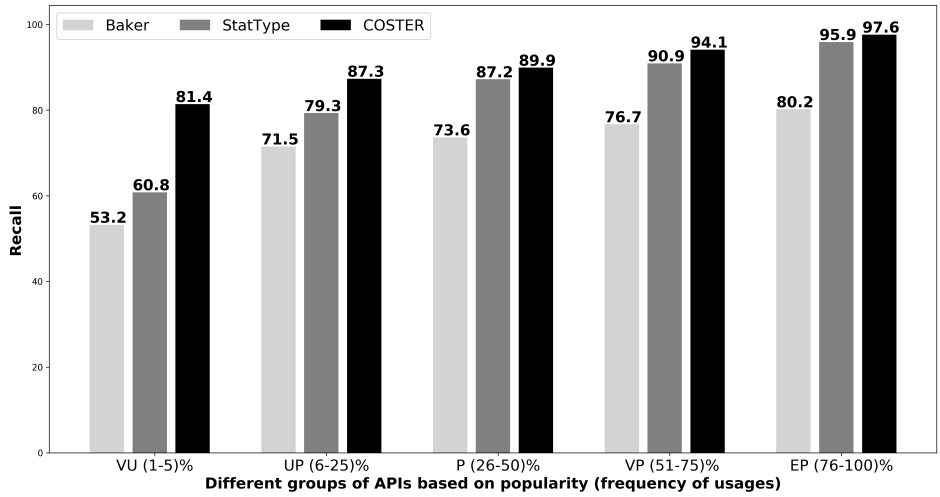
4.6.3 Impact on API popularity

This section investigates the relationship between the performance of recommending FQNs of APIs and popularity of those APIs. The popularity of an API is defined as the number of times that API is used in source code examples. We categorize APIs into five different groups based on their popularity or frequency of usages. The first group consists of APIs whose usage frequency is no more than 5% of all usages of APIs. We refer to this group as the very unpopular APIs (VU). The usage frequency of the second group of APIs ranges between 6-25%, referred to as the unpopular APIs (UP). The usage frequency of the next two groups ranges between 26-50% and 51-75%, and are called the popular (P) APIs and very popular (VP) APIs, respectively. Finally, APIs whose usage frequency is more than 75% of all API usages are referred to as the extremely popular (EP). We calculate the precision and recall for all five groups of APIs using the GitHub dataset.

From Fig. 4.4, we see that the performance of StatType and COSTER are very close for the extremely popular APIs. The difference is no more than 2% for both precision and recall. However, the performance difference becomes more significant as the popularity of APIs decreases. For example, for the popular APIs COSTER achieves 4-7% higher precision and 3-16% higher recall than the other two techniques. The difference becomes the highest for the very unpopular APIs, where COSTER is about 6-28% more accurate in terms of precision and recall compared with the other two techniques. Thus, among the three techniques we compared, API popularity affects the performance of COSTER the least. Moreover, for unpopular and very unpopular API categories, StatType obtains the worst precision values. For these APIs, StatType could not find enough examples in the training dataset and that affects the performance of the technique. We collected 30 examples of very unpopular APIs where StatType failed to produce the correct result and manually investigated them. We found that StatType returned FQNs in 16 cases which are nowhere close to the actual FQNs. This indicates that StatType cannot perform well in detecting FQNs of those APIs that are either unpopular or very unpopular. However, COSTER considers a rich set of information to form a context of an API and does not require a large number of examples for training. Statistically, the precision and recall of COSTER are significantly different than those of the compared techniques for API popularity analysis.



(a)



(b)

Figure 4.4: Comparing precision and recall of Baker, StatType and COSTER for API groups of different popularity.

4.6.4 Effect of receiver expression types

We categorized receiver expressions of API method or field calls based on their AST node types. We were interested in learning whether the performance of Baker, StatType, and COSTER vary across different receiver expression types.

Table 4.8 shows the performance of all three techniques across different receiver expression types. The second column of the table shows the percentage of test cases for each receiver expression type.

The simple name is the most popular expression type, followed by the qualified name and the method invocation. Around 95% of test cases belong to these three expression types. The difference in performance between COSTER and StatType for these expression types are small compared to other expression types.

Table 4.8: Precision (Prec.) and Recall(Rec.) of Baker, StatType and COSTER for different receiver expression types.

Expr. Type	Data(%)	Baker		StatType		COSTER	
		Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
Class Instance Creation	0.27	0	0	84.13	85.11	**89.43**	**91.53**
Array Access	0.28	76.14	78.34	85.43	87.76	**90.17**	**91.27**
Type Literal	0.34	66.34	72.43	86.73	87.11	**89.73**	**90.17**
String Literal	1.20	67.14	72.14	**98.34**	99.47	**98.34**	**99.71**
Simple Name	72.21	83.14	76.17	85.17	86.20	**90.43**	**91.83**
Qualified Name	16.21	80.73	78.59	86.74	89.43	**91.74**	**92.68**
Method Invocation	6.93	18.24	11.49	84.21	85.27	**84.91**	**86.72**
Field Access	1.11	64.14	75.18	87.34	87.66	**88.17**	**89.17**

The lack of code examples contributes to the difference between StatType and COSTER for other expression types (discussed in Section 4.6.3). For the three most frequent receiver expression types, the precision and recall of StatType range between 84-86% and 85-89%, respectively. In the case of COSTER, the precision and recall range between 85-91% and 86-92%, respectively. We investigated 50 incorrect predictions made by StatType which were correctly inferred by COSTER, and found that global context played the primary role for such difference. Due to the presence of global context, COSTER was able to find similar contexts from OLD and determined the correct FQNs. StatType considers only the last four tokens and was not able to determine FQNs of those cases. Baker performs very poorly compared to StatType and COSTER. Finally, results from two-tailed Wilcoxon signed-rank test [136] after adjusting p-values show that, statistically the precision and recall of COSTER are significantly different than Baker and StatType for all receiver expression types.

4.6.5 Multiple Mapping Cardinality Analysis

Name ambiguity poses a threat for resolving FQNs of APIs, as indicated by prior studies [28, 96, 108, 120]. Name ambiguity occurs when multiple classes, methods, or fields with the same name exist in different libraries or different packages of the same library. This section investigates COSTER's ability in resolving the name ambiguity for API elements with one or multiple FQN mapping candidates and compared the result with that of Baker and StatType. We use the term cardinality to refer to the number of FQN mapping candidates and the test cases are categorized based on different cardinality values. Next, we calculate the precision of Baker, StatType, and COSTER for those categories for the top-1 recommendations. We only consider precision because we cannot determine cardinality for missing cases. The first column of Table 4.9 shows different cardinality values. The second column shows the percentage of test cases for each cardinality value. The remaining three columns show the precision of Baker, StatType, and COSTER.

Table 4.9: Precision (for top-1 recommendation) of Baker, StatType and COSTER for multiple mapping cardinality analysis

Cardinality	Data (%)	Baker	StatType	COSTER
1	46.72	**100**	**100**	**100**
<3	16.54	91.73	92.61	**96.73**
<10	11.46	84.36	90.43	**92.47**
<20	7.30	78.98	88.81	**91.26**
<50	6.34	68.51	86.72	**90.72**
<100	4.50	62.43	85.12	**89.43**
<500	3.68	54.72	84.73	**89.02**
<1K	2.92	43.57	84.28	**88.43**
1K+	0.53	19.76	83.52	**88.17**

Table 4.9 shows that 46.7% of total test APIs have only one mapping candidate. Therefore, for around half of the test cases, the techniques do not need to deal with ambiguities. According to Table 4.9, COSTER can solve all single mapping cases successfully similar to Baker and StatType. With the increase of cardinality, the precision decreases to 19.7% for Baker. In the case of StatType, the precision drops from 100% to 83.5% as we increase the cardinality. However, the performance of COSTER affects the least among all three competing techniques. The precision of COSTER drops from 100% to 88.17% when cardinality value changes from 1 to 1K+. Statistically, the precision of COSTER is significantly different than both Baker and StatType.

4.6.6 Limitation

Despite having the best results for all of the experiments, COSTER has some limitations that are discussed in this section.

First, if an API element contains multiple method calls, COSTER often fails to resolve the FQN of the last method call. For example, consider the method call statement at Listing 4.1, COSTER was able to detect the FQN of the first two method calls but failed for the last method call (i.e., toString). However, such cases are very rare (0.0004%).

```
DownloadManager.getInstance().getDownloadsListFiltered().toString()
```

Listing 4.1: An example [113] where the proposed technique failed to

Second, Stack Overflow code fragments can be very short, which can even contain only one line. In such cases, COSTER finds very few to no context at all and fails to resolve FQNs of API elements. However, we investigated 20 such cases and found that 16 of them can be solved by reading the posts. That gives us a

future research direction of resolving FQNs of API elements leveraging textual content of SO posts. That can be combined with the current implementation of COSTER.

