

A Simple, Efficient, Context-sensitive Approach for Code Completion

Muhammad Asaduzzaman^{1,*†}, Chanchal K. Roy¹, Kevin A. Schneider¹ and Daqing Hou²

¹*Department of Computer Science, University of Saskatchewan, Canada*

²*Electrical and Computer Engineering Department, Clarkson University, USA*

ABSTRACT

Code completion helps developers use application programming interfaces (APIs) and frees them from remembering every detail. In this paper, we first describe a novel technique called Context-sensitive Code Completion (CSCC) for improving the performance of API method call completion. CSCC is context sensitive in that it uses new sources of information as the context of a target method call. CSCC indexes method calls in code examples by their context. To recommend completion proposals, CSCC ranks candidate methods by the similarities between their contexts and the context of the target call. Evaluation using a set of subject systems and five popular state-of-the-art techniques suggests that CSCC performs better than existing type or example-based code completion systems. We conduct experiments to find how different contextual elements of the target call benefit CSCC. Next, we investigate the adaptability of the technique to support another form of code completion, i.e., field completion. Evaluation with eight different subject systems suggests that CSCC can easily support field completion with high accuracy. Finally, we compare CSCC with four popular statistical language models that support code completion. Results of statistical tests from our study suggest that CSCC not only outperforms those techniques that are based on token level language models, but also in most cases performs better or equally well with GraLan, the state-of-the-art graph-based language model. Copyright © 2016 John Wiley & Sons, Ltd.

Received 20 February 2015; Revised 2 April 2016; Accepted 9 April 2016

KEY WORDS: API methods; code completion; field completion; context sensitive; simhash; language model

1. INTRODUCTION

Developers rely on frameworks and libraries of APIs to ease application development. While application programming interfaces (APIs) provide ready-made solutions to complex problems, developers need to learn to use them effectively. The problem is that because of the large number of APIs, it is practically impossible to learn and remember them completely. To avoid developers having to remember every detail, modern integrated development environments provide a feature called Code Completion, which displays a sorted list of completion proposals in a popup menu for a developer to navigate and select. In a study on the Eclipse IDE, Murphy et al. [1] found that code completion is one of the top ten commands used by developers, indicating that the feature is crucial for today's development. In this paper, we focus our attention on method and field completion because these are the most frequently used forms of code completion [2] (other forms of code

*Correspondence to: Muhammad Asaduzzaman, Department of Computer Science, University of Saskatchewan, Canada.

†E-mail: parvez.usask@gmail.com

completion include word completion, method parameter completion, and statement completion). In the remainder of the paper, we use the term Code Completion to refer to method and field completion unless otherwise stated.

Existing code completion techniques can be divided into two broad categories. The first category uses mainly static type information, combined with various heuristics, to determine the target method call, but does not consider previous code examples or the context of a method call. A popular example is the default code completion system available in Eclipse, which utilizes a static type system to recommend method calls. It sorts the completion proposals either alphabetically or by relevance before displaying them to users in a popup menu. Hou and Pletcher [3] developed another technique, called Better Code Completion (BCC), that uses a combination of sorting, filtering and grouping of APIs to improve the performance of the default type-based code completion system of Eclipse.

The second category of techniques takes into account previous code examples and usage context matching to recommend target method calls [4–6]. For example, to make recommendations, the Best Matching Neighbor (BMN) [4] code completion system matches the current code completion context to previous code examples using the k-Nearest Neighbor (kNN) algorithm.

BMN has successfully demonstrated that the performance of method call completion can be improved by utilizing the context of a target API method call. BMN focuses on using a special kind of context for a given call site, i.e., the list of methods that have been invoked on the same receiver variable plus the enclosing method of the call site. But there are many other possible forms of context to be considered. As an example of other forms of context, consider the code shown in Figure 1, where a file is read via the `BufferedReader` object `br` in a `while` loop. In fact, the `readLine` method is commonly called as part of a `while` loop's condition located inside a `try-catch` block. Within a few lines of distance of the `readLine` method, developers usually create various objects related with that method call. For example, developers typically create a `BufferedReader` object from an instance of `FileReader` and later use that object to call the `readLine` method. Therefore, in addition to the methods that were previously called on the receiver object `br`, keywords (such as `while`, `try`, `new`), other methods (such as `FileReader`, `BufferedReader` constructor name) can be considered as part of the context of `readLine` as well. Adding these extra pieces of information can enrich the context of the targeted call to help recommend methods that are more relevant (`readLine`, in this case).

In this paper, we further explore the performance implications of these additional forms of context for code completion. To this end, we first propose a context-sensitive code completion technique, called Context-sensitive Code Completion (CSCC), that leverages code examples collected from repositories to extract method contexts to support code completion. Given a method call, we capture as its context *any method names, Java keywords, class or interface names* that appear within four lines of code. In this way, we build a database of context-method pairs as potential matching candidates. We use tokenization rather than parsing and advanced analysis to collect the context data, so our technique is simple.

When completing code, given the receiver object, we use its type name and context to search for method calls whose contexts match with that of the receiver object. To scale up the search, we use *simhash* technique [7] to quickly eliminate the majority of non-matching candidates. This allows us to further

```
String line;
StringBuilder sb = new StringBuilder();
br= new BufferedReader(new
    FileReader("file.txt"));
try{
while ((line = br.readLine()) != null){
    sb . append ( l i n e ) ;
    sb . append ( '\n ' ) ;
}
}finally {br . close ();}
```

Figure 1. An example of reading a file where `readLine` is the target of code completion.

refine the remaining much smaller set of matching candidates using more computationally expensive textual distance measures. This also makes our technique efficient and scalable. We sort the matching candidates using similarity scores, and recommend the top candidates to complete the method call.

We compare our context-sensitive code completion technique with five other state-of-the-art techniques [4, 8] using eight open source software systems. Results from the evaluation suggest that our proposed technique outperforms all five state-of-the-art type or example-based systems that we have compared. Moreover, to understand how exactly the context of a method call affects code completion, we propose a taxonomy for the contextual elements of a method call and compare how different techniques perform for each category of contextual elements (see Section 6). This experiment helps us clearly understand the strengths and limitations of CSCC and other existing techniques. We also conduct a set of experiments on CSCC to uncover the effect of context length, impact of different context information, effectiveness in cross-project prediction, performance for different frameworks or libraries, and evaluate building context using the four lines following a method call.

After releasing CSCC as an Eclipse plugin we received a number of requests to extend support for fields. During our investigation we found that the context CSCC uses for method call completion can easily capture the context of field access too. Evaluation on field completions using eight different subject systems and with four state-of-the-art code completion techniques reveals that CSCC is able to outperform all four techniques by a substantial margin.

Recently, a number of techniques have been developed that uses statistical language models for predicting the next token or completing API elements [5, 6, 9, 10]. They use different information sources to capture the context. These include token sequences, caching, API usage graphs or a combination of them. We are interested on how well CSCC performs compared to those techniques. Toward this goal, we compare CSCC with four other statistical language model techniques and present the study results in this paper.

This paper makes the following contributions, where (1), (2) and (3) are from the original ICSME 2014 paper [11], and the rest are new in this paper:

1. A technique called CSCC to support code completion using a new kind of context and previous code examples as a knowledge-base.
2. A quantitative comparison of the proposed technique CSCC with five existing state-of-the-art tools that shows the effectiveness of our proposed technique.
3. A taxonomy of method call context elements and an experiment that helps to identify strengths and limitations of CSCC and other existing techniques (see Section 6 for details of the taxonomy).
4. A set of studies that helps to understand different aspects of the technique (see Section 5).
5. Extending CSCC for field completion and a quantitative comparison with four other state-of-the-art tools.
6. A comparison of CSCC with four state-of-the-art statistical language model code completion techniques.

The remainder of the paper is organized as follows. Section 2 briefly describes related work. Section 3 describes our proposed technique CSCC. Section 4 compares CSCC with various code completion techniques using eight open source software systems for completing API method calls. We conduct a set of studies to uncover different aspects of the technique and present the results in Section 5. Section 6 describes the extension of the technique to support field completion including evaluation with four other state-of-the-art tools. We compare CSCC with statistical language model based code completion techniques in Section 7. Section 8 summarizes the threats to validity. Finally, Section 9 concludes the paper.

2. RELATED WORK

An important work related to our study is that of Bruch et al. [4]. They propose the BMN completion system that uses the kNN algorithm to recommend method calls for a particular receiver object. The most fundamental difference between BMN and CSCC lies in their definition of context. Our definition of a method call context includes any method names, keywords, class or interface names within the four lines prior to a method call, whereas BMN's context is made of the set of methods

that have been called on the receiver variable plus the enclosing method. Because of this difference, BMN and CSCC use different techniques to calculate similarities and distances. Last, BMN uses frequency of method calls to rank completion proposals, whereas CSCC ranks them based on distance measures.

Hou and Pletcher [3, 8, 12] propose a code completion technique that uses a combination of sorting, filtering and grouping of APIs. They implement the technique in a research prototype called BCC. BCC can sort completion proposals based on the type-hierarchy or frequency count of method calls. It can filter non-API public methods. BCC also allows developers to manually specify a set of methods that are logically related, and thus belong to the same group and appear together while displaying completion proposals in a popup menu. However, BCC does not leverage previous code examples. Moreover, BCC requires the filters to be manually specified which can only be performed by expert users of code libraries and frameworks. However, because CSCC considers the usage context of method calls to recommend completion proposals, methods that are not appropriate to call in a particular context would be automatically filtered out. So CSCC would require less effort to use than BCC.

Another important work related to our study is the GraLan, a graph-based statistical language model that targets completing API elements [6]. The term API element refer to method call, field accesses and control units (such as for, while, if etc.) used in the API usage examples. The technique mines the source code to generate API usage graphs, called Groums. Given an editing location, the technique determines the API usage subgraphs surrounding the current location and use them as context. GraLan then computes the probability of extending each context graph with an additional node. These additional nodes are collected, ranked and recommended for code completion. CSCC differs from GraLan in terms of context definition, recommendation formulation and ranking strategy. For example, GraLan rank completion proposals based on their probability distributions, but CSCC uses textual distance measures.

Besides GraLan, a number of code completion systems have been proposed that uses statistical language models. Hindle et al. develop a code suggestion engine that uses the widely adopted N-gram model to recommend the next token [5]. Tu et al. develop a language model (known as Cache LM) that uses a cache component to capture the localness of software [10]. Christine et al. develop an Eclipse plugin, called CACHECA, that combines the native suggestions made by the Eclipse IDE with that of the cache language model [13]. Nguyen et al. propose a statistical semantic language model for source code, called SLAMC [9]. The model incorporates the semantic information of code tokens. It also captures global concerns using a topic model and pairwise association of language elements. The model has been used to develop a code suggestion engine to predict the next token. Raychev et al. [14] propose a code completion technique, SLANG, that collects sequence of method calls to create a statistical language model. When a developer requests for a code completion, the tool completes the editing location using the highest ranked sequence of method calls computed by the language model. While all these techniques except SLANG work at the lexical level, CSCC works at the API level. CSCC differs from SLANG in context formulation. The technique not only collects method calls, but also collects keywords and type names.

Nguyen et al. [15, 16] use a graph-based algorithm to develop a context-sensitive code completion technique, called GraPacc. The technique mines API usage graphs or Groums to capture API usage patterns in open source code bases. This creates an API usage database. During the development phase, the technique extracts context-sensitive features and matches them with usage patterns in the database. It then recommends a list of matched patterns to complete the remaining code.

Although both GraPacc and CSCC utilize code context to make recommendations, the goals and approaches are different. GraPacc recommends multiple statements at a time, but CSCC completes a single method call. Similar to GraLan, GraPacc also leverages Groums for identifying code context. Because the objective of CSCC is similar to GraLan and we already include that in our study, we did not compare with GraPacc.

Robbes and Lanza [2] propose a set of approaches to support code completion that use program history to recommend completion proposals. They define a benchmark to measure the usefulness of a code completion system and evaluate eight different code completion algorithms. They found that the typed optimist completion technique provides better results than any other techniques because it merges the benefits of two different techniques.

