Evaluating the Evaluations of Code Recommender Systems: A Reality Check

Sebastian Proksch, Sven Amann, Sarah Nadi, Mira Mezini
Software Technology Group
Technische Universität Darmstadt, Germany
{proksch, amann, nadi, mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

While researchers develop many new exciting code recommender systems, such as method-call completion, code-snippet completion, or code search, an accurate evaluation of such systems is always a challenge. We analyzed the current literature and found that most of the current evaluations rely on artificial queries extracted from released code, which begs the question: *Do such evaluations reflect real-life usages?* To answer this question, we capture 6,189 fine-grained development histories from real IDE interactions. We use them as a ground truth and extract 7,157 real queries for a specific method-call recommender system. We compare the results of such real queries with different artificial evaluation strategies and check several assumptions that are repeatedly used in research, but never empirically evaluated. We find that an evolving context that is often observed in practice has a major effect on the prediction quality of recommender systems, but is not commonly reflected in artificial evaluations.

CCS Concepts

•General and reference → Evaluation; •Information systems → Recommender systems; •Software and its engineering → *Software notations and tools;* **•Human-centered computing →** Design and evaluation methods;

Keywords

Empirical Study, Artificial Evaluation, IDE Interaction Data

1. INTRODUCTION

Research in the area of Recommendation Systems for Software Engineering (RSSE) regularly produces exciting ideas on how to automatically support developers with their daily coding and maintenance tasks. Examples include recommending the next syntactic token [19], links to related code snippets [15], and which windows to close [14]. To provide (quantitative) evidence that such tools are accurate and valuable, extensive evaluations are needed.

Conducting such evaluations is often challenging. Many evaluations of RSSE involve humans for realistic evaluations (e.g., [3, 9, 15, 20]). Unfortunately, conducting such controlled experiments is often infeasible due to the high cost in terms of both time and resources [13]. Other issues include the replicability of the experiment or privacy constraints when analyzing developer behavior. In addition, controlled experiments are based on selected use cases, which limits the generalizability of the results. To overcome these challenges, many researchers have resorted to *artificial evaluation strategies* that generate evaluation queries from released code. These strategies can overcome the drawbacks of controlled experiments with real developers. They are easy to conduct and scale well, allowing them to cover different scenarios. However, using them raises the following critical question: How realistic are they? In other words, do artificial evaluations actually reflect real-life usages? Given that the method for creating queries from released code greatly differs between the evaluation strategies we surveyed, it is important to understand how different decisions may influence evaluations. To the best of our knowledge, these questions have not been systematically and empirically investigated.

In order to compare artificial approaches to real approaches, it is necessary to have the appropriate ground truth. We use an existing IDE interaction tracker to capture a dataset of fine-grained history of source code directly in the development environment of developers. This dataset provides the necessary means for a realistic evaluation of source-based RSSE. In our experiments, we feed a method-call recommender, PBN [22], with queries extracted from the captured code changes and compare this realistic evaluation strategy with artificial approaches to answer two research questions:

RQ1: Do artificial queries have an effect on the measured prediction quality of a recommender?
RQ2: Do real queries have properties that are not reflected in artificial queries?

Our results show that artificial evaluations can be misleading, often suggesting a higher prediction quality than what would be achieved in practice. We show that the differences result from ignoring evolving context that is not captured in artificial queries. Our results help toolsmiths make informed decisions about the evaluation strategy best suited for their goals and understand implications of these decisions.

In summary, this paper makes the following contributions:

- We present a survey of related work to identify common evaluation strategies and classify them based on the design decisions they make.

- Using a tool that captures developers' interactions, we collect a set of real code changes from professional software developers, researchers, and students.
- We use this dataset to conduct an extensive experiment to evaluate the quality of artificial evaluation strategies and tradeoffs in their design space.

We publish the tools used in our experiments on the artifact page of this paper [4]. We also publish all parts of the dataset for which we have the required permission. This dataset can be used as a benchmark for evaluating future recommenders.

2. CURRENT EVALUATION TECHNIQUES

We survey source-based RSSE to understand how they are currently evaluated. We consulted a RSSE survey [26], the recent RSSE book [27], and several recent publications in premier software engineering conferences to identify related evaluation techniques. Because we use a source-based method-call recommender in our experiments, we focus the survey on the evaluations of related RSSE. While the most related evaluations would be those of other method-call recommenders, we relax the survey criteria and also consider other source-based RSSE to get a general overview of how evaluations are typically done. We include approaches that consider the structural and context information found in code (as opposed to techniques that treat code as text [8]). Based on this survey, we summarize existing evaluation strategies and motivate our problem statement accordingly.

2.1 Existing Evaluation Strategies

Many RSSE are being evaluated through controlled experiments that involve human subjects. To quantify the performance of the system, researchers analyze if participants successfully complete a task (e.g., [9]) or measure how long subjects take for completion (e.g., [13]). In other more qualitative evaluations, experts judge the usefulness of the proposals (e.g., [9, 15]) or the subjects rate their experience with the tool (e.g., [3, 28]). All these are valid evaluations and they often create additional qualitative insights.