Finally, similar to StatType, out-of-vocabulary issue also affects the recall of our technique. However, our proposed technique received a high accuracy by considering code examples collected from open-source software repositories.

4.7 Threats to Validity

This section discusses threats to the validity of this study.

First, we considered 100 most frequently used libraries of the GitHub dataset in this study. One can argue that the result may not generalize to other frameworks or libraries. However, all these libraries are popular and have been actively used by developers. we also observed that increasing the number of libraries affected the performance of COSTER the least (see Section 4.6.2). Thus, our results should largely carry forward.

Second, the accuracy of our proposed technique can be affected by the ability to correctly find API usages in Stack Overflow code snippets. To mitigate this issue, each code snippet was analyzed by two different annotators. When there were ambiguities, they talked to each other to resolve the issue. However, such cases were very rare.

Third, the process of making Stack Overflow code compilable by the annotators can be erroneous by importing incorrect import statements for code compilation. However, we validated the random selection of those compilable Java source files manually, and they did not find any such errors.

Finally, we consider the likelihood score, cosine [77] based context similarity score and Levenshtein distance [63] based name similarity score. Other similarity measures could give us different results. However, those similarity measures that we selected are widely used and we are confident with the results.

4.8 Related Work

One closely related work to ours is that of Baker [120]. For each API element name, the technique builds a candidate list and shorten after each iteration based on the scoping rules and a set of relationships. The process continues until all elements are resolved or the technique cannot shorten those lists further. Our work is also closely related to StatType [96]. The technique uses the type and resolution context of API elements and statistical machine translation to infer FQNs of API elements. However, we capture long-distance relations of an API Element through global context along with local context and reduces search space step by step. Thus, COSTER performs better than both Baker and StatType with lesser training time than StatType (see Section 4.5.3).

Another related work is Partial program analysis (PPA) [26]. The technique leverages a set of heuristics to identify the declared type of expressions. PPA can be an inclusion of a technique rather than being standalone for resolving API names. For example, RecoDoc [28] uses PPA [26] to link between code elements and their

documentation. However, 47% of libraries in the Maven repository do not contain any documentation [99]. ACE [108] is another technique that works on SO posts, analyzes texts around the code and links them. ACE involves text to code linking rather than code to code linking, and thus not suited for evaluation.

Techniques have been developed that focus on type resolution for dynamically typed languages, such as JavaScript (JS) and Python [42, 69, 105, 120, 126]. JSNice [105] uses conditional random fields to perform a joint prediction of type annotation for JavaScript variables. DeepTyper [42] leverages a neural machine translation to provide type suggestions for JS code whereas NLP2Type [69] uses a deep neural network to infer the function and its parameter from docstring. Tran et al. [126] recognize the variable name from minified JS, and the work of Xu et al. [140] resolves Python's variable by applying probabilistic method on multiple sources of information. However, the primary challenge and application of these techniques are different than ours. An interesting research direction can be combining any of these techniques with our solution and examine the effect for dynamically typed languages.

A number of studies in the literature trace the links between source code and documentation using various approaches. These include but are not limited to topic modelling [11, 83], Latent Semantic Indexing [29, 72], text mining [12], feature location [65], Vector Space Model [130, 146], classifier [56], Structure-oriented Information Retrieval [74,112], ranking based learning [142] and deep learning [59]. However, these techniques primarily focus on documentation, bug reports, and email content whereas we focus on linking between code elements.

4.9 Summary

This study explores an important and non-trivial problem of finding FQNs of API elements in the code snippets. We propose a context-sensitive technique, called COSTER that collects local and global contexts for each API element, calculate the occurrence likelihood score and store the collected usage contexts, likelihood scores and FQNs of API elements in the occurrence likelihood dictionary (OLD). Using the likelihood score along with context and name similarity scores, the proposed technique resolves FQNs of API elements. Comparing COSTER with two other state-of-the-art techniques show that our proposed technique improves precision by 4-6% and recall by 3-22% along with an improvement of training time by a factor of ten. Experiments on the effect the number of libraries, API popularity, receiver expression types, multiple mapping cardinality, and sensitivity analysis elaborates why COSTER performs better than Baker and StatType. The compared techniques claim that their approaches will work for the dynamically typed programming languages such as JavaScript. However, there is a lack of study of validating such a claim. In the next chapter (Chapter 5), we thus explore the performance of the techniques build for the statically typed programming languages such as Java for the dynamically typed programming languages. Additionally, we propose a technique that captures the localness of the dynamically typed code and infer the types of the code element.

5 Exploring Type Inference Techniques of Dynamically Typed Languages

Our previous study (Chapter 4) explores how to resolve the API elements of the online forum code snippets for statically typed programming languages such as Java. While exploring the problem, we find that the techniques developed for Java are declared to be adaptable for the dynamically typed programming languages such as JavaScript without having any empirical analysis. Moreover, languages such as JavaScript can have better usability with typed annotations. Thus in this chapter, we explore the effectiveness of techniques build for Java in the case of JavaScript. Additionally, we propose a technique that can infer the type of JavaScript code elements using local properties of the code.

The rest of the chapter is organized as follows. Section 5.1 introduce the study along with the proposed technique. Section 5.2 presents a motivating example for the study, Section 5.3 presents prior studies related to our work. Section 5.4 introduces dataset and explains the evaluation procedure followed by answer of three research questions in Section 5.5, 5.6 and 5.7. Section 5.8 provides further insights of the comparison results. The key findindgs of the study is summarized in Section 5.9. We discuss threats to the validity of our work in section 5.10 and Section 5.11 summarizes the chapter.

5.1 Introduction

Dynamically typed programming languages enable developers to write less verbose code by removing the burden of specifying types in code, thus support quick prototyping. Such dynamic type systems allow language designers to avoid spending considerable time developing a type system to ensure the completeness of the program at compile time. However, the development and usages of TypeScript[1], Flow[2] and Closure[3] indicate that leading software companies are now considering static typing as an important part of developing code. Recent research results also show the benefits of static typing. For example, Gao et al. [33] find that adding type annotations in JavaScript can help to avoid 15% of the reported bugs. Prior studies [41, 104] also show that static type systems help in understanding undocumented code, fixing type issues and solving semantic errors, thus have a positive impact on the maintenance of software. Finally, building code completion tools also requires type information. For example, method completion tools remove irrelevant method names

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

based on the type of receiver variable [10]. The lack of type information thus makes the dynamically typed languages difficult to provide precise completion proposals. Therefore, it is important to add type information to dynamically typed languages.

Existing type inference techniques that learn from code examples can be divided into two broad categories. First, there are techniques [28,96,108,113,120] that are designed to resolve the types of online code snippets. These techniques are developed and tested for the code snippets written in statically typed languages (such as Java). While the language is statically typed, those online code snippets often do not contain type declarations [120]. This makes it difficult to determine which libraries need to be imported to compile the code correctly. Techniques such as StatType [96] and COSTER [113] fall under this category. For example, StatType [96] uses statistical machine translation whereas COSTER [113] leverages a combination of three similarity measures for inferring types. While both techniques achieve high precision and recall, they are only tested for code snippets written in Java. However, it is not clear whether such techniques can provide similar performance for dynamically typed languages, such as JavaScript.

The other group of techniques are specifically designed and tested for dynamically typed programming languages (i.e., JavaScript). For example, Raychev et al. [105] developed the technique JSNice that predicts the type of a code element based on the types of the surrounding code elements that are connected with the target code element through the dependency graph. Malik et al. [69] developed a technique, called NL2Type, that leverages JSDoc, comments, the formal signatures of the functions, and a recurrent neural network model to infer the types of the functions and its parameters. DeepTyper [42] uses a neural machine translation-based approach to infer the types of JavaScript code elements. Type inference techniques for JavaScript receive significant performance gain from using a deep learning technique. However, a problem with deep learning techniques is that they require considerable training time. While training often considers a one-time operation, supporting a new library or adding more training examples require retraining the model. This motivates us to investigate the type inference of JavaScript further.

In this paper, we first conduct an empirical study to understand how the type inference techniques developed for the statically typed language (i.e, Java) perform for the dynamically typed language (i.e, JavaScript). To answer the question, we apply two state-of-the-art techniques developed for Java to predict the types of JavaScript code elements. While doing the empirical study, we find code elements specific to a type usage are closely located to it, also known as locally specific. Thus, we investigate how to capture such localness property more accurately. We use a combination of word embedding, context similarity, and local model to infer the types of code elements. Finally, we compare the proposed technique with not only the techniques developed for Java but also the deep learning based type inference techniques developed for JavaScript. Thus, our study is based on the following research questions.

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

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

(a) JavaScript Code

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

(b) Typescript Code

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

Figure 5.1: A motivational example [114]

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

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

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

Findings from the study are aligned with prior studies [31, 43, 54, 76], indicating that our proposed technique can significantly reduce the training time with comparable performance.