The program change history can be considered the *temporal* context for a method call, whereas ours is the *spatial* context. While their technique requires a change-based software repository to collect program history, our technique can work with any repository.

Research related with recommending source code examples is also related to our study because of their use of context. Among various work on code examples recommendation, the most relevant work to ours is that of Holmes and Murphy [17]. They use the notion of structural context to recommend source code examples. The context in their case contains information about inheritance, method calls and types declared or used in a method. Although our method call usage context share some similarity with theirs, the objectives are completely different.

There are also a number of other techniques or tools that make use of previous code examples, but their goals are different than ours. For example, Mooty et al. [18] develop an Eclipse plugin, called Calcite, that helps developers to correctly instantiate a class or interface using existing code examples. While Calcite helps instantiate a class, we help developers complete method calls. Zhang et al. [19] develop a tool, called Precise, that mines existing code bases to recommend appropriate parameters for method calls. Hill and Rideout [20] focus on automatic completion of a method body by searching similar code fragments or code clones in a code-base. Lee et al. [21] introduce a technique that identify changes of program entities (such as method names) during the evolution of a software system. When a developer requests for a completion, the technique presents the program entities along with their changes through code completion. It also helps developers to navigate to the past code examples to see the changes. Jacobellis et al. [22] leverage code completion to automate the edit operations of source code from user specified custom, reusable template of code changes.

Keyword programming [23] is also related to our study, but it defines a completely different way of user interaction for code completion. Instead of typing a method name, users type some keywords that give hints about the method the user is trying to call. The algorithm then automatically completes the method call or makes appropriate suggestions to complete the remaining part. Han and Miller [24] later introduce abbreviation completion that uses a non-predefined set of inputs to complete the target method call.

Previously discussed techniques make use of floating menus to present completion proposals and none focuses on the improvement of the user interface. Omar et al. [25] present an approach that allows library developers to integrate specialized interfaces, called palettes, directly into the editor. The benefit of the approach is that developers do not need to write the code explicitly, rather the specialized interface allows users to provide required input and generate the appropriate code. They developed a tool, named Graphite, that allows Java developers to write a regular expression in a palette and found that the addition of such a specialized interface is helpful to the professional developers. Instead of focusing on a specialized interface for code completion, we focus on predicting target method calls as a basis for suggesting completion proposals. However, palettes can complement our technique to complete method parameters, which remains as a future work.

3. PROPOSED ALGORITHM

In this section, we describe our algorithm for finding method calls to recommend for a target object. Figure 2 presents an overview of the process. Our example-based, context-sensitive code completion system works in three steps:

- Collect the usage context of API method calls from code examples and index them by their contexts to support quick access. We model the context of an API method call by method calls, Java keywords and type names that appear within the four lines prior to the receiver object that called the method. We hypothesize that these elements around the target method call can provide a better, fuller context than other approaches [4, 8].
- Search for method calls whose context matches with that of the target object. One approach would be to directly measure the similarity between the context of the target object and that of each method call in the example code base using string edit-distances. However, string edit-distance operations are computationally expensive. To speed up the search, we instead use the Hamming distance over the *simhash* values as similarity measures. We determine a smaller list of method

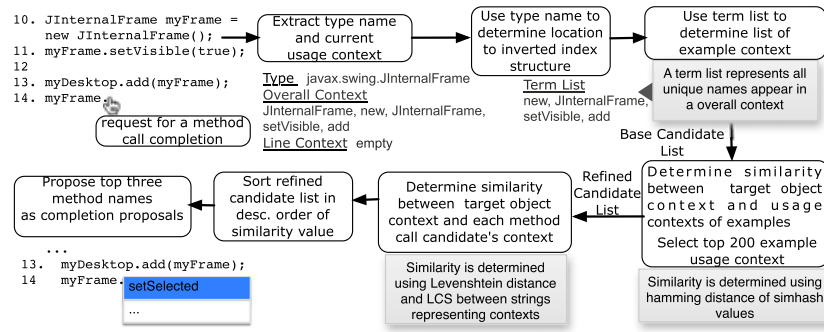


Figure 2. An overview of CSCC's recommendation process starting from a code completion request.

names that are the more likely candidates for code completion, which we refer to as the *candidate list*.

- The final step synthesizes the method calls from the candidate list. For each method name in the candidate list, we use a combination of token-based Longest Common Subsequence (LCS) and Levenshtein distance to determine a similarity value of its context with that of the receiver object. We then sort the method names in descending order of similarity value and recommend the top three names to complete the method call.

We describe the three steps in detail as follows.

3.1. Collect usage context of method calls

In this step, CSCC mines code examples to find the usage context of API method calls. To capture a method call context, we consider the content of the n lines prior to it, including the line where the target method call appears. In this study, we use $n=4$ and we validate this decision in Section 5.

We collect the following three kinds of information from the four lines of context, which we refer to as the *overall context* of the method call:

1. Any method names.
2. Any Java keywords except access specifiers.
3. Any class or interface names.

When extracting the overall context, we ignore blank lines, comment lines or lines containing only curly braces. We also remove any duplicate tokens from the overall context.

In addition, we separately collect a *line context* for the target method, which includes any method names, keywords (except access specifiers), class or interface names and assignment operators that appear on the same line but before the target method call. When the overall contexts are completely different and fail to match, line contexts act as a secondary criterion for matching.

To further explain the construction of both overall and line context, consider the method call at line number 13 as shown in Figure 3. The contents of both contexts include tokens and their locations. Note that although line number 10 is within four lines of our target method call *getDisplay*, it is not considered part of the context as it is located outside of the *createContents* method containing the target call *getDisplay*.

We use a two-level indexing scheme to organize the collected usage contexts of method calls (see Figure 4 for an example of it). We use the type name of a receiver object to group all method calls that have been invoked on the type. We use an inverted index structure [26] to organize such a group of method calls. More specifically, an inverted index is a data structure that maps each term to its location in a document. We represent each overall context of a method call as a document, and use tokens from the context to index the set of documents where they appear.

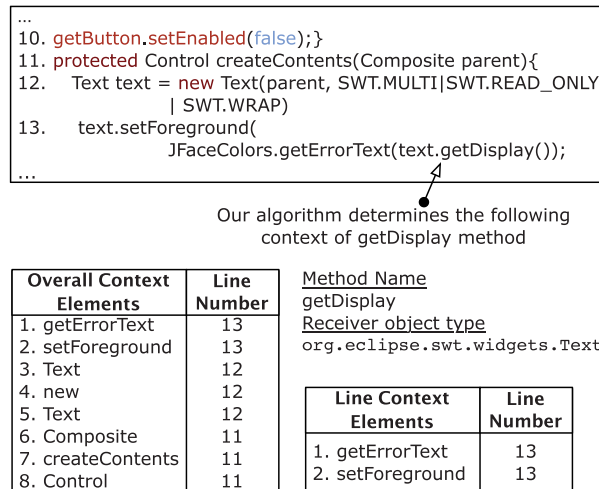
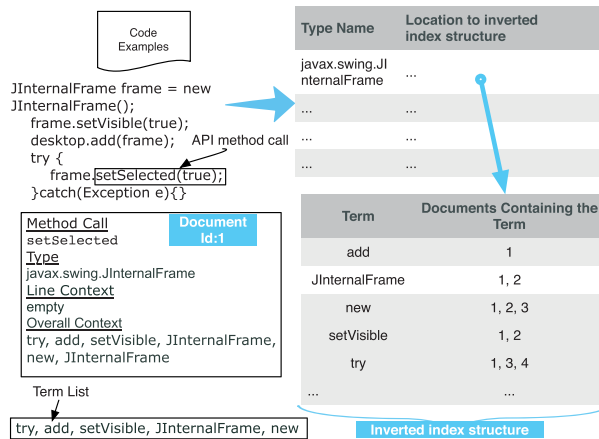
Figure 3. Overall context and line context for *getDisplay*.

Figure 4. Database of method call contexts are grouped by receiver types using inverted index structure. The figure on the left side shows the collected context information for an API method call. The figure on the right side shows how we use the information to build an inverted index structure.

3.2. Determine candidates for code completion

When a user requests a method call completion (for example, in Eclipse typing a dot (.) after an object name initiates such a request), our algorithm first extracts both overall and line contexts for the receiver object. To find candidate methods for code completion, we match the current context to those extracted from the example code base. Specifically, we use the type name of the receiver object as an index to determine the related inverted index structure, which contains all method calls made on the receiver type (Figure 4). We then use tokens from the overall context as keys to the inverted index structure to collect all those method calls in the code examples that have the same type as the receiver object. We refer to these matching method calls as the *base candidate list*.

The *base candidate list* often contains thousands of method calls, so we need to reduce them to a small number of most likely candidates in order to recommend. We follow a two-step process to search for the most likely candidates. We first use the *simhash* technique to determine a short list of method names (currently the top 200 that are deemed most similar to the target context) that are more likely to complete the current method call and quickly eliminate the majority of others. To calculate string similarity metrics, we concatenate all the tokens of each context and generate a *simhash* value for the concatenated string (see Figure 5 for an example). We use the *simhash*

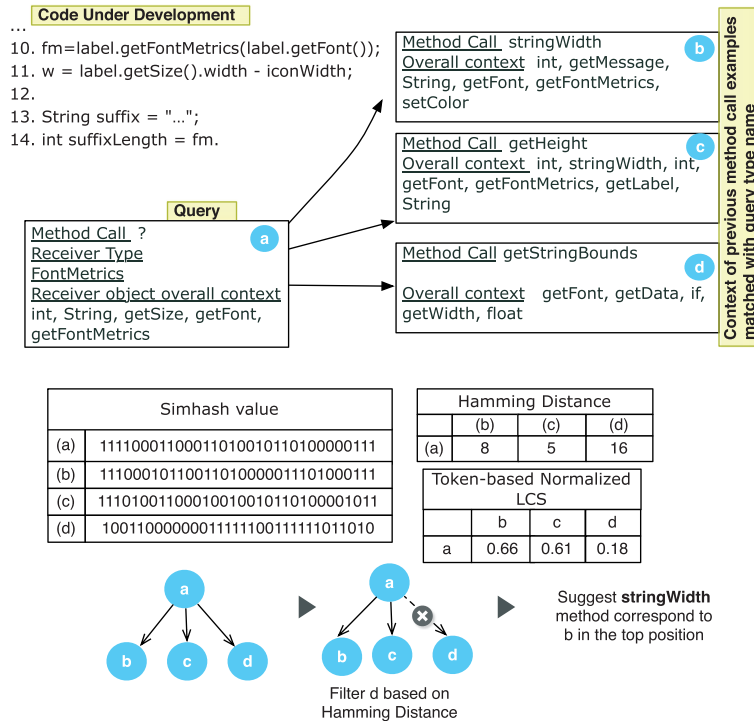


Figure 5. Context similarities are measured using the Hamming distance between simhash values.

technique to eliminate most of the irrelevant matching candidates because it is both fast and scalable. Second, we use the normalized LCS and Levenshtein distances to measure the fine-grained similarity between the target context and the context of each likely matching candidate to obtain a refined candidate list. Calculating LCS and Levenshtein distances are both time consuming operations. Although they provide fine-grained similarity measures, because of the near real-time constraint on code completion, we cannot apply them directly on the *base candidate list*.

We use the *simhash* technique [7] to identify the most likely method call candidates. *Simhash* uses a cryptographic hash function to generate binary hash keys, also known as *simhash* values. An important property of *simhash* is that strings that are similar to each other have either identical or very similar *simhash* values. Therefore, we determine the similarity between each pair of contexts using the Hamming distance of their corresponding *simhash* values. We use the Hamming distance of the overall context to sort the matching candidates unless the Hamming distance of the line context exceeds a predefined threshold value, in which case we use the Hamming distance of the line context as the distance measure. After sorting by similarity, we take the top k method contexts as the likely matching candidates of the target context. After experimentation with different values of k , we found that $k=200$ is a good choice to work with and we use that value in our study. We refer to this list as the *refined candidate list*.

