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

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

***Abstract*—Developers often reuse code snippets from online forums, such as Stack Overflow, to learn API usages of software frameworks or libraries. These code snippets often contain ambiguous undeclared external references. Such external references make it difficult to learn and use those APIs correctly. In particular, reusing code snippets containing such ambiguous undeclared external references requires significant manual efforts and expertise to resolve them. Manually resolving fully qualified names (FQN) of API elements is a non-trivial task. In this paper, we propose a novel context-sensitive technique, called COSTER, to resolve FQNs of API elements in such code snippets. The proposed technique collects locally specific source code elements as well as globally related tokens as the context of FQNs, calculates likelihood scores, and builds an occurrence likelihood dictionary (OLD). Given an API element as a query, COSTER captures the context 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 significantly reduces the training time compared to the StatType, a state-of-the-art technique, without sacrificing accuracy. Extensive analyses on results demonstrate the robustness of the proposed technique.**

***Index Terms*—API usages, Context sensitive technique, Recommendation system, Fully Qualified Name**

I. 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 However, inadequate and outdated documentation of APIs hinder the learning process [1], [2]. As a result, developers favor code examples over documentation [3]. 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 [4]. However, such usage examples can suffer from external reference and declaration ambiguities when one attempts to compile them [2], [5]. External reference ambiguity occurs due to missing external references, whereas declaration ambiguity is 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 [6] only 1% of the Java and C# code examples included in the Stack Overflow posts are compilable. Yang et al. [7] 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) [8], text analysis [2], [9], and iterative deductive analysis [5]. All these techniques except Baker [5], need adequate documentation or discussion in online forums. However, 47% of the APIs do not have any documentation [10] and such APIs cannot be resolved by those techniques. Baker [5] 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 [11]. Recently, Statistical Machine Translation (SMT) is used to determine FQNs of APIs in StatType [11]. 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 paper, 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 [5] and StatType [11], using more than 600K code snippets from GitHub [12] and two different Stack Overflow (SO) datasets. We not only reuse the SO dataset prepared by Phan et al. [11] 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

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

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 paper 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 to answer why the proposed technique outperforms others.

The rest of the paper is organized as follows. Section II presents a motivating example and explains the challenges in resolving FQNs of API elements. We describe our proposed technique in Section III. Section IV introduces datasets and explains the evaluation procedure. Section V evaluates COSTER against two other state-of-the-art techniques and Section VI provides further insights on the performance of our proposed technique. We discuss threats to the validity of our work in section VII and Section VIII presents prior studies related to our work. Finally, Section IX concludes the paper.