Furthermore, analysis of sensitivity, overlapping and number of training examples justify the findings. Thus, we make the following three contributions.

1. Conduct an empirical study to evaluate the performance of type inference techniques that are developed for Java code snippets for JavaScript code snippets.

2. Propose a technique that uses local specific code tokens as context, Word2Vec, context similarity, and a local model to infer the types of the code elements of JavaScript.

3. Conduct a comparative study of the proposed technique with the deep learning-based state-of-the-art techniques along with the extensive analysis of the result.

5.2 Motivational Example

Let us consider a JavaScript function as shown in Fig. 5.1 that takes any physical body *imposter*, three dimensional *force* and *point* as parameters, and applies impulse on the *imposter* by the *force* at that *point*.

TypeScript enables optional types to be added to the JavaScript code. While developers may benefit from such type annotations of code elements (see Fig. 5.1 (b)), annotating an existing codebase is a time

consuming operation, requires expertise and often can be erroneous [42]. An automated technique that can learn from existing type annotations of codebases and can recommend types of JavaScript code elements as a developer types code can be useful in this case.

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

Finally, the function does not have any JSDocs or line comments. Thus, techniques that depend on JSDoc and line comments, such as NL2Type [69], will not not work in this case. On the other hand, since the function is very small in size, COSTER [113] will not be able to collect any code tokens outside the top and bottom four lines. This can impact the performance of the technique.

5.3 Related Study

5.3.1 Empirical studies on type inference

A number of studies explore the usefulness of type inference in dynamically typed languages. Hackett and Guo [40] analyze JavaScript snippets and show that a type inference engine can increase the performance of different functionalities of a website by 50%. Pradel et al. [97] analyze scripts in the runtime, find inconsistencies of types, report them as bugs. Gao et al. [33] investigate the TypeScript[1] and Flow[2] for detecting buggy code and find that around 15% of bugs can be detectable by both engines. Ray et al. [104] conduct a large scale empirical study on GitHub projects and find that statically typed languages are less defect prone than dynamically typed languages. The above works either examine buggy type annotations or the importance of a type inference engine whereas we focus on investigating type inference techniques of a dynamically typed language (i.e., JavaScript)

5.3.2 Type inference in statically typed languages

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

Linking code from text based techniques use documentation [11, 28, 72], bug reports [29, 83], emails [12] and posts from online Q&A sites [108] to find appropriate types in Java code snippets. However, these approaches suffer from accuracy due to the lack of documentation [99] and informal nature of bug reports and posts [113].

Baker [120] is the first to link code from the other code tokens situated within the same scope and uses an iterative deducing technique to infer types of code elements. However, the technique fails to infer types of a number of code elements due to strict scope rules [96, 113]. StatType [96] uses the original code fragment

as the source language and type resolved code fragment as the target language. It leverages a statistical machine translation technique to find mapping between them. The technique performs poorly for the types having a lesser number of examples [113]. COSTER [113] is another technique that can infer types of code elements in Java code snippets based on the type usage contexts and three different similarity measures (i.e, occurrence likelihood, context similarity, and name similarity scores). In our RQ1, we find that COSTER is not applicable for dynamically typed languages (such as JavaScript) since it cannot capture the differences in the structure of languages.

5.3.3 Type inference in dynamically typed languages

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

TypeScript[1] developed by Microsoft and Flow[2] developed by Facebook are the type annotation based solutions. However, developers need to manually annotate the type information, which requires considerable time and effort [42].

Program analysis-based type inference techniques are largely formal and static rule based [1, 3, 7, 8, 32, 52, 62, 67, 123]. Such approaches are unable to perform well for statistically uncomputable functions such as *eval* [40, 69].

JSNice [105] is the first work on probabilistic type inference that constructs a dependency network between the code elements of known properties and unknown properties. The unknown node of the dependency network is predicted using the conditional random field. DeepTyper [42] is a neural machine translation-based technique that considers the stream of code tokens as the source language and the stream of types as the target language. Based on the bidirectional Recurrent Neural Network (RNN), the technique learns the mapping between the source and target languages. The technique then infers types based on the mapping. NL2Type [69] is another deep learning-based type inference technique for JavaScript that focuses on the parameters and return types of functions. The technique creates contexts based on the JSDoc, comments and the formal signatures of methods. Those natural texts are preprocessed and passed through a Word2Vec and a bidirectional Long Short Term Memory based neural network to learn the nonlinear relations. Both NL2Type and DeepTyper outperform JSNice whereas we develop a technique that significantly reduce the training time without sacrificing the accuracy (see Section 5.7).

5.4 Experimental Design

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

Table 5.1: Dataset Overview

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

5.4.1 Dataset Description

For evaluating the performance of type inference techniques, we use the dataset developed by Hellendoorn et al. [42]. The dataset consists of the top 1000 open source JavaScript projects selected based on the star count whose code elements are annotated by developers using TypeScript (ts). We successfully retrieve 774 projects in September 2019 out of those 1000 projects. The rest of the projects are either deleted or made private. Thus, we cannot include them in our study. Table 5.1 shows an overview of the dataset we used for our experiments. The dataset contains more than 100K files. All projects are parsed using the TypeScript compiler[1]. The compiler returns a type for each variable, class object, literal, function's return type, and parameter. The type of a code token represents an instance in our dataset.

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

5.4.2 Evaluation Procedure

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

$$Precision = \frac{recommendations\ made \cap relevant}{recommendations\ made} \tag{5.1}$$

$$Recall = \frac{recommendations\ made \cap relevant}{recommendations\ requested} \tag{5.2}$$

$$F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{5.3}$$

Here, *recommendations requested* refers to the total number of instances in the test set. *Recommendations made* is the number of instances for which a technique infers types. *Recommendations requested* is the number of instances required inference. We use a two-tailed Wilcoxon signed-rank test [136] to determine whether the difference between the performance of two compared techniques are statistically significant or not. For each evaluation metric (i.e., precision, recall and F_1 score), we collect the result of each competing technique

for each fold as one data point and compare ten data points obtained from ten different folds with that of the compared technique.

5.5 RQ1: How do the techniques developed to infer types in Java code snippets perform for JavaScript code snippets?

5.5.1 Motivation

Techniques that infer types in online Java code snippets showed great performance [96, 113]. Prior studies of those techniques also argued that such techniques can easily be adapted to dynamically typed programming languages (such as JavaScript). We are interested in learning the effectiveness of those techniques to detect types in JavaScript code snippets. Results from the study can help us to decide whether we can reuse those techniques for JavaScript code snippets or further modification is required.

5.5.2 Approach

We choose two state-of-the-art techniques, StatType [96] and COSTER [113], that are developed to detect types in code snippets written in Java language. We made necessary changes to adapt those techniques for JavaScript language. COSTER collects both local and global contexts to capture the type usage context of code elements. The technique collects any types, keywords, function calls and operators that appear within the top and bottom four lines as the local context. The global context consists of methods outside of the local contexts that are invoked on the receiver variable and that use the code element or the receiver variable as the parameter. To adapt the technique for JavaScript, we leverage the TypeScript compiler to collect both contexts. We then leverage a combination of three different similarity measures to predict types of code elements. In case of StatType, we collect the stream of the resolved code elements as the source language and the stream of types of the code elements as the target language. For the target language, similar to COSTER, we collect the type of code elements such as variables, class objects, literals, function's return type and parameter. Next, we use the same Phrasal [37] tool used by the authors to train and test the statistical machine translation model. We then collect the precision, recall and F_1 score (F_1) for top-1, top-3 and top-5 recommendations. We summarize the results in Table 5.2.

5.5.3 Evaluation

Both StatType and COSTER did not perform well for detecting types in JavaScript files, as shown in Table 5.2. For the top-1 recommendation, the precision and recall of StatType are 28.69% and 25.73%, respectively. Performance improves as we consider more recommendations but can only be considered as mediocre. For example, the precision and recall reach to 49.36% and 47.28% for the top-5 recommendations,respectively. COSTER performs comparatively worse than the StatType. The precision and recall

Table 5.2: Performance comparison of statically typed language based techniques StatType and COSTER for JavaScript snippet

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

of COSTER are 9.9-16.75% and 11.07-18.87%, respectively, lower than StatType for all recommendations. However, both techniques performed remarkably well when applied for code snippets written in Java [96,113].

5.5.4 Discussion

To understand the reasons that contributed to such poor performance, we determined the differences between Java and JavaScript languages considering the length of methods, identifiers, and the contexts (i.e., local and global context) collected by COSTER. We used the GitHub dataset of COSTER [113] for Java and our dataset for JavaScript. For both datasets, we consider the number of lines as the length of a method, the number of characters as the length of an identifier, and the number of tokens as the length of the local and global contexts. Fig. 5.2 summarizes the results. The reasons behind COSTER's poor performance can be derived as follows.