To recommend method calls, we further sort the method names in the refined candidate list by combining both overall and line context similarities as follows. We use the normalized LCS distance to measure the similarity of the token sequences from the overall context. We use Levenshtein distance to measure the similarity of the token sequences from the line context. We sort matching candidates in descending order of their overall context similarity. However, in case of a tie for the overall context similarity, we use the line context similarity. We ignore all matching candidates whose similarity value drop to a certain threshold. We empirically found that 0.30 is a good choice to work with.

The *simhash* technique has been found effective for detecting similar pages in a large collection of web documents [27] and also has been used successfully in detecting similar code fragments in code clone detection [28]. Although various hash functions are available, we use the *Jenkin* hash function

because it has been found effective in a previous study [28]. We generate a 64 bit *simhash* value for both overall and line contexts of the target object. To save computation time, we precompute the *simhash* values. We determine the similarity between each pair of contexts using the Hamming distance of the corresponding *simhash* values.

3.3. Recommend top-3 method calls

The objective of this step is to recommend a list of completion proposals (i.e., method names). Because there are many code examples associated with the same method call, the sorted list of method names obtained from the previous step may contain many duplicates. After eliminating duplicates, we present the top three method names to the users.

4. EVALUATION

We evaluate our technique and compare CSCC with five state-of-the-art code completion systems using eight open-source systems (Table I). For a given subject system, we determine all locations where methods from a target API have been called. This set of method calls constitutes our data set. We then apply the ten-fold cross validation technique [4] to measure the performance of each algorithm. This is a popular way of measuring performance of information retrieval systems [29] and has been used previously in many research projects. First, for each system we divide the data set into ten different folds, each containing an equal number of method calls. Next, for each fold, we use code examples from the nine other folds to train the technique for method call completion. The remaining fold is used to test the performance of the technique.

We use recall to measure how many cases a technique produces relevant (correct) recommendations out of the total test cases. Clearly a technique that recommends all possible methods would always achieve a great recall of 1. That is why we consider the precision measure. It is defined as the ratio between the number of times a technique produces relevant recommendations and the number of times that technique produces any recommendations. The higher a relevant recommendation is in the list of recommendations, the better. That is why we collect precision measure at top-1, top-3 and top-10 recommendations. Finally, we use the F-Measure, a widely accepted measure, to correlate precision and recall by computing their harmonic means.

$$Recall = \frac{recommendations \text{ made} \cap relevant}{recommendations \text{ requested}} \quad (1)$$

$$Precision = \frac{recommendations \text{ made} \cap relevant}{recommendations \text{ made}} \quad (2)$$

$$F - measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \quad (3)$$

where *recommendations requested* is the number of method calls in our test data for which we will make a code completion request. *Recommendations made* is the number of times where a code completion system makes a recommendation.

4.1. Test systems

We chose to focus on two API's, SWT and Swing/AWT, as the target for our evaluation. These are popular libraries extensively used for developing GUI applications.

We selected four systems that used SWT. The largest one is Eclipse 3.5.2 [30], a popular open source IDE. Vuze [31] is a P2P file sharing client using the bittorrent protocol. Subversive [32] provides support to work with Subversion directly from Eclipse. RSSOwl [33] is an RSS newsreader.

Table I. Evaluation results of code completion systems. *Delta* shows the improvement of CSCC over BMN.

Subject systems	Precision					Recall					F-measure					Delta (%)					
	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC	Delta (%)							
Eclipse	Top-1	0.006	0.01	0.24	0.31	0.36	0.60	24	1	1	1	0.77	0.99	22	0.008	0.020	0.39	0.47	0.49	0.75	26
	Top-3	0.052	0.20	0.52	0.49	0.63	0.80	17	1	1	1	0.77	0.99	22	0.10	0.34	0.68	0.66	0.69	0.88	19
	Top-10	0.14	0.34	0.80	0.73	0.69	0.90	21	1	1	1	0.77	0.99	22	0.25	0.51	0.89	0.84	0.73	0.94	21
Vuze	Top-1	0.005	0.10	0.23	0.33	0.35	0.56	21	1	1	1	0.76	0.98	22	0.009	0.18	0.37	0.50	0.48	0.71	23
	Top-3	0.03	0.16	0.49	0.49	0.59	0.73	14	1	1	1	0.76	0.98	22	0.06	0.28	0.66	0.66	0.66	0.84	18
	Top-10	0.20	0.36	0.94	0.71	0.61	0.83	22	1	1	1	0.76	0.98	22	0.33	0.53	0.97	0.83	0.68	0.90	22
Subversive	Top-1	0.01	0.03	0.30	0.36	0.58	0.68	10	1	1	1	0.38	0.97	60	0.02	0.058	0.46	0.53	0.46	0.80	34
	Top-3	0.02	0.07	0.63	0.62	0.77	0.86	9	1	1	1	0.38	0.97	60	0.04	0.13	0.77	0.77	0.51	0.91	40
	Top-10	0.07	0.20	0.96	0.88	0.79	0.91	12	1	1	1	0.38	0.97	60	0.13	0.34	0.98	0.94	0.51	0.94	43
Rsowl	Top-1	0.01	0.078	0.25	0.32	0.48	0.65	17	1	1	1	0.72	0.98	26	0.20	0.14	0.40	0.48	0.58	0.78	20
	Top-3	0.024	0.16	0.58	0.51	0.74	0.84	10	1	1	1	0.72	0.98	26	0.046	0.28	0.73	0.68	0.73	0.90	17
	Top-10	0.077	0.29	0.85	0.74	0.80	0.90	10	1	1	1	0.72	0.98	26	0.14	0.45	0.92	0.85	0.76	0.94	18
NetBeans	Top-1	0.12	0.13	0.34	0.29	0.43	0.66	23	1	1	1	0.67	0.98	31	0.21	0.23	0.51	0.45	0.52	0.79	27
	Top-3	0.18	0.25	0.62	0.53	0.67	0.86	19	1	1	1	0.67	0.98	31	0.31	0.40	0.77	0.69	0.67	0.92	25
	Top-10	0.36	0.48	0.86	0.73	0.70	0.92	22	1	1	1	0.67	0.98	31	0.70	0.65	0.92	0.84	0.68	0.95	27
JEdit	Top-1	0.009	0.12	0.41	0.35	0.52	0.62	10	1	1	0.98	0.70	0.94	24	0.02	0.21	0.58	0.52	0.60	0.75	15
	Top-3	0.14	0.29	0.62	0.53	0.74	0.79	5	1	1	0.98	0.70	0.94	24	0.25	0.45	0.77	0.69	0.72	0.86	14
	Top-10	0.32	0.49	0.83	0.74	0.79	0.85	6	1	1	0.98	0.70	0.94	24	0.48	0.66	0.91	0.84	0.74	0.89	15
ArgoUML	Top-1	0.03	0.13	0.40	0.32	0.46	0.58	12	1	1	0.99	0.68	0.95	27	0.058	0.23	0.57	0.48	0.55	0.72	17
	Top-3	0.12	0.27	0.65	0.53	0.68	0.74	6	1	1	0.99	0.68	0.95	27	0.21	0.43	0.79	0.69	0.68	0.83	15
	Top-10	0.27	0.47	0.83	0.78	0.74	0.81	7	1	1	0.99	0.68	0.95	27	0.43	0.64	0.91	0.87	0.71	0.87	16
JFreeChart	Top-1	0.02	0.08	0.35	0.32	0.42	0.63	21	1	1	1	0.75	0.98	23	0.040	0.15	0.52	0.48	0.54	0.77	23
	Top-3	0.05	0.19	0.52	0.63	0.76	0.85	9	1	1	1	0.75	0.98	23	0.10	0.32	0.68	0.77	0.75	0.91	16
	Top-10	0.27	0.58	0.63	0.92	0.84	0.94	10	1	1	1	0.75	0.98	23	0.43	0.73	0.77	0.96	0.79	0.96	17

We also chose four open source software systems for AWT/Swing. NetBeans 7.3.1 [34], the largest, is an IDE; jEdit [35] is a text editor; ArgoUML [36] is a UML modeling tool and JFreeChart [37] is a Java charting library.

4.2. Evaluation results

In this section, we discuss the results of evaluating and comparing CSCC with five other code completion systems (ECCAlpha, ECCRelevance, Frequency-based Code Completion (FCC), BCC and BMN) using the eight test systems. We have introduced BCC and BMN earlier. ECCAlpha and ECCRelevance are two default Eclipse code completion systems that leverage the static type system. ECCAlpha sorts the completion proposals in alphabetical order, and ECCRelevance uses a positive integer value, called relevance, to sort them. The value is calculated based on the expected type of the expression as well as the types in the code context (such as return types, cast types, variable types etc.). The Frequency-based code completion system (FCC) considers the frequency of method calls in a model to make recommendations. The more frequent a method occurs, the higher its position is in the completion proposals.

Table I shows the precision, recall and F-measure values for the six code completion systems. The top four rows are results collected for SWT, and the bottom four rows for AWT/Swing. Overall, CSCC achieves higher precision and recall values than any of the other techniques for both single and top three completion proposals.

For the top three proposals, it has precision of 73–86% and recall of 97–99%. The recalls for ECCAlpha, ECCRelevance and BCC are all one. But both ECCAlpha and ECCRelevance performed poorly, and BCC outperformed both of them.

Except in a few cases, BCC also outperforms FCC. Furthermore, the performance of FCC is not great, while its recall is close to 100%, its precision is only 49–62% for the top three proposals. Interestingly, BMN did not perform well either. Although its precision is better than both BCC and FCC for the single and top-3 suggestions, its recall is poorer in both cases. This is because of the fact that BMN targets local variable method calls but there are many places in source code where methods are called on fields, parameters, chained expressions or even static types. We performed further experiments to elaborate on this issue in Section 4.3.1.

The results for AWT/Swing shown in the bottom four rows of Table I are consistent with those of SWT. For example, for the top three proposals and for the largest subject system (NetBeans), the F-measure of CSCC is higher by 15% compared to the closest performing technique.

To test whether CSCC performed significantly better than other techniques, we also performed directional Wilcoxon Signed Rank Tests for the top three completion proposals between CSCC and other techniques. The null hypothesis is that there is no difference in precision and recall values for top-3 completion proposals. The test shows that the difference in precision and recall values between CSCC and other techniques are statistically significant at the p value of 0.05.

4.3. Evaluation using a taxonomy of method calls

While the evaluation in Section 4.2 provides a ranking of the six techniques in terms of their performance, it does not reveal what factors contribute to CSCC's better performance. We hypothesize that it is because of CSCC's ability to capture a fuller context for method calls. To further shed light on this hypothesis, we propose a taxonomy for the characteristics of a method context, and compare the techniques using each category of method call characteristics within the taxonomy.

Our taxonomy (Figure 6) includes three categories of characteristics for the context of a target method call: the AST node types for its receiver expression, the AST node types for its parent node and the enclosing overridden method that contains the target method call. Although we cannot guarantee that the taxonomy covers every possible aspect of method call completions, it can provide insights into code completion techniques and can also help us to decide where more effort is needed.

We use the following procedure for our evaluation. For each category of method calls within each test fold of a subject system, we count how many of them are correctly predicted by a code completion technique. For each system, we then present the final result after adding the numbers for all ten test folds.

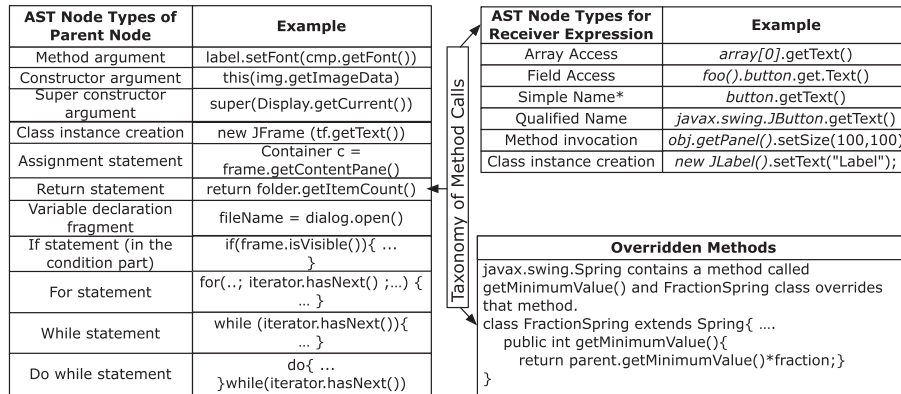


Figure 6. Taxonomy of method calls.