II. 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
```

Fig. 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 [2], [5], [13], or statistical machine translation [11]. Rule-based systems (such as RecoDoc [2] and ACE [13]) search documentation or discussion to resolve the types. However, they have three limitations: documentation is rarely available [10], discussions are usually informal [5] and using Partial Program Analysis [8] results in partially qualified names [11]. Baker [5] 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 [5]. 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 [11] 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 [14]–[16]. 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*}.

[3]https://stackoverflow.com/questions/20157996/

[4]https://docs.oracle.com/javase/8/

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 [17] and localness [18] 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.

III. 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. 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 II, 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.

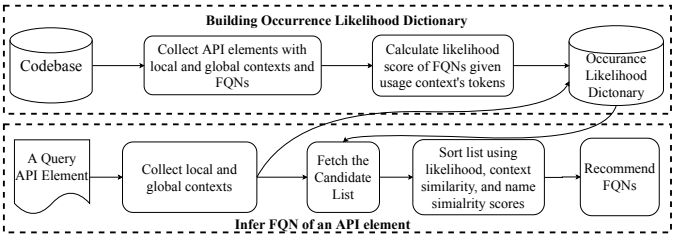


Fig. 2. Overview of COSTER's entire process of building OLD and recommending FQN of a query API element

We describe each of these steps in detail as follows.

A. 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 [19]. 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 [19], we estimate the likelihood of the FQN f_p given the token t appeared 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.

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

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:

$$ls\ (p, t) = ls\ (f_p|t) \quad \frac{w_{weight}}{distance(p, t) + k} \tag{2}$$

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

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

TABLE I
EXAMPLE CODE SNIPPET 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*
...	...	...	...	...	...

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. 3.

$$\log(ls(f_p|T)) \ \ ' \ \ \log(ls\ (p,T)) \\ ' \ \ \log(ls(p,t_1)) + \log(ls(p,t_2)) + ... + \log(ls(p,t_n)) \quad (3)$$

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.

B. Inferring FQN of an API element

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

1) 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 I. 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 I 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.

2) 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 I 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.

3) 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 [20] 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:

$$Sim_{context}(T_q, T_{ci}) = \frac{N_{order}}{N_{matched}} \quad \frac{T_q \ \ T_{ci}}{jjT_qjj\ jjT_{ci}jj} \quad (4)$$

In Eq. 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 I, the column *Context similarity*

shows the similarity score between the query context and each candidate context.

4) 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 [21] 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:

$$Sim_{name}(p_q, f_{ci}) = \begin{cases} 1 & \text{if len>lev} \\ \frac{lev(p_q, f_{ci})}{len(f_{ci})} & \text{otherwise} \\ 0 & \end{cases} \quad (5)$$

In Eq. 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 I 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.

5) Recommending top-k FQNs: The candidate FQNs are sorted in descending order of the similarity scores calculated using Eq. 6.

$$\begin{aligned} candidate\ score(f_{ci}) = \alpha \quad & ls\ (f_{ci} | T_{ci}) \\ + \beta \quad & Sim_{context}(T_q, T_{ci}) \\ + \gamma \quad & Sim_{name}(p_q, f_{ci}) \end{aligned} \quad (6)$$

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 [22] over the training data. For our example in Table I, the final score for *java.util.Queue*, *java.lang.String*, *java.util.List* and *java.util.LinkedList* based on Eq. 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.

IV. Evaluation

This section compares COSTER with two state-of-the-art techniques, Baker [5] and StatType [11]. 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 [12]?
- **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.

TABLE II
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

A. 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 II.

GitHub Dataset: We consider a collection of 50K Java projects collected from GitHub, called 50K-C projects [12]. We use the term GitHub Dataset to refer to the dataset as shown in Table II. 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 [11]. 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 [11], 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 II.

B. 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.

$$Precision = \frac{recommendations\ made \setminus relevant}{recommendations\ made} \quad (7)$$

$$Recall = \frac{recommendations\ made \setminus relevant}{recommendations\ requested} \quad (8)$$

$$F_1 = \frac{2 \quad Precision \quad Recall}{Precision + Recall} \quad (9)$$

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 [23] 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 [24]. 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.

V. Experimental Result and Analysis

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

A. *RQ1: Intrinsic Accuracy. How accurate is COSTER in identifying FQNs of API elements in Java source code snippets collected from Github dataset [12]?*