First, the global context does not play a significant role in JavaScript. This is because functions in JavaScript are typically small in lengths compared to that of Java. Therefore, globally related tokens are difficult to find in JavaScript. While the median function length of Java is 6 lines, the number drops to 3 for JavaScript, as shown in Figure 5.2(a). Moreover, the median length of global context in Java is 2.5 times higher than that of JavaScript (see Figure 5.2(d)). However, the differences drops significantly when we compare the length of local contexts between Java and JavaScript datasets, as shown in Figure 5.2(c)). The median values are 9 and 10 respectively. Thus, COSTER has a hard time collecting globally related tokens for JavaScript comparing to Java and fails to show similar performance.

Second, the name of identifiers in JavaScript is comparatively shorter than that of Java, as shown in Figure 5.2(b). While the median identifier length in JavaScript is 2 , the value reaches to 4 for Java. Thus the name similarity measure in COSTER do not perform well when tested for JavaScript code snippets.

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

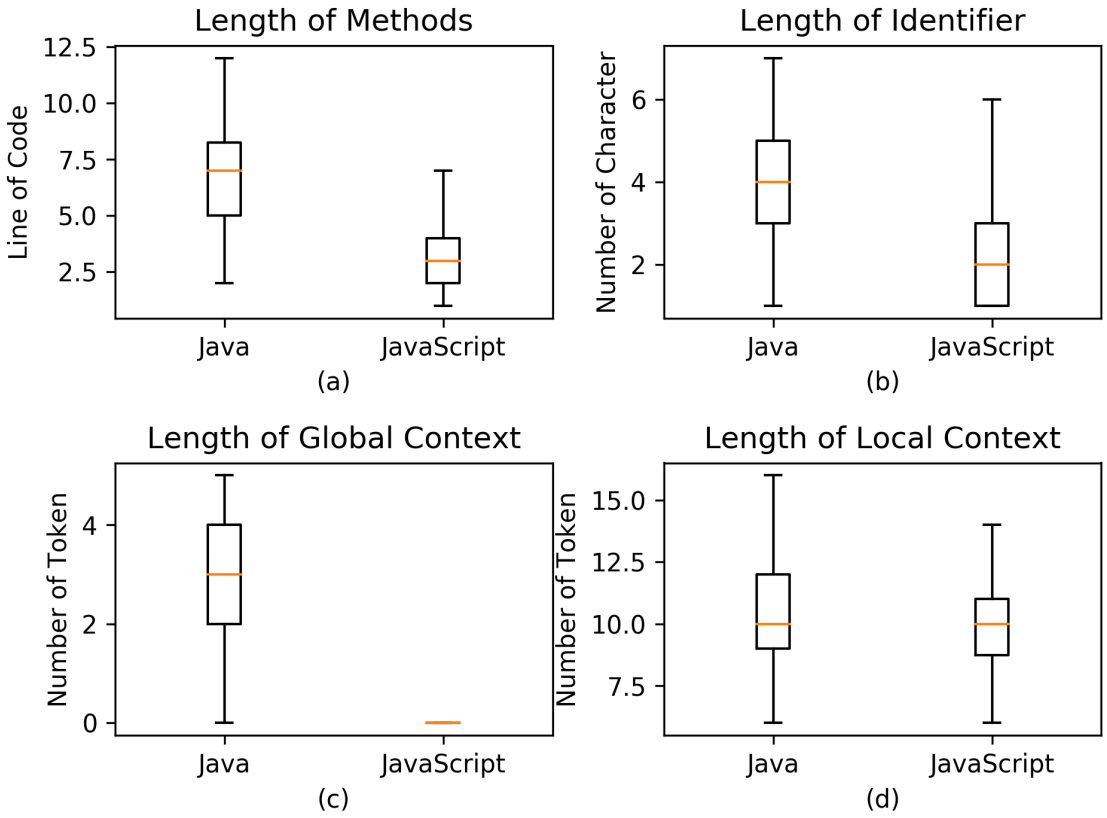


Figure 5.2: Length of method in terms of line of code and identifier in terms of number of character of Java and JavaScript

JavaScript code snippets. All these motivate us to investigate the problem further.

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

5.6.1 Motivation

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

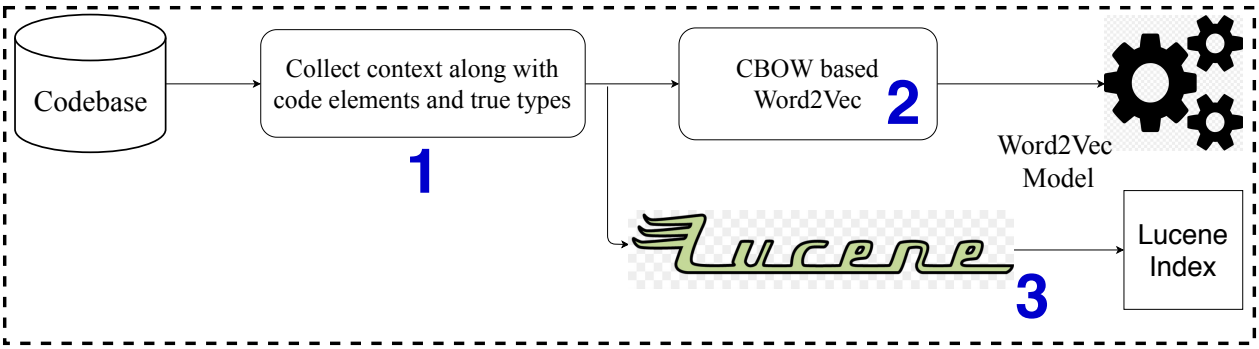


Figure 5.3: Overview of the training step of the proposed technique

5.6.2 Technique Description

In this section, we describe our proposed technique that infers the types of JavaScript code elements. The steps of the technique are discussed below.

Collect Type Usage Context

We parse each JavaScript file using the TypeScript compiler, create the Abstract Syntax Tree (AST) and collect the type usage context for each variable, class object, literal, function's return type and parameter. We refer to them as the code element unless specified otherwise. The type usage context of a code element consists of tokens that include the types of identifiers, keywords, function calls, class objects, and operators within the top and bottom four lines of that code element. Our selection of four lines is based on the fact that we obtain the best result using this setting. For example, the context for class object *impulse* at line 3 would be $function$, any, $Vector3$, $Vector3$, var, $Vector3$, $=$, new, $BJSCANNON$, $Vec3$, $Vector3.x$, $Vector3.y$, $Vector3.z$, var, $=$, new, $BJSCANNON$, $Vec3$, $Vector3.x$, $Vector3.y$, $Vector3.z$, $applyImpulse$, $Vector3$, $Vector3$. The primary motivation for choosing such a context is two-folded. First, the context contains locally specific code tokens which are inspired by the principle of naturalness [45] and localness [128] properties of the source code. Second, we consider the type information of code elements (i.e., identifiers) rather than their lexical information. Such context showed good performance in prior studies [42, 96].

We use an inverted index structure to organize type usage contexts along with their associated types. Each type usage context appears as a document and tokens of those documents are used to index those sets of documents. Such an index structure allows us to quickly retrieve types whose usage context matches with that of a query context. Instead of implementing the inverted index structure from the scratch, we use the implementation available in the Lucene search engine [18].

Training models

Next, we apply the Continuous Bag of Word (CBOW) architecture of Word2Vec [78] technique on the training dataset (2 in Fig. 5.3). Word2Vec [78] is a word embedding technique that takes words/tokens from

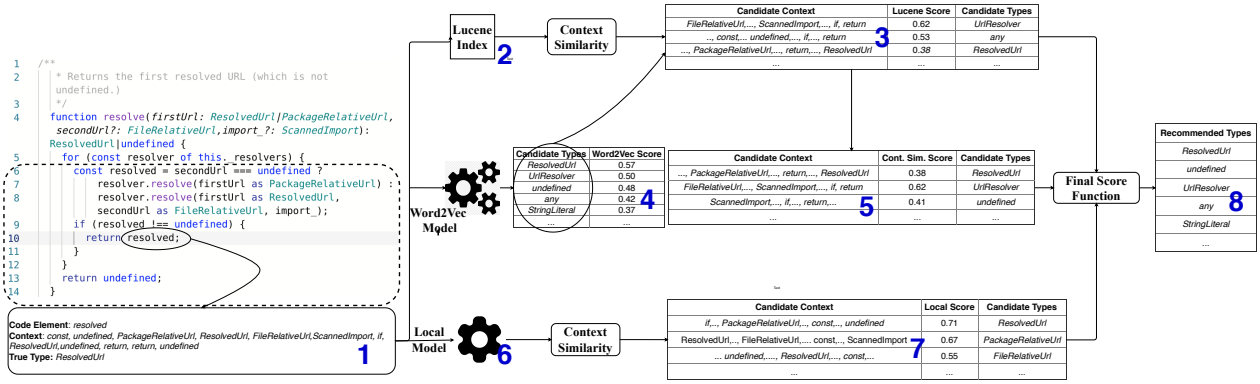


Figure 5.4: Overview of the inference steps with an example [114] of the proposed technique

the training contexts as input and creates a d-dimensional continuous vector space. Each word/token is then represented by a vector in such a way that, if plotted in a vector space, semantically similar words/tokens appear close to each other [111]. There are two ways to create such a word vector. The first one is the Continuous Bag of Word (CBOW) architecture where words/tokens are embedded into the vectors based on their context. The other one is the Skim-gram architecture where the context is embedded based on the word/token. We used the former one since it considers the whole context as one observation and predicts the type based on the context during the inference step. For example, if contexts having tokens such as *Identifier* and *ImportKeyword* are found frequently for the type *String*, the Word2Vec model will learn that context with these tokens are very closely related to the type *String*. During inference, if any context with such tokens are found, the model will predict the type *String* with a higher probability value.