4.3.1. AST node type for receiver expression. We categorize the receiver expressions of the test method calls according to their AST node types. We count the number of test method calls that each code completion technique correctly recommends for each kind of AST node. Table II shows the results for the top three proposals from the two largest test systems, Eclipse and NetBeans.

Table II suggests that the majority of receiver expressions fall in the simple name category. A simple name can be a variable name (declared as a method parameter, a local variable or a field) or a type name (static method calls). The original BMN technique considers only local variables and their types to compute completion proposals. However, Table II shows that many receiver expressions of method calls are not local variables, and thus for which BMN produces no recommendations. This explains why we did not receive good results for BMN (Table I). The way BMN collects usage context is quite limited. In contrast, CSCC can identify usage context even when the receiver is not a local variable and thus can recommend method names for those cases too.

We performed another experiment where we train and test both BMN and CSCC using only those method calls where the receiver is a local variable. For the top three proposals, BMN achieves 68% recall and 77% precision for the Eclipse system, both of which are higher than those of any other techniques except CSCC. The recall and precision for CSCC are 84% and 86%, respectively, indicating that CSCC performs better than BMN even when the receivers are local variables.

4.3.2. AST node types of parent node. We consider those method call expressions where a particular type of object or value is expected. For example, a framework method call can be located in the condition part of an if statement that expects a boolean value. A method call can be located in the right hand side of an assignment expression. If the left hand side of that assignment expression is of

Table II. Categorization of method calls for different AST node types for receiver expressions (for the top-3 proposals).

Expression types	NetBeans			Eclipse		
	Quantity (%)	Correctly predicted BMN (%)	Correctly predicted CSCC (%)	Quantity (%)	Correctly predicted BMN (%)	Correctly predicted CSCC (%)
Array access	0.54	0	54.16	1.65	0	79.73
Class instance creation	0.18	0	100	0.11	0	100
Field access	0.60	0	80.77	0.49	0	77.27
Method invocation	21.66	0	94	11.88	0	75.42
Simple name	Type	5.48	79.21	3.02	96.26	98.26
	Local variable	32.13	68.26	41.86	0.64	83.14
	Field	51.94	52.07	46.92	50	75.46
	Parameter	10.44	52.80	9.47	46.81	79.50
	Total	74.08	58.84	85.02	56.86	79.65
Qualified name	2.94	57.36	82.17	0.85	60.53	71.05

Container type, the right hand side should return an object of type Container or a sub-type of it. The goal is to identify how well techniques that consider type information as context perform in these cases compared to others. To make the result comparable with the BMN code completion system, we consider those method calls where the receiver is a local variable.

Table III shows the results of top three proposals for the Eclipse system. We can see that considering the expected type as contextual information can help improve code completion techniques. That is why the accuracy of BCC becomes close to that of CSCC, which achieves the highest accuracy for all but one category of AST node. Other code completion systems, such as BMN, FCC and default code completion systems of Eclipse, did not perform well in this experiment.

4.3.3. Overridden methods. The objective is to verify whether methods called from within overridden methods impose any challenge to the evaluated code completion techniques. Our informal observation is that it may be difficult to identify usage context for method calls within overridden methods. When we manually analyze some of the code examples, we notice that method calls within overridden methods may contain very limited contextual information. For example, a number of methods in Java Swing applications result from implementing the ActionListener interface and those methods contain only a few methods called on the receiver objects. This can affect the performance of those techniques that leverage receiver method calls for making recommendations. Similar to the previous experiment we test only those method calls where the receiver is a local variable. CSCC again performs better than any other techniques in this experiment. Table IV summarizes the results of the study. While CSCC correctly recommends more than 84% method calls for the top three recommendations for both subject systems, none of the other technique achieves more than 58% accuracy.

We were interested to see whether we can take advantage of this special case. There are two possible ways we can exploit the overridden method. First, we can include the method name in the context. Second, we can use the name to index training examples. To recommend a method call inside an overridden method we can then access candidates by using receiver type name and overridden method name.

To check the first option, we explicitly add the overridden method name to both overall and line contexts. It should be noted that CSCC may include the overridden method name as part of the overall context, but only in the case where the overridden method name is located within four lines of distance. In those cases the overall context may contain the overridden method name twice. This

Table III. Correctly predicted method calls where the parent expression expects a particular type of value (top-3 proposals for the Eclipse system).

AST node types of parent node	Total cases	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC
Method argument	696	6	441	527	378	336	553
Assignment	429	3	282	338	144	184	305
If statement	526	36	47	225	217	214	373
While statement	8	0	0	4	0	2	4
Return	99	1	56	62	38	32	65
Variable declaration fragment	1388	10	738	1018	498	515	1084
For statement	5	0	3	3	0	0	4
Class instance creation	126	3	76	95	46	54	98
Prefix expression	425	30	75	396	282	228	346
Total	3702	89	1718	2668	1603	1565	2832
		2.4%	46.4%	72%	44.5%	42.27%	76.5%

Table IV. Percentage of correctly predicted method calls that are called in the overridden methods (for the top-3 proposals).

Subject systems	Percentage of correctly predicted method calls by Code Completion Systems (%)					
	ECCAlpha	ECCRel	BCC	FCC	BMN	CSCC
Eclipse	4.98	15.19	55.56	55.12	57.04	84.28
NetBeans	12.00	33.60	52.20	52.34	56.25	85.24

even gives more weight on the overridden method name. For this experiment, we use Eclipse and NetBeans as subject systems, and we test the accuracy of correctly predicting method calls within overridden methods. Table V shows the percentage of correctly predicted method calls that are located in overridden methods. For the Eclipse system, adding overridden method name to both contexts results in an increase of 2.52%. For the NetBeans system, we also obtain a relative improvement of 1.75%. The results from the study thus suggest that adding overridden method names to the context can help better model method call usage context for those that are located within overridden methods.

To check the second option, we again use Eclipse and NetBeans as subject systems. We indexed the code examples by receiver type and enclosing method name and we test the accuracy of the correctly predicted method calls for those test cases where the method calls appeared within an overridden method. Table VI shows the results of the study. The results suggest that such an indexing scheme will not be very effective. Although we can use the indexing scheme to correctly recommend method calls, there are a number of cases where similar training examples may not be located inside overridden methods with same name. Figure 7 shows an example of indexing method calls by

Table V. Comparing performance (percentage of correctly predicted method calls) of CSCC at two different settings. The default setting does not explicitly include method name to both contexts, but the second setting does. Here, we only test those method calls that are located in overridden methods.

	Subject systems			
	Eclipse		NetBeans	
Recommendations	Default	Include overridden method name (%)	Default	Include overridden method name (%)
Top-3	80.93	83.68	81.36	82.61

Table VI. Percentage of correctly predicted method calls that are called in the overridden methods (for the top-3 proposals).

Recommendations	Subject systems			
	Eclipse		NetBeans	
	Without index %	With index %	Without index %	With index %
Top-3	83	73.4	83.10	70.9

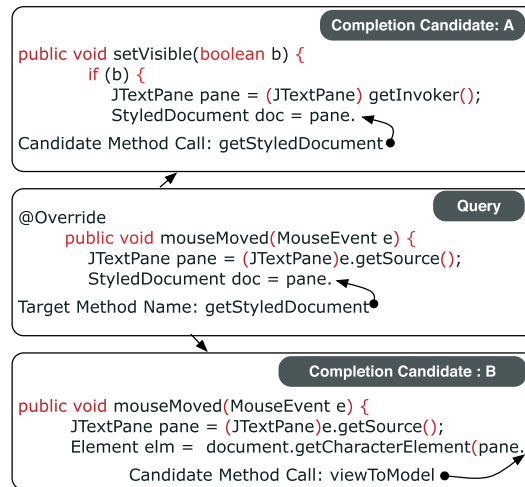


Figure 7. Indexing method calls by enclosing method name can lead to the wrong recommendation.

enclosing method name that can lead to wrong recommendation. Here, the target method call is *getStyledDocument*. The top and the bottom examples represent two completion candidates. Although the enclosing method name of the completion candidate B matches with that of the query code (code under development, shown in the middle of the figure), the context does not match. Instead the correct match should be the completion candidate A whose enclosing method name does not match with that of the query code.

4.4. Comparison with code recommenders

Code Recommenders [38] is a more advanced Eclipse plugin evolved from BMN. It utilizes a model trained using a set of code examples to provide intelligent code completion. When a developer invokes code completion in the IDE, Code Recommenders collect the current usage context and then looks in its model for possible matches to complete the code. Because the model is proprietary, we cannot train it with new code examples. Furthermore, because we did not have access to its internal API to obtain completion proposals automatically, we could only perform a manual comparison instead. Our comparison indicates that CSCC performed better than Code Recommenders. Our comparison is limited in scale because of its manual nature. Extensive evaluation may be possible in future if Code Recommenders becomes more open.

From the code examples in a book on the Java Swing framework [39], we randomly selected 309 Swing/AWT method calls as our test cases. We enabled the Code Recommenders intelligent call completion in an Eclipse IDE. Then for each test case, we manually opened the corresponding file in the IDE, removed any code after the target object and tried to complete the method call by typing a dot (.) after the object name. We recorded the list of completion proposals suggested by Code Recommenders and determined the rank of the target method name in that list.

To obtain the performance result for CSCC, we trained CSCC with the remaining examples and tested CSCC against the 309 selected method calls. CSCC achieves better result than Code Recommenders. The precision, recall and F-measure for Code Recommenders are 62%, 76% and 68%, and for CSCC 92%, 82% and 87% respectively.

4.5. Runtime performance

To be useful, code completion must be done at near real-time in order to not interrupt a developer's flow of coding. Thus, to study the time required to suggest completion proposals, we measured the runtime of the first and last two steps of CSCC. The first step is responsible for building a candidate method call database and the last two steps are about recommending completion proposals. All experiments were performed on a computer running Ubuntu Linux with a 3.40 GHz Intel Core i7 processor and 10 GB of memory.

As shown in Table VII, we provide runtime data for the Eclipse subject system where the model is built using 40,863 method calls (column two) and the running time is the time required to test all 4540 queries (column three). As expected, the first step takes the most time but the database needs to be built only once. On average, it takes 1.94 ms to compute the completion proposals for each method call, which is negligible.

To understand the benefits of using the inverted index structure, we also developed a variant of our algorithm without using the inverted index structure and measured the runtime again. The result is summarized in the second row of Table VII. While the model generation time reduces slightly, the code completion running time increases considerably. Using the inverted index structure not only

Table VII. Runtime performance of CSCC generating a database of 40,863 method calls (column 2) and performing 4540 code completions (column 3) for the Eclipse system.

Setup used	Database generation time	Code completion time
CSCC (with inverted index)	8066 ms	8998 ms (Avg.1.94 ms)
Without inverted index	8000 ms	12,888 ms (Avg.2.77 ms)

reduces the runtime of the algorithm, but also improves the result slightly by eliminating many irrelevant mapping candidates.

5. DISCUSSION

5.1. Why does CSCC consider four lines as context?

For an API method call, we use four lines prior to the method call to determine the overall context. The number four is determined experimentally as follows.

For this experiment, Eclipse is used as a subject system. We collect all SWT method calls and randomly select 10% for testing. The remaining 90% of calls are used to train CSCC. Next, we run the algorithm 10 times by varying context line numbers from one to ten. The higher the context line number, the larger the context size, which results in an increase in computation time. We need to keep the number of context lines as low as possible without impacting performance. Figure 8 shows the accuracy of CSCC at various context line numbers. From Figure 8, we can see that at the beginning, the number of correctly predicted method calls drops when we increase the context size from one to two lines. However, we observe a sharp increase from that point for increasing the context size. When the context size increases to more than four lines there is no significant change in the number of correct predictions. Therefore, we set the context size to four lines.

5.2. Impact of different context information

We evaluate the impact of CSCC's various context information on the performance of method call prediction. We run an experiment on NetBeans, our largest Swing/AWT system.