Table III 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 III only shows the top-1 recommendation for Baker since the technique only recommends the best match.

TABLE III
Precision (Prec.), Recall(Rec.) and F_1 score (F_1) of all competing for GitHub dataset [12]

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**

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, 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.

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

Table IV 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 [11], we only report results for StatType and COSTER.

Interestingly, as we see from Table IV 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

TABLE IV
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

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%.

TABLE V
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**

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

C. *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 VI. 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.

TABLE VI
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

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 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 VI show that our proposed technique is capable to exhibit the best performance (reported in Table III), 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.

VI. 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

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

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.

A. 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 VII.

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 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.

TABLE VII
PRECISION (PREC.), RECALL (REC.) AND F_1 SCORE (F_1) OF COSTER FOR CONSIDERING DIFFERENT CONTEXTS AND SIMILARITY SCORES

Models	Description	Recc.	Prec.	Rec.	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**

B. 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.

Fig. 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

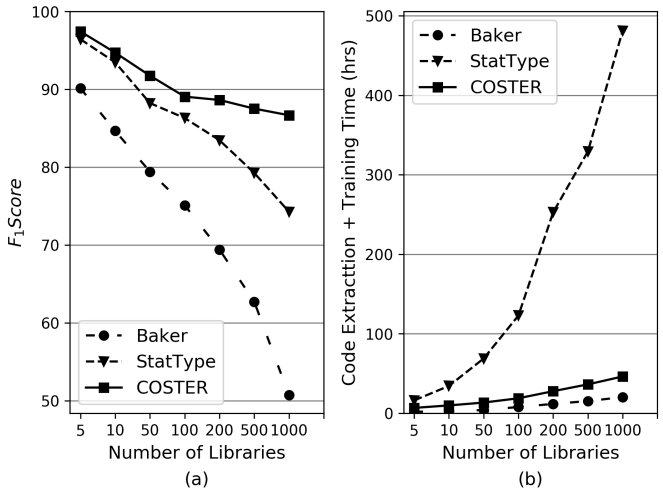


Fig. 3. The effect of increasing the number of libraries on the (a) performance (i.e., F_1 score) and (b) Code extraction + Training time of Baker, StatType and COSTER

(discussed more in the next analysis).

With respect to timing, shown in Fig. 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.

C. 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, 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.

D. Effect of receiver expression types

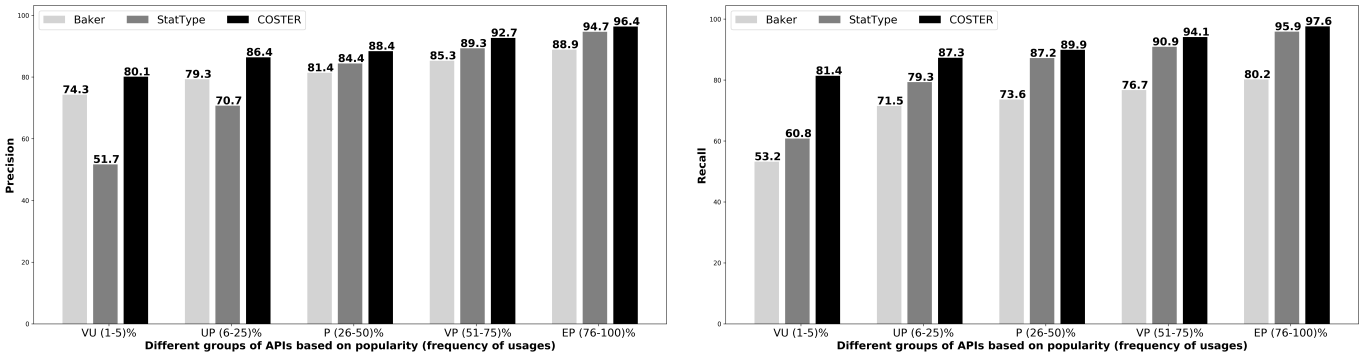
We categorized receiver expressions of API method or field calls based on their AST node types. We were interested in learning whether the performance of Baker, StatType, and COSTER vary across different receiver expression types.

Table VIII 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.

TABLE VIII
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 Inst. Creat.	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 Invoc.	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 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. The lack of code examples contributes to the difference between StatType and COSTER for other expression types (discussed in Section VI-C). 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



(a) (b)

Fig. 4. Comparing precision and recall of Baker, StatType and COSTER for API groups of different popularity.

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 [23] 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.

E. Multiple Mapping Cardinality Analysis

Name ambiguity poses a threat for resolving FQNs of APIs, as indicated by prior studies [2], [5], [11], [13]. 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 IX 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 IX 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 IX, 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.

TABLE IX
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**

F. 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 following method call statement: "*DownloadManager.getInstance().getDownloadsListFiltered().toString()*", 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%).

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.

VII. 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 VI-B). 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, the first two authors of the paper manually validated the random selection of those compilable Java source files, and they did not find any such errors.

Finally, we consider the likelihood score, cosine [20] based context similarity score and Levenshtein distance [21] 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.

VIII. Related Work

One closely related work to ours is that of Baker [5]. 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 [11]. 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 V-C).