Inferring Types

To infer the type of a code element, we follow the following sequence of steps. First, our proposed technique collects the type usage context of a target code element that can be a variable, a class object, a literal, return type of a function or a parameter. We use the term query element to refer to the target code element, the associated type usage context as the query context and the actual type of the code element as the true type. For our example shown in Fig. 5.4, code element *resolved* at line 10 is the query code element, *ResolvedUrl* is the true type and the code tokens within 6-14 formulate the query context.

Second, we pass the query context (C_q) to the Inverted Index File (2 in Fig. 5.4) and it returns a ranked list of all contexts with associated types that are stored during the training time. We then collect the top 500 usage contexts and their associated types. We use the term candidate context to refer to any element of the list of contexts returned by the search engine. The types associated with those contexts are referred to as the candidate types. Next, we calculate the context similarity score between the query and each candidate context. To do that, we apply the Cosine [77] similarity method that use Eqn. 5.4 to calculate the context similarity ($Sim_{con}(C_q, C_{ci})$) between query context (C_q) and each candidate context (C_{ci}).

$$Sim_{con}(C_q, C_{ci}) = \frac{N_o}{N_m} \times cosine(C_q, C_{ci}) \tag{5.4}$$

Here, N_o is the number of tokens of the candidate context that appear in the same order of that of query context, and $N_{matched}$ is the number of tokens that are matched. The equation returns a score between 0 to 1, signifying the similarity between the query and each candidate context. In our example in Fig. 5.4, the candidate context of type $URLResolver$ has the highest context similarity score of 0.62 followed by the candidate context of type *any* and $ResolvedUrl$.

Third, we present the query context to the previously trained Word2Vec model. Since the Word2Vec model is learned using the entire training dataset, we refer to this as a global model. The model returns a score for each type that represents how similar the type is given the query context. We pick top-k types based on the Word2Vec score (4 in Fig. 5.4) and refine the list of candidate contexts based on these types (5 in Fig. 5.4). In Fig. 5.4, Word2Vec score of $ResolvedUrl$ becomes the highest followed by $UrlResolver$, $undefined$, and so on.

Fourth, we present the query context to the local model (6 in Fig. 5.4). By local model we mean the contexts inferred so far within the project associated by their types. To create the local model, we save the context and the top-1 recommendation result after each inference as long as we remain on the same project. If we consider that the given JavaScript snippet in Fig. 5.4 is the only code in the project, then the local model for our example will consists of the contexts of all code elements before line 10 that are inferred so far with their top-1 types. Contexts of local models are referred to as candidate local contexts and the top-1 types are referred to as candidate local types. The motivation behind considering the local model is two-folded. First, the global model can be biased by the types that have a very large number of examples, such as *any*. The local model can help to skip such bias by capturing the project specific similarities. Second, the use of a local model along with a global model is found effective in literature [43]. We calculate the context similarity between the query context and each candidate local context using Eqn. 5.4 (7 in Fig. 5.4). In our example, candidate local types such as $ResolvedUrl$ are found having the highest local score followed by $PackageRelativeUrl$, $FileRelativeUrl$ and so on.

Finally, we sort the list of candidate types based on three scores: Word2Vec score, context similarity score and local score (8 in Fig. 5.4). Word2Vec score ($Sim_{word2vec}\ (T_{ci}, C_q)$) dictates how similar the ith candidate type (T_{ci}) is with the query context (C_q). Context similarity score ($Sim_{con}\ (C_q, C_{ci})$) tells how similar the ith candidate context (C_{ci}) is with the query context (C_q) and the local context similarity score ($Sim_{local}\ (C_q, C_{lci})$) captures the localness tendency of the query context (C_q) with respect to the ith local candidate context (C_{lci}). We use the Eqn. 5.5 to calculate the score of the ith candidate type ($Score(T_{ci})$):

$$
\begin{aligned}
Score(T_{ci}) = \alpha \times Simword2vec\ (T_{ci}, C_q) \\
+ \beta \times Simcon\ (C_q, C_{ci}) \\
+ \gamma \times Simlocal\ (C_q, C_{ci})
\end{aligned} \tag{5.5}
$$

Here, α, β and γ are the coefficients of Word2Vec, context similarity, and local context similarity scores, respectively. The coefficients are tuned using Hill Climbing Adaptive Learning algorithm [121]. The sorted list of candidate types is considered as the top-k recommendations of the proposed technique. We calculate the precision(Prec.), recall (Rec.) and F_1 score (F_1) for top-1, top-3 and top-5 recommendation for the compared techniques, as shown in Table 5.3.

Table 5.3: Performance comparison of StatType, COSTER and the proposed technique

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

5.6.3 Evaluation

Our proposed technique outperforms both StatType and COSTER with a big margin, as shown in Table 5.3. The precision of the proposed technique for the top-1 recommendation is 24.45% higher than StatType and 35.77% higher than COSTER. The recall of the proposed technique is 20.48% and 33.37% higher than that of StatType and COSTER respectively. The differences increase as we increase the number of recommendations. In case of the precision, the difference increases from 24.45 to 29.88% for StatType and 35.8 to 46.63% for COSTER when we increase the number of recommendation from 1 to 5. Similarly recall increases from 20.48 to 26.63% for StatType and 33.37 to 45.1% for COSTER. The differences of performances are also statistically significant.

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

5.7.1 Motivation

Despite having promising results in different problems, prior studies [31, 43, 54, 76] show that deep learning techniques are not always the best solution for different software engineering problems. Two state-of-the-art deep learning-based type inference techniques are developed for JavaScript programs [42, 69]. This section compares our proposed technique with those two state-of-the-art techniques to understand the importance of using deep learning techniques.

5.7.2 Approach

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

5.7.3 Evaluation

DeepTyper and our proposed technique can detect types of variables, class objects, literals, function's return types and parameters. However, NL2Type focuses on detecting only the function's parameters and return type. Thus, we compare our technique with DeepTyper for all code elements. We calculate the precision (Prec.), recall (Rec.) and F_1 score (F_1). The results of DeepTyper and our proposed technique are shown in Table 5.4.

As shown in Table 5.4, DeepTyper has higher precision than our proposed technique. However, the difference is not much significant, ranges between 1-1.5%. However, our technique achieves better recall and F_1 scores than the DeepTyper. The differences become more noticeable as we increase the number of recommendations. For example, our technique achieves 5.27% higher recall values than the DeepTyper for the top-5 recommendations. All differences are statistically significant. We applied Choen's d [24] to measure

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

Table 5.4: Performance comparison of DeepTyper and the proposed technique for all code elements

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

the effect size. We found that the effect size is negligible (0.1) for precision. Next, we collect the results of the function's return type and their parameters for the proposed technique and DeepTyper. We compare the results with that of NL2Type.

Table 5.5: Performance comparison of DeepTyper, NL2Type and the proposed technique for function's return type and their parameters

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

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

5.7.4 Discussion

In our evaluation, we see that the proposed technique is very competitive with the state-of-the-art deep learning-based type inference techniques. However, deep learning techniques require a significant amount of time and memory. Therefore, we are interested to compare the time and memory requirement with DeepTyper and NL2Type. We consider the time required for extracting code tokens, training the technique and inferring

types. To compare the memory requirements, we consider two different aspects. First, we calculate the size of models and indexes. Second, we calculate both the random access memory (RAM) and the video random access memory (VRAM) usages. The time required to parse the JavaScript code into the desired dataset is the code extraction time, time to train the neural network or collecting Word2Vec and inverted index file is the training time and the time to infer a code element is the inference time. The size of the neural network or the word2vec model is the model size, the size of the inverted index file is the index size, and the amount of CPU memory consumed while training is the RAM consumption and the amount of GPU memory consumed while training is the VRAM consumption. All the results are shown in Table 5.6. For fair comparison we used the same server having 12 CPUs of Intel Xeon processor with 2.10 GHz processing speed each, 32 GB of memory and NVIDIA Tesla K20c with 4GB of memory.