Table VIII shows the percentage of correctly predicted method calls for different combinations of context information. In the first row, we model the overall context considering those methods that were previously called on the receiver object but without the enclosing method name. Next, we consider a variation of the previous model that takes into account enclosing method name. For the above two models, we neither consider any line context nor put any limit on context size. But all the models in the following five rows use overall context to recommend completion proposals. The third row corresponds to a model that only considers those method names that were previously called within four lines of distance on the same receiver object of the target method call. The fourth row represents a model that in addition to the above information, also considers the line context. The fifth row corresponds to another model that in addition to the previous information, also considers any other method names located within four lines of distance. The model in the sixth row takes into account any type names (class or interface names) appearing as part of class instance creation plus the previous information. Finally, the last row implements the complete CSCC, which also includes any Java keywords except access specifiers.

According to the results, CSCC in the last row achieves the highest accuracy. It is also clear from the table that the performance of CSCC is increasing with the addition of additional context information.

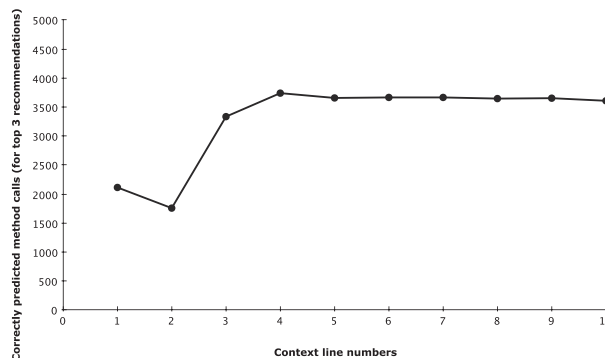


Figure 8. The number of correct predictions at different context size.

Table VIII. Sensitivity of performance for different context information.

Model	Correctly predicted method calls (%)			
	Top-1	Top-3	Top-5	Top-10
Rec. method calls	46.5	69.5	77.5	82.5
Rec. method calls + enclosing method	46.6	68.7	76.9	81.8
Rec. method calls (within four lines)	33	60.20	76	84.80
Rec. method calls (within four lines) + line context	34.8	61.62	76.26	84.89
Previous factors + other method calls	58	78	83	87
Previous factors + type name	62	82	86	89
Previous factors + keyword (CSCC)	64	84	88	90

Because adding the enclosing method name does not improve performance significantly (compare the first two rows), we did not include enclosing method name in CSCC. Among various additional information we considered, method names and the type names (appears in the class instance expression) contributed the most. Although the addition of the line context improves the overall performance by only around 1%, during our manual investigation we found that in those small number of cases, overall context differs considerably, so line context complements the overall context in this case. Surprisingly, adding keyword names did not improve the performance significantly. We analyzed some cases manually and found that while they are effective, their effect diminishes in the matching process because of the presence of a large number of methods, and class/interface names in the context.

5.3. Effectiveness of the technique in cross-project prediction

We performed another experiment to evaluate the effectiveness of CSCC in cross project prediction. For this experiment, we considered Swing/AWT library method calls for four subject systems. This includes NetBeans, JEdit, ArgoUML and JFreeChart. We followed the approach described by Nguyen et al. [9]. We divide the data set of each system into ten different folds. For each system, we determine the precision and recall values as follows. For each fold of a system, we trained with nine other folds of the same system plus code examples from all other systems and then perform testing on the remaining fold. We then calculate the average of precision and recall values. To make the results comparable with BMN, we only provide prediction results for local variable method calls. Table IX shows the results of our method call prediction.

CSCC once again performs better than other techniques. There are two important lessons to be learned from the result. First, both precision and recall of CSCC either slightly increase or are consistent with those of Table I, indicating that CSCC can recommend correct completion proposals even when the training model contains examples from different systems. As long as we have relevant usage contexts, CSCC can find them and can recommend completion proposals. Second, despite the considerable increase in the size of the training model, we did not notice significant improvement in performance. This seems to be consistent with Hindle et al.'s finding that the degree of regularity across project is similar to that in a single project [5].

5.4. Performance of CSCC for other frameworks or libraries

We already evaluated the performance of CSCC for API method call completion using two libraries, SWT and Swing/AWT. Both of them are used for developing graphical user interfaces. Thus, a threat to the validity of the study is that the performance of CSCC may be different for a different framework or library. We were interested to see whether the performance of CSCC is stable across different frameworks or libraries. For answering the question, we conduct another study using java.io and java.util method calls using two of our largest subject systems. The objective and usage pattern of these library method calls are different than those used for developing graphical user interfaces. The first one provides various API method calls to support system input and output, serialization and the file system. The second one contains various utility classes to facilitate working with date, time, collections and events.

Table IX. Cross-project prediction results. P, R and F refer to precision, recall and F-measure, respectively.

Subject systems			FCC (%)	BMN (%)	CSCC (%)
NetBeans	Top-1	P	39	46	69
		R	100	90	98
		F	56	61	81
	Top-3	P	64	77	84
		R	100	90	99
		F	78	83	91
JEdit	Top-1	P	57	70	66
		R	100	84	98
		F	73	76	79
	Top-3	P	69	85	83
		R	100	84	98
		F	82	84	90
ArgoUML	Top-1	P	48	48	54
		R	100	85	100
		F	65	61	70
	Top-3	P	65	68	77
		R	100	85	100
		F	79	76	87
JFreeChart	Top-1	P	43	44	68
		R	100	89	100
		F	60	59	81
	Top-3	P	74	85	88
		R	100	89	100
		F	85	87	94

Table X shows comparison results of CSCC with five other state-of-the-art tools for the java.io API method calls. Results from the study suggest that both ECCAlpha and ECCRelevance perform poorly in the study. While the recall of BMN is lower than FCC, it achieves higher precision than FCC for the top recommendation and for the Eclipse system. In all other cases FCC performs better than BMN for both subject systems. Interestingly, BCC performs better than both FCC and BMN. While the precision ranges from 47 to 58% for the top recommendation, the value increases to 70–81% for the top three recommendations. CSCC mostly achieves better result than any other techniques. For example, for the top position and for the Eclipse system CSCC achieves 15% more precision value (11% more F-measure value) than its closest competitor. For the NetBeans system, CSCC achieves at least 14% higher F-measure value than any other technique, 9% for the top three recommendations.

Table XI shows comparison results for the java.util API method calls. ECCAlpha, ECCRelevance and BCC perform poorer than the other completion systems. Similar to the previous result CSCC performs better than other code completion systems. For the top recommendation and for the Eclipse system, CSCC achieves a minimum of 25% more precision value (15% for the top three recommendations) than any other techniques. The technique also achieves at least 25% more recall value (10% for the top three recommendations). Although for the NetBeans system the difference is not that much, CSCC still performs the best.

5.5. Using bottom lines for building context

In the previous experiments we assume a top-down programming approach and we ignore the presence of any code after the tested method call. Although our approach is consistent with the previous study [4] we cannot guarantee that the code was developed in that way. It may be the case that the developer copied the entire piece of code from a different place and performed edit operations. It may also be the case that during the code review process a developer replaced a line or changed a method call on the line with a different one. In such cases, we may leverage the bottom lines of code for building context. Because of the lack of complete edit history we were unable to find those method calls only. However, an alternative approach can be to go for the best situation, that is use the bottom lines of

Table X. Evaluation results of code completion systems for the java.io method calls.

Measurement	Code completion systems	Subject systems					
		Eclipse			NetBeans		
		Top-1	Top-3	Top-10	Top-1	Top-3	Top-10
Precision	ECCAlpha	0.038	0.10	0.19	0.036	0.10	0.24
	ECCRelevance	0.13	0.27	0.29	0.14	0.30	0.57
	BCC	0.58	0.81	0.96	0.47	0.70	0.94
	FCC	0.47	0.72	0.91	0.38	0.63	0.89
	BMN	0.53	0.75	0.77	0.34	0.79	0.80
Recall	CSCC	0.73	0.89	0.95	0.65	0.84	0.97
	ECCAlpha	1	1	1	1	1	1
	ECCRelevance	1	1	1	1	1	1
	BCC	1	1	1	1	1	1
	FCC	1	1	1	1	1	1
F-Measure	BMN	0.89	0.89	0.89	0.73	0.73	0.73
	CSCC	1	1	1	0.99	0.99	0.99
	ECCAlpha	0.073	0.18	0.32	0.07	0.18	0.39
	ECCRelevance	0.23	0.43	0.32	0.25	0.46	0.73
	BCC	0.73	0.90	0.98	0.64	0.82	0.97
	FCC	0.64	0.84	0.95	0.55	0.77	0.94
	BMN	0.66	0.81	0.83	0.46	0.76	0.76
	CSCC	0.84	0.94	0.97	0.78	0.91	0.98

Table XI. Evaluation results of code completion systems for the java.util API method calls.

Measurement	Code completion systems	Subject systems					
		Eclipse			NetBeans		
		Top-1	Top-3	Top-10	Top-1	Top-3	Top-10
Precision	ECCAlpha	0.18	0.21	0.58	0.087	0.11	0.27
	ECCRelevance	0.34	0.43	0.76	0.17	0.21	0.32
	BCC	0.38	0.61	0.71	0.27	0.28	0.35
	FCC	0.36	0.74	0.97	0.73	0.86	0.95
	BMN	0.50	0.77	0.79	0.82	0.91	0.91
Recall	CSCC	0.75	0.92	0.97	0.87	0.94	0.96
	ECCAlpha	1	1	1	1	1	1
	ECCRelevance	1	1	1	1	1	1
	BCC	1	1	1	1	1	1
	FCC	1	1	1	0.98	0.98	0.98
F-Measure	BMN	0.91	0.91	0.91	0.88	0.88	0.88
	CSCC	0.99	0.99	0.99	0.98	0.98	0.98
	ECCAlpha	0.30	0.35	0.73	0.16	0.20	0.43
	ECCRelevance	0.51	0.60	0.86	0.29	0.35	0.48
	BCC	0.55	0.76	0.83	0.43	0.44	0.52
	FCC	0.53	0.85	0.98	0.84	0.92	0.96
	BMN	0.65	0.83	0.85	0.85	0.89	0.89
	CSCC	0.85	0.95	0.98	0.92	0.96	0.97

code already developed and test whether considering those lines with the top lines improve the performance of our code completion systems.

For this experiment, we develop a new version of CSCC that uses both top and bottom four lines including the line in which the method call appears to generate the usage context (Figure 9). For evaluation, we use two of our largest subject systems, Eclipse and NetBeans. For Eclipse, we collect the API method calls for the SWT library and for NetBeans we collect AWT and Swing API method calls. We then use the ten-fold cross validation technique to compare the performance of new version of CSCC with the old one.

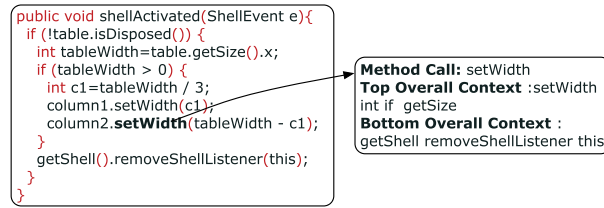


Figure 9. An example of top and bottom overall context. When we use both top and bottom context, we concatenated the terms appearing in the bottom context to the terms appearing in the top context.

Table XII compares the correctly predicted method calls for two different forms of CSCC. Here, the term Top Context refers to the original version of the technique. The term Both Context refers to the modified version of the technique that considers both top and bottom four lines to construct usage context of method calls. Results from the study suggest that considering the bottom four lines does not improve results for top three or top ten recommendations. However, we observe improvement for the top recommendation. Although we find only 1% improvement of accuracy for the Eclipse system, the number increases to 6% for the NetBeans system. This suggest that using bottom four lines for building context can be useful.

6. EXTENDING CSCC FOR FIELD COMPLETION

After releasing CSCC as an Eclipse plugin we received a number of feedbacks from the users. Many of them asked to extend the automatic code completion support for fields also. During our investigation we find that field completions are not trivial because of the possibility of a large number of choices. The objects we instantiate from different libraries and frameworks often contain a large number field variables. Before calling methods on those objects we either assign values to those fields or we may access them to check preconditions. Many of these fields are also constants. The problem is that there are a large number of them and it is difficult for a developer to remember each of them. Although classes define or inherit a large number of field variables from other classes, objects instantiated from those classes only use a few of them in practice.