Another related work is Partial program analysis (PPA) [8]. 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 [2] uses PPA [8] to link between code elements and their documentation. However, 47% of libraries in the Maven repository do not contain any documentation [10]. Therefore, RecoDoc cannot be applied to those libraries. ACE [13] 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 [5], [25]–[28]. JSNice [25] uses conditional random fields to perform a joint prediction of type annotation for JavaScript variables. DeepTyper [26] leverages a neural machine translation to provide type suggestions for JS code whereas NLP2Type [28] uses a deep neural network to infer the function and its parameter from docstring. Tran et al. [27] recognize the variable name from minified JS, and the work of Xu et al. [29] 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 [30], [31], Latent Semantic Indexing [32], [33], text mining [9], feature location [34], Vector Space Model [35], [36], classifier [37], Structure-oriented Information Retrieval [38], [39], ranking based learning [40] and deep learning [41]. However, these techniques primarily focus on documentation, bug reports, and email content whereas we focus on linking between code elements.

IX. Conclusion

This paper 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. COSTER collects locally specific source code elements (i.e., local context) and globally related tokens (i.e., global context) for each API element. We calculate the likelihood of appearing those tokens and the FQN of each API element. The collected usage contexts, likelihood scores and FQNs of API elements are stored 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 for both intrinsic and extrinsic settings show that our proposed technique is more robust and time-efficient. The technique improves precision by 4-6% and recall by 3-22% along with an improvement of training time by a factor of ten in comparison with existing state-of-the-art technique. 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. Future studies can combine our solution with those techniques developed for dynamically typed programming languages.

Acknowledgments: We would like to thank the authors of StatType for providing us the implementation (both StatType and Baker) and the dataset (StatType-SO). This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] B. Dagenais and M. 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*, 2010, pp. 127–136.

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

[3] J. Singer, "Practices of software maintenance," in *Proceedings of the 15th International Conference on Software Maintenance*, 1998, pp. 139–145.

[4] C. Parnin and C. Treude, "Measuring api documentation on the web," in *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, 2011, pp. 25–30.

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

[6] 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*, 2016, pp. 391–402.

[7] E. Horton and C. Parnin, "Gistable: Evaluating the executability of code snippets on the web," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution*, 2018, pp. 217–227.

[8] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proceedings of the 23rd International Conference on Object-oriented Programming Systems Languages and Applications*, vol. 43, no. 10, 2008, pp. 313–328.

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

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

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

[12] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 1–5.

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

[14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38–49.

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

[16] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.

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

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

[19] R. Lau, R. Rosenfeld, and S. Roukos, "Trigger-based language models: A maximum entropy approach," in *Proceedings of the 19th International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, 1993, pp. 45–48.

[20] R. Mihalcea, C. Corley, C. Strapparava *et al.*, "Corpus-based and knowledge-based measures of text semantic similarity," in *AAAI*, vol. 6, 2006, pp. 775–780.

[21] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

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

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

[24] M. Aickin and H. Gensler, "Adjusting for multiple testing when reporting research results: the bonferroni vs holm methods." *American journal of public health*, vol. 86, no. 5, pp. 726–728, 1996.

[25] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages*, vol. 50, no. 1, 2015, pp. 111–124.

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

[27] H. Tran, N. Tran, S. Nguyen, H. Nguyen, and T. Nguyen, "Recovering variable names for minified code with usage contexts," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1–11.

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

[29] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 24th Joint Meeting on the Foundations of Software Engineering*, 2016, pp. 607–618.

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

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

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

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

[34] 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*, 2007, pp. 234–243.

[35] L. 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*, 2014, pp. 171–180.

[36] 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*, 2012, pp. 14–24.

[37] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1597–1610, 2013.

[38] 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*, 2011, pp. 111–120.

[39] 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*, 2013, pp. 345–355.

[40] 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*, 2014, pp. 689–699.

[41] 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*, 2015, pp. 476–481.