Table 5.6: Time and memory comparison of Proposed Technique(Pro. Tech.), DeepTyper and NL2Type

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

Our proposed technique requires the lowest amount of memory and time, as shown in Table 5.6. More importantly, the proposed technique does not require any GPU support. The compared techniques are time and memory efficient when GPU is provided. However, the proposed technique is 5-7 times faster when all the techniques are executed with the help of GPU and 10-14 times faster when run without GPU in case of training. The difference in training time can have a good impact, if a user wants to complete training on a cloud server. For example, to complete the training in Amazon EC2 instances[7], the proposed technique will need 20-30 USD whereas DeepTyper needs 100-150 USD based on GPU or CPU instance, and NL2Type needs 80-120 USD. The above results clearly show that the proposed technique is faster than the compared techniques. In case of memory requirements, the proposed technique takes 17-30% less memory to store the model, 1.5 to 3 time less RAM consumption than the compared techniques. Additionally, the proposed technique does not require any VRAM. Thus, the proposed technique outperforms all the compared techniques in terms of memory requirements.

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

5.8 Discussion

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

5.8.1 Sensitivity Analysis

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

Considering only tokens in the top four lines and using Lucene search engine, we obtain 21.43% of precision and 11.86% of recall for the top-1 position, respectively. Next, we combine Lucene with tokens from the top and bottom four lines. This helps to improve precision and recall by 2.95% and 4.57%, respectively. The inclusion of the context similarity score improves both precision and recall that reach to 31.83% and 22.51%, respectively. The inclusion of the Word2Vec with the previous model helps to obtain 42.18% precision and 29.62% recall. Finally, we obtain the best result when we consider all sources of information and all similarity measures. The precision and the recall reach to 53.14%, and 46.21% for the top-1 recommendation. We also observe a similar scenario for the top-3 and top-5 recommendations. Results from our study show that the ranked list of types generated by only using the Lucene search engine is not much effective. The reason we use Lucene is to quickly find a list of types whose usage contexts match with the query context. We employ additional sources of information to further refine and rank that list of types..

5.8.2 Analysis of overlapping

We are interested in learning how different type inference techniques complement each other. We consider DeepTyper, NL2Type and our proposed technique for this analysis. For this study, we consider the top-1

Table 5.7: Sensitivity Analysis

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

recommendation and we refer to a test example as a data point. The overlapping of correctly predicted types between DeepTyper and our proposed technique is shown in Figure 5.5a. Since NL2Type only focus on predicting the type of function's parameters and the return type, we use Figure 5.5b to show how many of the correctly predicted types overlap between NL2Type and our proposed technique.

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

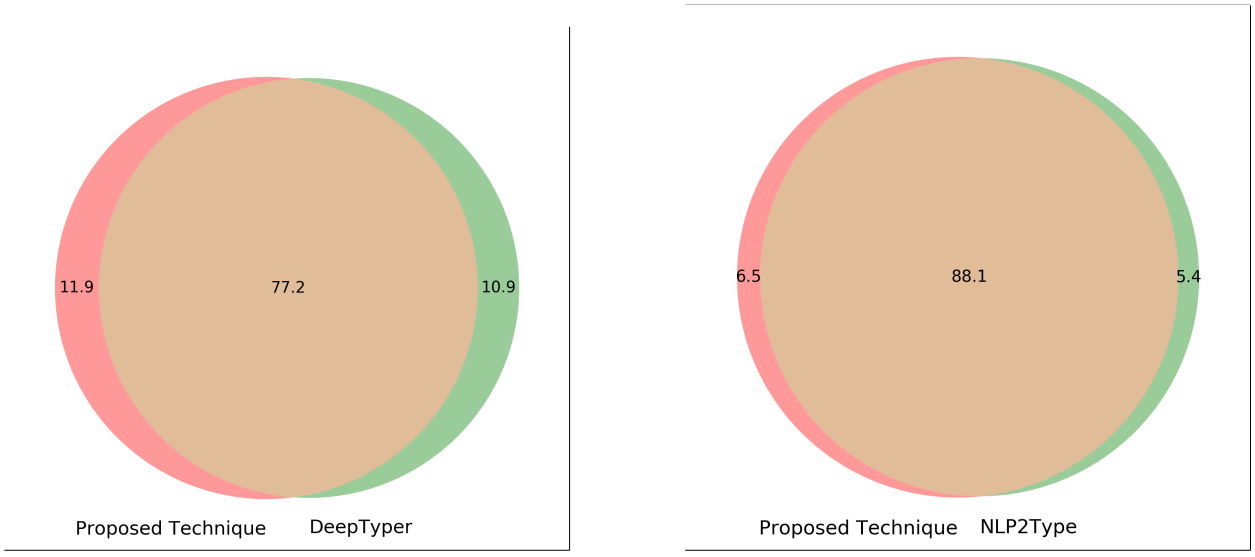


Figure 5.5: Overlapping analysis of the instances correctly predicted in top-1 by the proposed technique, DeepTyper, NL2Type

5.8.3 Effect of the number of training examples

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

The recall of our proposed technique is higher than the DeepTyper for all different groups of types by 5-12%, as shown in Table 5.8. Our proposed technique has lower precision in the case of *any*, very popular and extremely popular types. However, the differences are very small, ranges between 0.5-2.5%. On the

Table 5.8: Effect of number of training examples

Type(% of data)	DeepTyper			Proposed Technqiue		
	Prec.	Rec.	F_1	Prec.	Rec.	F_1
any (47%)	**63.24**	51.27	56.63	62.75	**58.29**	**60.44**
VU (1-5)%	18.43	9.43	12.48	**24.83**	**18.62**	**21.28**
UP (6-25)%	22.19	11.49	15.14	**29.17**	**22.43**	**25.36**
P (26-50)%	26.14	14.23	18.43	**32.18**	**25.73**	**28.60**
VP (51-75)%	**44.76**	30.75	36.46	43.35	**36.82**	**39.82**
EP (76-100)%	**69.95**	56.18	62.31	67.42	**61.82**	**64.50**

contrary, our proposed technique has 6.40%, 6.98%, and 6.04% higher precision than DeepTyper for the very unpopular, popular and less popular types, respectively.

5.8.4 Limitations

This section discusses the limitations of our proposed technique.

First, the TypeScript compiler returns *any* type if it cannot bind the type properly. We observe in some cases our technique disagrees with the compiler by returning types other than the *any*. However the type returned from our technique in some of these cases found to be more appropriate. For example, the TypeScript compiler returns *any* as the type of the code token *isValidPrivKey* (see line 2) as shown in Fig. 5.6 whereas our technique finds it as a function that returns a *boolean* value.

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

Figure 5.6: An example where TypeScript compiler extract wrong type [114].

Second, 2.4% of functions in our dataset contain only one line of code. Our technique fails to detect types of code elements in those cases because those code elements have very little or no prior context to collect. For example, the proposed technique fails to infer the types of *arr* and *i* as shown in Fig. 5.7. One possible solution is to consider the documentation in those cases to address such issues. However, such cases are not very common.

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

Figure 5.7: An example where the proposed technique failed to infer types [114].

5.9 Key Findings

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

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

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

(c) Applications of deep learning techniques need to be carefully justified: Prior studies [31,43,54,76] show that deep learning techniques may not be the best choice for software engineering problems. Our study also support their findings. The deep neural network can find a nonlinear relation between the data. However, such a relationship may not always exist in the data. Furthermore, deep learning techniques are computationally more expensive than traditional machine learning techniques. It is thus important to apply alternative techniques first to justify the need for deep learning techniques. Future research should explain the need for deep learning techniques for the problem first.

5.10 Threats to validity

This section discusses threats to the validity of this study.

First, the dataset we used in this study is created by collecting JavaScript projects from GitHub. One can argue that our findings may not generalize to a different dataset. However, we would like to point to the fact

that we considered a large number of projects in our study. All these projects are active in the development and have a long development history. Thus, our results should largely carry forward.

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

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

5.11 Summary

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

6 Conclusion

6.1 Concluding Remarks

Developers rely on Application Programming Interface (API) usages to fasten the development time as well as the effort of writing those code from scratch [10,20,113,144]. However such libraries and APIs need to be learned in order to use them effectively. To learn the APIs, developers follow either documentation or code completion feature of IDEs or the online question answer forums [10,27,96].