Fortunately, many of these field variables are meant to be used in distinct contexts. For example, when we add a swing component to a container we need to state a field constant indicating the location of the container where the component needs to be added. BorderLayout is a popular layout manager that uses static field variables to represent various locations of a container. It supports both absolute and relative positioning constants but mixing them can result in an unpredictable result. Automatic code completion support can help in this regard by suggesting the correct positioning constants. As another example, the java.awt.event class contains 78 field variables (67 of them are used as field constants). However, depending on the context of using an event object, only a few of them are meant to be accessed or used. For instance, while writing a piece of code for mouse handling we need to access an event object to detect the mouse button pressed by a user. There are only five field constants of the event object that can be used to determine different states of a mouse

Table XII. The number of correctly predicted method calls using two different forms of usage context.

Recommendations	CSCC			
	Eclipse		NetBeans	
	Top context %	Both context %	Top context %	Both context %
Top-1	62	63	64	70
Top-3	80	81	85	86
Top-10	90	90	91	91

and other field constants are simply irrelevant in this context. Automatic code completion support can help in this regard by recommending relevant field constants in top positions.

Figure 10 shows an example of a field access (see the top figure). The objective of the method is to display information about the key that generates the event. Although the key event object has a `getChar()` method, we can only rely on that if the event is a key typed event. Thus the event id is checked using an if condition to determine whether the generated event is a key typed event. An intelligent code completion system should suggest the correct completion proposal in the top places. However, the default code completion systems of Eclipse did not perform well for this case. ECCRelevance suggests the target code completion proposal in the tenth position and ECCAlpha performs the worst, suggesting the target proposal in the 44th position. Developers typically call the `getId()` method to collect the key id before using the field access. CSCC can collect this information as usage context and can recommend the correct field access in the top position.

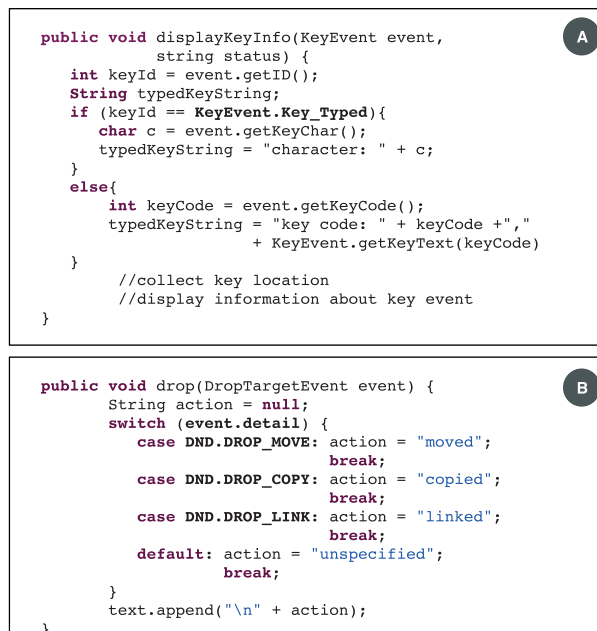
6.1. Changes made

Extending CSCC to support field completion was easy because of the simplicity of the technique. Recall that the algorithm works in three steps where the first step deals with collecting usage examples for target code completion. We change this step so that CSCC mines code examples to identify and collect usage context of field accesses. The usage context of a field access consists of the same information we use to capture the method call usage context.

We again use a two-level indexing scheme where the type name of the receiver object is used to group all the fields that are used with that type and we use an inverted index structure to organize the group of field accesses. When a developer requests for a field completion, we first determine the candidates for the field completion and then synthesize the results to recommend the top-3 field names. These steps are identical to those we described earlier for API method completion. In summary, CSCC did not require major changes to support automatic field completion.

6.2. Evaluation procedure

We evaluate field completion of CSCC with four other state-of-the-art code completion systems. For a given subject system, we determine locations of all field accesses where the receiver type matches with



```

public void displayKeyInfo(KeyEvent event,
    string status) {
    int keyId = event.getID();
    String typedKeyString;
    if (keyId == KeyEvent.Key_Typed){
        char c = event.getKeyChar();
        typedKeyString = "character: " + c;
    }
    else{
        int keyCode = event.getKeyCode();
        typedKeyString = "key code: " + keyCode + ","
            + KeyEvent.getKeyText(keyCode)
    }
    //collect key location
    //display information about key event
}

```

```

public void drop(DropTargetEvent event) {
    String action = null;
    switch (event.detail) {
        case DND.DROP_MOVE: action = "moved";
        break;
        case DND.DROP_COPY: action = "copied";
        break;
        case DND.DROP_LINK: action = "linked";
        break;
        default: action = "unspecified";
        break;
    }
    text.append("\n" + action);
}

```

Figure 10. Examples of field accesses (highlighted in bold).

the type name of the target framework or library. We then apply ten-fold cross validation technique to collect evaluation results. Next, we use the precision, recall and F-measure to measure the performance of each algorithm. These are same measures we used earlier for API method completion. The only change is that instead of method calls we now consider field accesses.

The code completion systems we consider in this study besides CSCC are ECCAlpha, ECCRelevance, FCC and BMN. ECCAlpha and ECCRelevance are the two default code completion systems that also support automatic field completions. We include FCC in this study that sorts field accesses based on their frequency in the training data. We exclude BCC from this study because the current implementation of the tool does not support the field completion. We also include BMN in this study to see whether the usage context BMN used to recommend method calls can be used to recommend field accesses.

We consider two APIs (SWT and Swing/AWT) and all eight subject systems we used previously for evaluating method call completion. Because developers request for a field completion in the same way of a method completion (for example, in Eclipse this can be done by typing a dot after a receiver name), we train each technique using both API field and method calls, but we test them for field accesses only. For the four subject systems (Eclipse, Vuze, Subversive and Rswl), we collect all SWT field and method calls. We collect Swing/AWT field and method calls for the remaining four subject systems.

6.3. Evaluation results

Table XIII shows the precision, recall and F-measure values for five code completion systems using Eclipse and NetBeans as the subject systems. Unsurprisingly, both ECCAlpha and ECCRelevance did not perform well in our field completion study. For example, for the Eclipse system, ECCAlpha performs the worst. ECCRelevance becomes the second last, and it achieves better precision, recall and F-measure values than ECCAlpha. Both FCC and BMN perform better than the default code completion systems of Eclipse. BMN achieves higher accuracy for the top position comparing to the FCC. While the precision of FCC is 52% for the top-3 positions, BMN achieves 54%. We also do not observe much difference in recall and F-measure values. CSCC achieves the highest F-measure value than any other code completion systems for the Eclipse system. The technique achieves 12% higher precision value than its closest competitor for the top-3 positions and the recall value is also high. For the remaining three subject systems that uses SWT API, CSCC performs better than any other techniques considered in this study. While it has precision 73–83%, the recall is 94–95%.

In general, for the remaining four systems that use Swing/AWT API, CSCC holds its position. It has a precision of 61–85% and recall 87–96%. While for the top position CSCC achieves the highest precision value, FCC achieves comparable performance as we increase the number of recommendations. The performance of CSCC is also less significant compared with FCC this time. This is because of the fact that the test cases have less possible completion candidates for systems that use Swing/AWT API compared to those that use SWT. For example, NetBeans has less number of completion candidates per query than the Eclipse system (see Figure 11). While BMN achieves precision value similar to that of the previous four systems, the recall value drops, largest for the NetBeans system. After investigation we found that for the NetBeans system a considerable number of test case receivers (almost 50%) are other than simple name type and BMN could not make any recommendations for those cases.

6.4. Training with method calls and field accesses together

In all the previous experiments, we train example-based code completion systems using only method calls or field accesses. Therefore, given a receiver type the code completion systems retrieve either method names or field accesses, not both. However, in an integrated development environment (such as in the Eclipse IDE), when a developer requests for a code completion by typing dot (.) after a receiver name, the target can be a method call or a field access. Therefore, it is required to train code completion systems using both method calls and field accesses. However, given a receiver type, the

Table XIII. Evaluation results of code completion systems for field accesses.

Subject systems	Precision					Recall					F-Measure				
	ECCAlpha	ECCRel	FCC	BMN	CSCC	ECCAlpha	ECCRel	FCC	BMN	CSCC	ECCAlpha	ECCRel	FCC	BMN	CSCC
Eclipse	Top-1	0.01	0.09	0.24	0.29	0.39	1	1	0.95	0.95	0.02	0.16	0.38	0.44	0.55
	Top-3	0.02	0.23	0.52	0.54	0.70	1	1	0.95	0.95	0.04	0.36	0.69	0.69	0.81
	Top-10	0.19	0.45	0.74	0.74	0.89	1	1	0.95	0.95	0.31	0.62	0.85	0.83	0.92
Vuze	Top-1	0.02	0.09	0.28	0.28	0.42	1	1	0.91	0.94	0.03	0.16	0.44	0.42	0.58
	Top-3	0.03	0.22	0.56	0.54	0.73	1	1	0.91	0.94	0.05	0.36	0.71	0.68	0.83
	Top-10	0.18	0.42	0.77	0.72	0.92	1	1	0.91	0.94	0.30	0.60	0.87	0.81	0.93
Subversive	Top-1	0.00	0.03	0.31	0.30	0.40	1	1	0.89	0.95	0.01	0.05	0.48	0.45	0.57
	Top-3	0.01	0.06	0.58	0.60	0.73	1	1	0.89	0.95	0.09	0.11	0.74	0.71	0.83
	Top-10	0.20	0.41	0.82	0.74	0.94	1	1	0.89	0.95	0.34	0.58	0.90	0.81	0.94
Rsowl	Top-1	0.01	0.07	0.22	0.14	0.46	1	1	0.82	0.94	0.01	0.13	0.37	0.24	0.62
	Top-3	0.02	0.15	0.50	0.28	0.83	1	1	0.82	0.94	0.04	0.27	0.67	0.42	0.88
	Top-10	0.07	0.23	0.78	0.58	0.94	1	1	0.82	0.94	0.13	0.38	0.88	0.68	0.94
NetBeans	Top-1	0.11	0.30	0.35	0.21	0.43	1	1	0.46	0.95	0.19	0.46	0.51	0.30	0.59
	Top-3	0.16	0.50	0.60	0.49	0.71	1	1	0.46	0.95	0.28	0.66	0.75	0.48	0.82
	Top-10	0.38	0.79	0.90	0.87	0.90	1	1	0.46	0.95	0.55	0.88	0.95	0.61	0.93
JEdit	Top-1	0.04	0.19	0.21	0.26	0.32	1	1	0.69	0.78	0.08	0.31	0.35	0.37	0.45
	Top-3	0.07	0.46	0.54	0.57	0.61	1	1	0.69	0.78	0.13	0.63	0.70	0.62	0.68
	Top-10	0.35	0.69	0.79	0.72	0.75	1	1	0.69	0.78	0.52	0.81	0.88	0.70	0.76
ArgoUML	Top-1	0.01	0.16	0.26	0.25	0.31	1	1	0.70	0.87	0.01	0.27	0.41	0.36	0.46
	Top-3	0.03	0.39	0.60	0.56	0.65	1	1	0.70	0.87	0.05	0.57	0.74	0.62	0.74
	Top-10	0.21	0.71	0.81	0.71	0.85	1	1	0.70	0.87	0.35	0.83	0.89	0.70	0.86
JFreeChart	Top-1	0.32	0.40	0.48	0.47	0.56	1	1	0.90	0.96	0.48	0.57	0.65	0.61	0.71
	Top-3	0.39	0.59	0.75	0.71	0.85	1	1	0.90	0.96	0.56	0.74	0.85	0.80	0.90
	Top-10	0.57	0.66	0.95	0.90	0.97	1	1	0.90	0.96	0.73	0.79	0.97	0.90	0.96

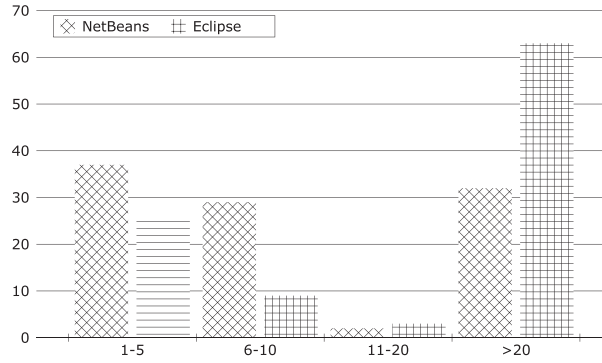


Figure 11. Distribution of test cases in different candidate groups. We group the test cases into four categories based on the number of completion candidates. X-axis positions four different groups, and the Y-axis value refers to the percentage of test cases in a group.

system can retrieve both method calls and field accesses whose receiver type matches with the target receiver type. This larger set of code completion candidates can affect the performance of code completion systems. To determine how this affects their performance, we conduct the experiment of field completion proposals for the largest two subject systems (Eclipse and NetBeans). This time we train each technique using both method calls and field accesses (i.e., setting A), and using only field accesses (i.e., setting B). The test data sets are identical to the previous study on evaluating field completion proposals. We exclude the default code completion systems of Eclipse (ECCAlpha and ECCRel) because they do not require any training.

Table XIV shows accuracy (in percentage) of three example-based code completion systems for two different subject systems. As shown in the table, there is a little or no effect of training code completion systems using only fields for the Eclipse system. When we investigate setting A further, it reveals that in general the query completion candidates retrieved for field completion overlap very little with those for method completion. For the NetBeans system, the completion candidates per query retrieved by the receiver type contain higher percentage of methods than the Eclipse system. That is why we notice changes in accuracy between two different settings for the NetBeans system. For example, FCC achieves more than 10% accuracy for the top three recommendations when training with only fields. BMN is no exception, and the least affected code completion system is the CSCC with only around 1% change in accuracy between two different settings. It has two important implications. First, CSCC can easily be adapted to different forms of recommendations with little changes. This is because of the simplicity in usage context construction, synthesizing examples and recommendation formulation of the technique. Second, despite the simplicity, the usage context used by the CSCC is generic in nature and can easily capture different usage scenarios. Thus, CSCC can support both method and field completions without making any significant changes.

Table XIV. The accuracy (in percentages) of correctly predicted field accesses for two different settings. For setting A, we train code completion systems using both fields and methods. For setting B, we only use fields for training. Both settings use fields for testing.

subject Systems	Code completion systems	Accuracy (%)					
		Top-1		Top-3		Top-10	
		A	B	A	B	A	B
Eclipse	FCC	23.09	23.10	51.53	51.56	74.77	74.79
	BMN	27.45	27.71	52.74	52.84	72.22	72.47
	CSCC	39.23	39.96	71.24	70.01	89.98	89.64
NetBeans	FCC	33.92	36.57	59.33	69.63	90	92.53
	BMN	10.92	14.25	25.36	28.27	45.31	47.09
	CSCC	43.17	44.58	70	71.12	88.34	89.43

7. COMPARISON WITH STATISTICAL LANGUAGE MODEL-BASED CODE COMPLETION TECHNIQUES

A statistical language model computes either a probability distribution over a sequence of words or the likelihood of occurring a word w_n given a sequence of prior words. These models show great promise in various software engineering tasks. This includes, but not limited to predicting source code comments [40], locating syntax errors in source code [41], and identifying coding conventions [42]. These techniques become successful because of the high degree of repetition and repetitiveness that exist in source code. A number of code completion techniques has been recently developed leveraging statistical language models. We are interested in evaluating the effectiveness of those techniques in completing method or field accesses and compare the result with CSCC. Unless otherwise specified we now use the term code completion to refer to both.

7.1. Statistical language models

We consider four statistical language model based code completion techniques and CSCC in this study. This includes N-gram, Cache LM, CACHECA and GraLan [5, 6, 10, 13]. N-gram identifies naturally occurring sequences of tokens in source code. Given a token sequence, N-gram identifies those tokens that tend to follow that token sequence in corpus. Cache LM improves the performance of traditional N-gram model by capturing locally repetitive token sequences using a cache component. We add Cache LM in our study to find the effectiveness of the cache component for code completion. CACHECA becomes the third technique in our study, and we include it to determine whether combining Eclipse suggestions with Cache LM leads to better result or not.

Unlike the previous three techniques that work at the lexical level, GraLan works at the statistical and data dependency level. We do not include SLANG in this study because GraLan showed better code suggestion accuracy than the technique and we have already included GraLan in this study. We do not consider SLAMC in this study because the tool is not available at the time of writing the paper. We include ECCRel in this study because CACHECA merges results of ECCRel with that of Cache LM. We are interested to identify the performance improvement of CACHECA over ECCRel.

For the N-gram, we use a trigram language model. For the Cache LM, we set the cache scope to the current file or related files (file cache). For cache size and order, default settings are used. We also enable the back-off technique for cache-LM. GraLan requires two parameters. The first parameter (θ) is used to limit the number of API calls that are used to discover context subgraphs. The second parameter (δ) is used to limit the number of context graphs. Same as in [6], we set both values to 8.

7.2. Evaluation procedure

We use the ten-fold cross validation to measure the performance of each technique. To train these techniques folds are created based on source files. This means that all framework method calls appear in a source file are either used for training or for testing. We create each fold in such a way that they contain equal number of framework method calls, although the number of source files can be different. We use the same eight subject systems we used in the previous experiment. We collect all SWT method calls and field accesses for the first four subject systems (Eclipse, Vuze, Subversive, and Rswl) and for the remaining four subject systems (NetBeans, JFreeChart, JEdit, ArgoUML) we collect all Swing/AWT method calls and field accesses.

To make the result comparable with Cache LM, we use the Mean Reciprocal Rank (MRR) and top-k accuracy in this study. For each framework method calls/field accesses in the test data, each technique produces a ranked list of suggestions. The reciprocal rank is calculated by taking the multiplicative inverse of a rank. Mean reciprocal rank is the average of reciprocal ranks for all n framework method calls in the test data. This can be defined as follows:

$$MRR = \frac{1}{n} \sum_{i=1}^n \frac{1}{rank_i} \quad (4)$$

where $rank_i$ denotes the rank of the i th test method call or field access. If a suggestion list does not contain the correct answer we use reciprocal rank 0. We also measure the top-k suggestion accuracy

for each technique, which is the ratio of the number of cases a technique produces the correct recommendation within the top-k recommendations over the total number of test cases. We report results for top-1, top-3 and top-10 positions.

7.3. Evaluation results

Table XV shows the results of our study. We compare CSCC with the four statistical language models (N-gram, CacheLM, CACHECA and GraLan) in terms of MRR, Top-1, Top-3 and Top-10 accuracies, by performing the directional Wilcoxon Signed Rank statistical test. The tests reveal that CSCC in most cases either outperforms or performs equally well as GraLan, which is the best performing among the statistical language models we examined. This is because of the fact that the technique leverages API usage graphs for recommending method calls or field accesses. For the first four systems, the accuracy of GraLan ranges from 57 to 71% for the top-3 recommendations. The value ranges from 54 to 76% for the bottom four systems.

As expected, N-gram obtains the lowest accuracy and MRR value among all statistical language model techniques. However, N-gram performs better than EccRel, indicating that prior code context leads to better result than using only static type information. Cache LM augments the N-gram model with a cache component that results in performance improvement. While the MRR of N-gram ranges

Table XV. Comparing CSCC with four statistical language model-based techniques (N-gram, Cache LM, CACHECA and GraLan). Because CACHECA merges results of ECCRel with those of Cache LM, we also include ECCRel in the comparison.

Subject systems		Code completion techniques					
		N-gram	Cache LM	EccRel	CACHECA	GraLan	CSCC
Eclipse	MRR	0.24	0.39	0.15	0.30	0.60	0.61
	Top-1	0.13	0.29	0.07	0.17	0.44	0.46
	Top-3	0.30	0.45	0.16	0.35	0.71	0.70
	Top-10	0.47	0.59	0.32	0.66	0.84	0.85
Vuze	MRR	0.34	0.44	0.13	0.26	0.59	0.60
	Top-1	0.23	0.35	0.06	0.13	0.34	0.45
	Top-3	0.41	0.51	0.14	0.27	0.59	0.69
	Top-10	0.57	0.65	0.29	0.64	0.80	0.80
SubVersive	MRR	0.36	0.43	0.10	0.34	0.58	0.68
	Top-1	0.25	0.32	0.03	0.18	0.42	0.52
	Top-3	0.44	0.52	0.06	0.44	0.63	0.76
	Top-10	0.57	0.64	0.30	0.69	0.83	0.86
Rsowl	MRR	0.36	0.48	0.20	0.43	0.49	0.59
	Top-1	0.24	0.36	0.12	0.28	0.30	0.42
	Top-3	0.45	0.57	0.23	0.51	0.57	0.64
	Top-10	0.61	0.71	0.33	0.77	0.75	0.72
NetBeans	MRR	0.43	0.52	0.33	0.48	0.70	0.68
	Top-1	0.31	0.40	0.19	0.31	0.50	0.54
	Top-3	0.54	0.62	0.40	0.59	0.76	0.75
	Top-10	0.66	0.74	0.61	0.82	0.88	0.86
JEdit	MRR	0.25	0.42	0.23	0.38	0.60	0.58
	Top-1	0.15	0.33	0.11	0.21	0.37	0.34
	Top-3	0.31	0.49	0.29	0.51	0.63	0.53
	Top-10	0.48	0.61	0.41	0.71	0.74	0.61
ArgoUML	MRR	0.30	0.45	0.31	0.47	0.54	0.59
	Top-1	0.20	0.35	0.15	0.32	0.32	0.41
	Top-3	0.37	0.53	0.37	0.57	0.54	0.56
	Top-10	0.48	0.64	0.66	0.78	0.69	0.65
JFreeChart	MRR	0.38	0.63	0.29	0.53	0.57	0.64
	Top-1	0.26	0.54	0.18	0.38	0.35	0.46
	Top-3	0.46	0.69	0.29	0.62	0.69	0.74
	Top-10	0.65	0.82	0.61	0.93	0.89	0.88

from 24% to 43%, for the CacheLM the value ranges from 39% to 63%. The number of correct recommendations for the top position is also much higher than the N-gram technique. However, Cache LM may fail to recommend those method calls or field accesses in top positions that are not locally repetitive. CACHECA combines the recommendations of cache LM with that of Eclipse. However, we observe that the strategy CACHECA uses to combine recommendations from two different sources affect the MRR value. CACHECA can recommend those cases where CacheLM fails but with a sacrifice of the MRR value. The MRR of CACHECA ranges from 26% to 53%. The accuracy of top-3 recommendations ranges from 27% to 62%.

In general CSCC performs the best comparing all other techniques. While the accuracy ranges from 46 to 74%, the MRR ranges from 58 to 68%. Despite the simplicity CSCC shows better performance than GraLan. When we investigate the reason we found that CSCC is able to recommend suggestions in more cases than GraLan. For example, there are cases where the prior context is empty for GraLan. Although GraLan cannot recommend in those cases CSCC can because it either finds context or considers empty context for recommending suggestions. We also observe a few cases where GraLan outperforms CSCC. For example, GraLan performs better than CSCC for all metric values for the JEdit system and three out of the four metric values of the NetBeans system. It would be interesting to investigate in future why GraLan does better in these cases.

Overall, CSCC shows better performance than statistical language model-based techniques. We investigate whether performance improvement of CSCC is statistically significant. We perform directional Wilcoxon Signed Rank Test for the MRR and accuracy at top-1, top-3 and top-10 recommendations. The null hypothesis is that there is no difference in MRR or accuracy values. The tests show that the difference in accuracy at top-1 is statistically significant at the p value of 0.05. However, the result is not statistically significant for MRR or accuracy at top-3 and top-10 recommendations. We conclude that CSCC outperforms GraLan for top-1 recommendation and performs equally well in other cases (MRR and accuracy at top-3 and top-10 recommendations).