Studies [99,120] shows that documentations are not always available and they contain a lot of text. Thus learning APIs using documentation remain impractical. Singer [20] also found that developers prefer code examples than the documentation to learn the APIs. Code completion feature in the IDEs helps the developer to learn the usages of the APIs within the code editor. Thus the feature is found to be one of the top ten used commands by the developers [81]. However, the code completion feature in IDEs is very trivial and is not a great help to the developers. A number of method name [10, 84, 98], argument [9, 66, 144] and code completion [45, 86, 89, 106, 128] techniques are proposed in the literature. However, none of them is suitable for the method call along with the argument completion. Finally, learning through online forums is hindered due to the code snippets are being nonexecutable [96, 113, 120]. To alleviate the challenges of the online forums code snippets, several studies [28, 96, 108, 120] are conducted. However, the studies limit themselves either within the documentation or locally specific code tokens. Moreover, the techniques designed statically typed programming language are very unlikely to work for dynamically typed programming languages.

In our first study, we formulate the method call along with the argument completion problem as a sequence to sequence learning task. We captured the lexical, syntactic and semantic contexts of a method call as the sequence of input. The lexical context is defined as the code tokens within the top four lines of the method call, syntactic context is the AST information of the previous lines of code and semantic context is the type information of the identifier present in the previous lines of the code. Such representation of code is motivated by the prior studies [86, 129] that show better performance of the machine learning models when represented in different ways for Software Engineering problems. The contexts are passed through an attention mechanism based neural encoder-decoder model that returns a list of method names along with arguments as the completion suggestions. Evaluation results with six state-of-the-art method name, argument, code completion and program synthesis techniques for ten large subject systems and three frameworks show that the proposed technique, **DAMCA** achieves 5-25% more accuracy and 10-30% more MRR than the compared techniques for both inter-cross project settings.

Next, we explore the techniques that resolve the Fully Qualified Name (FQN) of the API element for the online forum code. We found that the techniques consider only the code element within the scope that causes poor performance. Moreover, the techniques behave poorly for the APIs that have a small number of examples in the training dataset. Thus we propose a novel context-sensitive technique, **COSTER** that considers code elements within the top and bottom four lines (local context), as well as the method calls related to the API elements (global context) as the context. The candidate FQNs for a given API element is generated based on the common contexts shared by the API elements and the examples stored at Occurrence Likelihood Dictionary (OLD). Finally, the candidate list is sorted based on likelihood, context similarity, and name similarity scores. While evaluating the proposed techniques with two state-of-the-art techniques for 600K code examples collected from Github and Stack Overflow, we find that the proposed technique improves the precision by 4-6% and recall by 3-22% with one-tenth reduced training time than the compared techniques. We find the answer to why the proposed technique outperforms the compared techniques while analyzing the results using sensitivity, an increasing number of the library, API popularity, receiver expression types, and mapping cardinality analyses.

While conducting the previous study, we find that the techniques developed for the type inference of statically typed programming language (e.g, Java) claim that they will work for dynamically typed programming language (e.g, JavaScript) without any empirical evaluation. Thus, we investigate the techniques developed for Java with the 25 million code examples of JavaScript collected from the Github. The investigation results suggest that the techniques lose more than 50% of the accuracy when applied for the JavaScript examples. While analyzing the result we find the locally specific code tokens in case of dynamically typed programming languages have more semantic similarity than the lexical or syntactic similarities. Thus we proposed a technique in our Study 3 that leverages the locally specific code tokens of a code element and utilizes the Word2Vec, context similarity as the global models, and previous type inference outputs from the same project as a local model to infer the type of the code elements. The combination of global and local models helps to improve the result by 20-47% than the techniques developed for Java. We also observe that there are some deep learning based techniques [42, 69] for the type inference of JavaScript. When we compare the proposed technique with these techniques, we find the proposed technique performs 5-14 times faster than the deep learning techniques without sacrificing accuracy. In this study too we find the answer to better performance of the proposed technique than the compared techniques when analyze the results using overlapping and number of training examples analyses.

6.2 Future Works

While in this thesis, we explore different items as context while solving different recommendation tasks. In the future, we like to explore more problems that use context for both statically and dynamically typed programming languages. This section discussed our plans with the research works in this thesis.

Deep Method Argument Recommendation: In this study, we explore sequence to sequence learning for solving the argument completion along with the method name. Our study focuses primarily on Java code snippets. However, the technique can be explored for the same problem in other languages such as C#, JavaScript, Python and so on. Moreover, the sequence to sequence technique would be explored to other software engineering problems that include other forms of code completion, bug localization, code search and so on. Lastly, few neural network-based learning mechanisms such as Memory Network [133], Neural Turing Machine [36], Generative adversarial networks [35] and so on are proposed in recent times. We will like to explore such learning mechanisms in our future studies.

Context Sensitive Type Inference for Java: We use different types of context while resolving the API elements in this study. There are several software engineering problems such as code completion, code search, bug detection where contexts are needed to be considered. In the future, we will explore both local and global contexts in the above-mentioned problems. Moreover, we are planning to use string similarity functions such as string search, word embedding and so on with our current implementation. These methods are well established and used in a lot of software engineering problems successfully. Lastly, we like to incorporate text to code linking techniques with COSTER. In our limitation, we stated that, for a very small code snippet, COSTER is ineffective. In those cases, text to code linking techniques can be a good choice. So we will establish a mechanism to combine the text to code linking techniques with COSTER.

Type Inference technique for dynamically typed programming language: In this study, we found that the inference techniques developed for statically typed programming language (i.e. Java) perform poorly for dynamically typed programming language (i.e, JavaScript). There are a number of studies where the techniques are developed for statically programming languages. In the future, we will explore the effectiveness of these techniques for dynamically typed programming language. Furthermore, one of our key ideas states that developing different techniques for different languages can not be the solution. Therefore, our future research direction would be developing techniques that will be effective for all languages. Lastly, we find that deep learning techniques, DeepTyper, NLP2Type can be replaced by a simpler, fast and white box technique. In the future, we will analyze deep learning techniques applied to different software engineering problems and whether they can be replaced by simpler techniques or not.

References

[1] Ole Agesen. Constraint-based type inference and parametric polymorphism. In *Proceedings of the 2nd International Static Analysis Symposium (SAS)*, pages 78–100, 1994.

[2] Mikel Aickin and Helen Gensler. Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods. *American journal of public health*, 86(5):726–728, 1996.

[3] Alexander Aiken and Brian Murphy. Static type inference in a dynamically typed language. In *Proceedings of the 18th Symposium on Principles of Programming Languages (POPL)*, pages 279–290, 1991.

[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 281–293, 2014.

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 38–49, 2015.

[6] Ben Allison, David Guthrie, and Louise Guthrie. Another look at the data sparsity problem. In *Proceedings of the 9th International Conference on Text, Speech and Dialogue (TSD)*, pages 327–334, 2006.

[7] Jong-hoon David An, Avik Chaudhuri, Jeffrey S Foster, and Michael Hicks. *Dynamic inference of static types for Ruby.* ACM, 2011.

[8] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 428–452, 2005.

[9] Muhammad Asaduzzaman, Chanchal K Roy, Samiul Monir, and Kevin A Schneider. Exploring api method parameter recommendations. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 271–280, 2015.

[10] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. Cscc: Simple, efficient, context sensitive code completion. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 71–80, 2014.

[11] Hazeline U. Asuncion and Richard N. Asuncion, Arthur U.and Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 95–104, 2010.

[12] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 375–384, 2010.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[14] Colin Bannard and Chris Callison-Burch. Paraphrasing with bilingual parallel corpora. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics (ACL)*, pages 597–604, 2005.

[15] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proceedings of the 28th International Conference on Advances in Neural Information Processing Systems (NIPS)*, pages 1171–1179, 2015.

[16] Yoshua Bengio. Neural net language models. *Scholarpedia*, 3(1):3881, 2008.

[17] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003.

[18] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *Proceedings of the SIGIR Workshop on Open Source Information Retrieval*, page 17, 2012.

[19] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pages 513–522. ACM, 2010.

[20] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 213–222, 2009.

[21] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected api subgraph via text phrases. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE)*, page 10, 2012.

[22] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[23] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, October 2014.

[24] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[25] Nick Craswell. Mean reciprocal rank. *Encyclopedia of Database Systems*, pages 1703–1703, 2009.

[26] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, volume 43, pages 313–328, 2008.

[27] Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE)*, pages 127–136, 2010.

[28] Barthélémy Dagenais and Martin P Robillard. Recovering traceability links between an api and its learning resources. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 47–57, 2012.

[29] A. De Lucia, R. Oliveto, and G. Tortora. Adams re-trace. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 839–842, 2008.

[30] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th Symposium on Principles of Programming Languages (POPL)*, pages 599–612, 2017.

[31] Wei Fu and Tim Menzies. Easy over hard: A case study on deep learning. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 49–60, 2017.

[32] Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 24th Symposium on Applied Computing (SAC)*, pages 1859–1866, 2009.

[33] Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 758–769, 2017.