7.4. How good CSCC is in recommending locally repetitive method calls or field accesses?

Previous experiment shows that a cache component is helpful to capture the locally repetitive method calls or field accesses that are otherwise difficult to detect by the N-gram model. This raises the following question: is CSCC capable of capturing those locally repetitive calls? We conduct a study to answer the question. We use the same ten-fold cross validation technique, but this time we collect those field accesses or method calls for testing that appear more than once in a file. We exclude those that appear only once in a file. We also exclude the first occurrence of those calls from testing that appear multiple times in a file because a cache language model may fail to recommend them because of a cache miss. We conduct the experiment using two of our largest subject systems, Eclipse and NetBeans. For the Eclipse system, we collect SWT field accesses and method calls. For the NetBeans system, we collect Swing/AWT library field accesses and method calls. Table XVI summarizes results of our study. For both systems, CSCC obtains similar or slightly better MRR metric values than Cache LM. While for the top position Cache LM achieves better result than CSCC (7% higher than CSCC for the Eclipse system and 4% higher for the NetBeans system), performance improves with the increase of number of recommendations. For example, CSCC obtains 5% higher accuracy than Cache LM for the top-3 positions. These indicate that CSCC is able to recommend locally repetitive field or method calls with good accuracy.

Table XVI. Comparison of CSCC with Cache LM considering locally repetitive method calls and field accesses.

Techniques	Eclipse				NetBeans			
	MRR	Top-1	Top-3	Top10	MRR	Top-1	Top-3	Top-10
CSCC	0.63	0.47	0.75	0.88	0.71	0.56	0.79	0.90
Cache LM	0.63	0.54	0.70	0.77	0.70	0.60	0.81	0.88

8. THREATS TO VALIDITY

In this section, we briefly describe several threats to the validity of this study.

First, we considered only two APIs during the evaluation of field completion. One can argue that the result may be different for a different framework or library. While it can be beneficial to test with additional libraries for other reasons, given that CSCC does not directly rely on these libraries, we believe that this is highly unlikely and that the results we obtain in this paper should largely carry over to additional libraries. Moreover, we tested CSCC for method call completion for two different libraries other than those two used in the evaluation of field completion. Results from that study also suggest that the performance of CSCC is not affected by the kind of library.

Second, we re-implement the BMN system because both the data and implementation of the technique are not available. We also re-implement GraLan because of lack of its implementation. However, instead of working from the scratch we reuse the implementation of Groum for identifying API usage graphs. Although we cannot guarantee that our replication of these techniques does not contain any errors, we have spent a considerable amount of time implementing and testing the technique to minimize the possibility of introducing errors.

Third, in this study we only consider top four lines to determine the context of a method call because we assume a developer is typing code in a top-down manner, which is consistent with previous studies [4]. However, it is also possible that a developer can edit existing code, in which case we can use both top and bottom lines of a method call to create context. Although we conduct an experiment to see the benefit of generating context using both top and bottom four lines, we cannot guarantee that the code was developed in the same way as we tested it because of the lack of a change based software repository.

Fourth, we compare CSCC with GraLan to determine the effectiveness of using a graph based statistical language model in method call completion. The API code suggestion engine of GraLan allows users to adjust two parameter values. One can argue that the accuracy of GraLan can be improved by adjusting those parameter values. We would like to point out that we use the same settings used in the GraLan study [6]. It might be possible to further fine-tune GraLan implementation by changing parameter values as a future work.

Fifth, to compare GraLan with CSCC we use the mean reciprocal rank (MRR) and the top-k accuracy metrics. We chose these metrics because they were used by previous code completion studies [9, 10], thus providing a common way of comparison. As future work we would also be able to compare CSCC with GraLan by considering training time, recommending time, total memory or external space usage.

Finally, statistical language model-based code completion techniques (i.e., N-gram, Cache LM and CACHECA) were originally developed for recommending a variety of tokens whereas CSCC is designed to recommend method calls and field accesses. Thus, statistical language model-based code completion techniques are typically evaluated considering all token kinds. Although it could be possible to adjust those techniques for recommending specifically method calls and field accesses, for example, by training the models with only method calls or field accesses, we did not do that. This is because we were interested in identifying the effectiveness of statistical language models in recommending method calls.

9. CONCLUSION

In this paper, we present a simple, efficient, scalable, context-sensitive code completion technique, called CSCC. CSCC mines previous code examples to recommend completion proposals. CSCC is simple because it is based on tokenization, instead of parsing or other more advanced analysis. It is efficient and scalable because of its ways of measuring context similarity: using *simhash* first as a coarse-grained but efficient filter, and LCS/Levenshtein distance second as a refined, more accurate similarity metrics. We have compared CSCC with other state-of-the-art code completion systems using two popular libraries. CSCC performed better than state-of-the-art static type or context-sensitive, example-based systems considered in our study. We then propose a taxonomy of method calls to identify the effect of different context elements on code completion techniques.

CSCC utilizes a simple form of usage context. It collects any method names, any keywords except access specifiers, any class or interface names that occurs within the top four lines including the line in which a method call appears prior calling a method. Although we initially developed and tested CSCC for method call completion, we later found that the usage context information is not limited to method calls only. We conduct a study to see whether CSCC can support the field completion using the same usage context information we collect for method calls. The results from the study suggest that CSCC can easily support field completions with little changes.

Finally, we compare CSCC with four statistical language models. CSCC not only outperforms all three techniques that work at the lexical level, but also in most cases performs better or equally well with GraLan, the state-of-the-art graph-based language model that leverages API usage graphs to recommend API elements. The code of CSCC and data files used in this experiment can be found online [43]. We also implement the technique as an Eclipse plugin which is also available online for public use [43].

ACKNOWLEDGEMENTS

We would like to thank Marcel Bruch and Andreas Sewe for providing useful comments and explaining APIs of Code Recommenders.

REFERENCES

1. Murphy GC, Kersten M, Findlater L. How are java software developers using the eclipse IDE? *IEEE Software* 2006; **23**(4):76–83.
2. Robbes R, Lanza M. How program history can improve code completion, in Proc. ASE, L'Aquila, Italy, 2008; 317–326.
3. Hou D, Pletcher DM. Towards a better code completion system by API grouping, filtering, and popularity-based ranking, in Proc. RSSE, Cape Town, South Africa, 2010; 26–30.
4. Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems, in Proc. FSE, Amsterdam, The Netherlands, 2009; 213–222.
5. Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software, in Proc. ICSE, Zurich, Switzerland, 2012; 837–847.
6. Nguyen AT, Nguyen TN. Graph-based statistical language model for code, in Proc. ICSE, Florence, Italy, 2015; 858–868.
7. Charikar MS. Similarity estimation techniques from rounding algorithms, in Proc. STOC, Montreal, Quebec, Canada, 2002; 380–388.
8. Hou D, Pletcher DM. An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion, in Proc. ICSM, Williamsburg, VI, 2011; 233–242.
9. Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A statistical semantic language model for source code, in Proc. FSE, Saint Petersburg, Russia, 2013; 532–542.
10. Tu Z, Su Z, Devanbu P. On the localness of software, in Proc. FSE, Hong Kong, China, 2014; 269–280.
11. Asaduzzaman M, Roy CK, Schneider KA, Hou D. CSCC: simple, efficient, context sensitive code completion, In Proc. ICSME, Victoria, BC, Canada, 2014; 71–80.
12. Pletcher DM, Hou D. BCC: enhancing code completion for better API usability, in Proc. ICSM, Edmonton, AB, 2009; 393–394.
13. Franks C, Tu Z, Devanbu P, Hellendoorn V. CACHECA: a cache language model based code suggestion tool, in Proc. ICSE, Florence, Italy, 2015; 705–708.
14. Raychev V, Vechev M, Yahav E. Code completion with statistical language models, in Proc. PLDI, Edinburgh, United Kingdom, 2014; 419–428.
15. Nguyen AT, Nguyen HA, Nguyen TT, Nguyen TN. GraPacc: a graph-based pattern-oriented, context-sensitive code completion tool, in Proc. ICSE, Zurich, Switzerland, 2012; 1407–1410.
16. Nguyen AT, Nguyen TT, Nguyen HA, Tamrawi A, Nguyen HV, Al-Kofahi J, Nguyen TN. Graph-based pattern-oriented, context-sensitive source code completion, in Proc. ICSE, Zurich, Switzerland, 2012; 69–79.
17. Holmes R, Murphy GC. Using structural context to recommend source code examples, in Proc. ICSE, St. Louis, MO, USA, 2008; 117–125.
18. Moity M, Faulring A, Stylos J, Myers BA. Calcite: completing code completion for constructors using crowds, in Proc. VLHCC, Leganes, Spain, 2010; 15–22.
19. Zhang C, Yang J, Zhang Y, Fan J, Zhang X, Zhao J, Ou P. Automatic parameter recommendation for practical API usage, in Proc. ICSE, Zurich, Switzerland, 2012; 826–836.
20. Hill R, Rideout J. Automatic method completion, in Proc. ASE, Linz, Austria, 2004; 228–235.
21. Lee YY, Harwell S, Khurshid S, Marinov D. Temporal code completion and navigation, in Proc. ICSE, San Francisco, CA, USA, 2013; 1181–1184.
22. Jacobellis J, Meng N, Kim M. Cookbook: in situ code completion using edit recipes learned from examples, in Companion Proc. ICSE, Hyderabad, India, 2014; 584–587.

23. Little G, Miller RC. Keyword programming in java, in Proc. ASE, Atlanta, Georgia, USA, 2007; 84–93.
24. Han S, Wallace DR, Miller RC. Code completion from abbreviated input, in Proc. ASE, Auckland, New Zealand, 2009; 332–343.
25. Omar C, Yoon YS, LaToza TD, Myers BA. Active code completion, in Proc. ICSE, Zurich, Switzerland, 2012; 859–869.
26. Introduction to information retrieval. (Available from: <http://www-nlp.stanford.edu/IR-book/>) (Accessed on February 2014).
27. Manku GS, Jain A, Sarma AD. Detecting near-duplicates for web crawling, in Proc. WWW, Banff, Alberta, Canada, 2007; 141–150.
28. Uddin MS, Roy CK, Schneider KA, Hindle A. On the effectiveness of simhash for detecting near-miss clones in large scale software systems, in Proc. WCRE, Limerick, Ireland, 2011; 13–22.
29. Bruch M, Schäfer T, Mezini M. On evaluating recommender systems for API usages, in Proc. RSSE, Atlanta, Georgia, 2008; 16–20.
30. The eclipse. (Available from: <http://www.eclipse.org/>) (Accessed on February 2014).
31. The Vuze. (Available from: <http://www.vuze.com/>) (Accessed on February 2014).
32. The Subversive. (Available from: <https://www.eclipse.org/subversive/>) (Accessed on February 2014).
33. The RSSOwl. (Available from: <http://www.rssowl.org/>) (Accessed on February 2014).
34. The NetBeans. (Available from: <https://netbeans.org/>) (Accessed on February 2014).
35. The jEdit. (Available from: <http://sourceforge.net/projects/jedit/>) (Accessed on February 2014).
36. The ArgoUML. (Available from: <http://argouml.tigris.org/>) (Accessed on February 2014).
37. The JFreeChart. (Available from: <http://sourceforge.net/projects/jfreechart/>) (Accessed on February 2014).
38. The code recommenders. (Available from: <http://www.eclipse.org/recommenders/>) (Accessed on February 2014).
39. Code Examples. (Available from: <http://examples.oreilly.com/jswing2/code/>) (Accessed on February 2014).
40. Movshovitz-Attias D, Cohen WW. Natural language models for predicting programming comments, in Proc. ACL, Sofia, Bulgaria, 2013; 35–40.
41. Campbell JC, Hindle A, Amaral JN. Syntax errors just aren't natural: improving error reporting with language models, In Proc. WCRE, Hyderabad, India, 2014; 252–261.
42. Allamanis M, Barr ET, Bird C, Sutton C. Learning natural coding conventions, In Proc. FSE, Hong Kong, China, 2014; 281–293.
43. The CSCC. (Available from: <http://asaduzzamanparvez.wordpress.com/csccl/>) (Accessed on February 2014).