[34] Yoav Goldberg. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, 10(1):1–309, 2017.

[35] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[36] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[37] Spence Green, Daniel Cer, and Christopher Manning. Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of the 9th Workshop on Statistical Machine Translation (WMT)*, pages 114–121, 2014.

[38] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 933–944, 2018.

[39] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*, pages 631–642, 2016.

[40] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd International Conference on Programming Language Design and Implementation (PLDI)*, pages 239–250, 2012.

[41] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014.

[42] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 26th Joint Meeting on the Foundations of Software Engineering (FSE)*, pages 152–162, 2018.

[43] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 763–773, 2017.

[44] R. Hill and J. Rideout. Automatic method completion. In *Proceedings of the 19th International Conference on Automated Software Engineering (ASE)*, pages 228–235, 2004.

[45] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 837–847, 2012.

[46] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[47] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[48] Reid Holmes, Rylan Cottrell, Robert J Walker, and Jorg Denzinger. The end-to-end use of source code examples: An exploratory study. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM)*, pages 555–558. IEEE, 2009.

[49] Eric Horton and Chris Parnin. Gistable: Evaluating the executability of code snippets on the web. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME)*, pages 217–227, 2018.

[50] Daqing Hou and David M Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 26–30, 2010.

[51] Daqing Hou and David M Pletcher. An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion. In *Proceedings of the 27th International Conference on Software Maintenance and Evolution (ICSME)*, pages 233–242, 2011.

[52] Simon Holm Jensen et al. Type analysis for javascript. sas, volume 5673 of lecture notes in computer science, 2009.

[53] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, pages 279–289, 2013.

[54] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, 2019.

[55] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 664–675, 2014.

[56] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *Transactions on Software Engineering*, 39(11):1597—1610, 2013.

[57] Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Proceedings of the International Conference on the Association for Machine Translation in the Americas (AMTA)*, pages 115–124, 2004.

[58] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1137–1145, 1995.

[59] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 476–481, 2015.

[60] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 476–481, 2015.

[61] Raymond Lau, Ronald Rosenfeld, and Salim Roukos. Trigger-based language models: A maximum entropy approach. In *Proceedings of the 19th International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 45–48, 1993.

[62] Benjamin S Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Tejas: retrofitting type systems for javascript. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS)*, pages 1–16, 2013.

[63] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics-doklady*, pages 707–710, 1966.

[64] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, 2017.

[65] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, pages 234–243, 2007.

[66] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 1063–1073, 2016.

[67] Francesco Logozzo and Herman Venter. Rata: rapid atomic type analysis by abstract interpretation–application to javascript optimization. In *Proceedings of the 19th International Conference on Compiler Construction (CC)*, pages 66–83, 2010.

[68] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270, 2015.

[69] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 304–315, 2019.

[70] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of 26the International Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, 2005.

[71] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

[72] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 125–135, 2003.

[73] Pedro Martins, Rohan Achar, and Cristina V Lopes. 50k-c: A dataset of compilable, and compiled, java projects. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2018.

[74] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering (ASE)*, pages 111–120, 2011.

[75] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 111–120, 2011.

[76] Tim Menzies, Suvodeep Majumder, Nikhila Balaji, Katie Brey, and Wei Fu. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pages 554–563, 2018.

[77] Rada Mihalcea, Courtney Corley, Carlo Strapparava, et al. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 775–780, 2006.

[78] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[79] Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *Transactions on Software Engineering and Methodology*, 26(1):3, 2017.

[80] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

[81] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *Software*, 23(4):76–83, 2006.

[82] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 25–34, 2012.

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

[84] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*, pages 511–522, 2016.

[85] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, volume 1, pages 858–868, 2015.

[86] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N Nguyen. A deep neural network language model with contexts for source code. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 323–334, 2018.

[87] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 69–79, 2012.

[88] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. Recommending api usages for mobile apps with hidden markov model. In *Proceedings of the 30th International Conference on Automated Software Engineering(ASE)*, pages 795–800, 2015.

[89] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 532–542, 2013.

[90] Franz Josef Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL)*, pages 160–167, 2003.

[91] Takayuki Omori, Hiroaki Kuwabara, and Katsuhisa Maruyama. A study on repetitiveness of code completion operations. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 584–587, 2012.

[92] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 311–318, 2002.

[93] Chris Parnin and Christoph Treude. Measuring api documentation on the web. In *Proceedings of the 2nd international workshop on Web 2.0 for software engineering (Web2SE)*, pages 25–30, 2011.

[94] Chris Parnin and Christoph Treude. Measuring api documentation on the web. In *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering (Web2SE)*, pages 25–30, 2011.

[95] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. *Georgia Institute of Technology, Tech. Report*, 11(6):11p, 2012.

[96] Hung Phan, Hoan Nguyen, Ngoc Tran, Linh Truong, Anh Nguyen, and Tien Nguyen. Statistical learning of api fully qualified names in code snippets of online forums. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 632–642, 2018.

[97] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 314–324, 2015.

[98] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *Transactions on Software Engineering and Methodology*, 25(1):3, 2015.

[99] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th International Conference on Mining Software Repositories (MSR)*, pages 221–224, 2013.

[100] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 357–367, 2016.

[101] Mohammad Masudur Rahman and Chanchal Roy. Nlp2api: Query reformulation for code search using crowdsourced knowledge and extra-large data analytics. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME)*, pages 714–714, 2018.

[102] Mohammad Masudur Rahman and Chanchal K Roy. Quickar: automatic query reformulation for concept location using crowdsourced knowledge. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 220–225, 2016.

[103] Mohammad Masudur Rahman, Chanchal K Roy, and David Lo. Rack: Automatic api recommendation using crowdsourced knowledge. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 349–359, 2016.

[104] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 155–165, 2014.

[105] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages (POPL)*, volume 50, pages 111–124, 2015.

[106] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th International Conference on Programming Language Design and Implementation (PLDI)*, pages 419–428, 2014.

[107] D Raj Reddy et al. Speech understanding systems: A summary of results of the five-year research effort. department of computer science, 1977.

[108] Peter C Rigby and Martin P Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 832–841, 2013.

[109] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[110] Martin P Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[111] Xin Rong. Word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[112] M. Saha, R. K.and Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013.

[113] C M Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy. Learning from examples to find fully qualified names of api elements in code snippets. In *Proceedings of the 34th International Conference on Automated Software Engineering (ASE)*, pages 243–254, 2019.

[114] C M Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy. Exploring type inference techniques ofdynamically typed languages. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, page 11p, 2020.

[115] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[116] Lifeng Shang, Zhengdong Lu, and Hang Li. Neural responding machine for short-text conversation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, pages 1577–1586, 2015.

[117] Chengxun Shu and Hongyu Zhang. Neural programming by example. In *Proceedings of the 32nd International Conference on Artificial Intelligence (AAAI)*, 2017.

[118] Janice Singer. Practices of software maintenance. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM)*, pages 139–145, 1998.

[119] Rishabh Singh and Pushmeet Kohli. Ap: artificial programming. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*, 2017.

[120] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 643–652, 2014.

[121] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 253–262, 2011.

[122] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.

[123] Peter Thiemann. Towards a type system for analyzing javascript programs. In *Proceedings of the 15th European Symposium On Programming (ESOP)*, pages 408–422, 2005.

[124] Yuan Tian, David Lo, and Julia Lawall. Automated construction of a software-specific word similarity database. In *Proceedings of the 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 44–53, 2014.

[125] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering (ICSE)*, pages 303–314. ACM, 2018.

[126] Hieu Tran, Ngoc Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Recovering variable names for minified code with usage contexts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 1–11, 2019.

[127] Christoph Treude and Martin P Robillard. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 392–403, 2016.

[128] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 269–280, 2014.

[129] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. Deep learning similarities from different representations of source code. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, 2018.

[130] Lo D. Wang, S. and J. Lawall. Compositional vector space models for improved bug localization. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 171–180, 2014.

[131] Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 297–308, 2016.

[132] Wei Wang and Michael W Godfrey. Detecting api usage obstacles: A study of ios and android developer questions. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 61–64, 2013.

[133] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[134] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 87–98, 2016.

[135] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR)*, pages 334–345, 2015.

[136] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[137] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[138] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. Learning to synthesize. In *Proceedings of the 4th International Workshop on Genetic Improvement (GI)*, pages 37–44, 2018.

[139] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 2048–2057, 2015.

[140] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 24th Joint Meeting on the Foundations of Software Engineering (FSE)*, pages 607–618, 2016.

[141] D. Yang, A. Hussain, and C. V. Lopes. From query to usable code: an analysis of stack overflow code snippets. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 391–402, 2016.

[142] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 689–699, 2014.

[143] Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 404–415, 2016.

[144] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 826–836, 2012.

[145] Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, pages 141–151. ACM, 2018.

[146] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.