However, controlled experiments also have their downsides. Their nature limits their scope and makes it hard to generalize the results. Many tools also work with external services (e.g., Q&A websites [20]), which hinders replication, or require very task-specific data like navigation information (e.g., [3]), which makes it hard to design appropriate tasks. Most importantly, designing a controlled experiment involving humans takes a lot of time. In addition, the risk of a failed experiment is also quite high, because a study cannot be simply replayed, when for example a bug is discovered after the fact. As a result, it is much more common to find artificial evaluations of RSSE in the literature, which motivated the research questions of our work. In the following, we will discuss several representatives of artificial evaluations that we found in the literature. The goal is not to present an exhaustive list of prior evaluations, but to introduce several high-level ideas by example.

Heinemann et al. present a method-call recommender [7]. In addition to considering the structure of a program, the approach considers identifiers, such as variable names. The recommender tokenizes source code into an event stream and learns a model of this stream. To evaluate the approach, they iterate over this event stream, predict every method invocation that they encounter, and measure the quality of the proposals. They assume linearity and do not include information that is found after the query point.

Zhang et al. propose a recommender that predicts parameters for method calls [28]. For evaluation, they use published source code and query the recommender at each observed parameter. Queries contain all observed information from the code with the exception of the parameter that is to be predicted. They also conducted a user study that analyzes the perceived usefulness and opinions of the participants to get qualitative feedback for their tool. However, they do not present quantitative data about the performance of the recommender or the correctness of the proposals.

Bruch et al. [2] propose a method-call recommender based on the Best Matching Neighbor (BMN) algorithm. Queries are automatically generated from API usages observed from code repositories. A query consists of a subset of the observed method calls in the API usage and the evaluation measures how well the recommender predicts the removed calls. The authors use two strategies to generate such queries: (1) a *no calls included* strategy that mimics the situation where a developer triggers code completion when she starts to implement a method and (2) a *first half* strategy that keeps only the first half of the method calls to mimic the situation where a developer triggers code completion after she wrote parts of a method.

Follow-up work replaces the BMN algorithm by a Pattern-Based Bayesian Network (PBN) [22] as the recommender engine. The evaluation also uses both the *no calls included* strategy and the *first half* strategy. In addition, queries are generated with a *random half* strategy that randomly selects which half of the method calls is kept to mimic that developers may not write code in a linear fashion. The random selection is repeated and the results are averaged.

GraPacc [18] recommends code snippets that are related to the current context. Patterns are mined from the source code of some Java projects to create the recommender, which is then evaluated on several other projects. For the evaluation, all method calls are extracted from the method bodies in the validation projects. These sets of methods are divided into two parts: the first part is used as a query for the recommender, the second part as the expectation. The evaluation measures the fraction of proposed method calls that are contained in the second part and the proposal. This evaluation technique is similar to the *first half* evaluation followed by Bruch et al. [2].

MAPO [29] is a miner and recommender for API usage patterns. The miner identifies API patterns in a large pool of released source code. In the IDE, the current context is matched against these patterns to retrieve related code snippets. The authors use code snippets selected from a tutorial book in the evaluation, which are considered correct and complete. In their queries, they use all context information and the first method call. Given such a query, the recommender suggests related code snippets, which are manually matched with the expectation.

Prospector [13] is a recommender that is queried with a tuple of two API types: an input type from the current context and a target type that the developer wants to obtain an instance of. Prospector returns a sequence of method calls that would return an object instance with the respective type. For the evaluation, the authors manually picked 20 example programming problems that they deem realistic and that Prospector is applicable to. For the queries, they always assume that the developer knows both types.

Table 1: Classification of existing artificial evaluation strategies. Columns show the identified query scenarios while rows show the selection strategies.

	0|M	**N|M**	**M-1|M**
Linear	[2,22]	[2,7,18,29]	[23,28]
Random		[5,13,22,23]	

Guervo et al. present InSynth [5], a tool to synthesize type-correct expressions. As such expressions can be complex structures that contain nested sub-expressions, the approach effectively recommends code snippets. For the evaluation, they manually create 50 query/expectation pairs as benchmarks, taken from several open-source projects. A query is a program snippet in which a single expression is removed for the benchmark. The evaluation measures whether InSynth can synthesize that expression again. Since the target expression is selected arbitrarily, the evaluation approach resembles the random removal approach used for PBN [22]. Unlike the PBN evaluation, the benchmark set is manually created rather than automatically generated.

Raychem et al. [23] develop a recommender that suggests multiple missing method calls in a piece of code. They traverse the syntax tree and reduce it to a sequence of method calls. Missing method calls are *holes* in this sequence. The recommender calculates the most likely sequences of method calls that fill the holes. Three kinds of queries are used for evaluation: (1) a single hole at the end of the program, (2) multiple holes that are manually introduced, and (3) one or more random holes that are automatically introduced. The first query strategy assumes linearity in code development; the other two assume non-linearity. The authors do not describe whether there is a limit on how many holes are introduced; our understanding is that only a small percentage of the method calls are removed for querying.

Note that the three last papers are somewhat different from the others, since they also include queries manually created by the tool smith. This is done to create realistic, meaningful queries that a real developer might trigger. However, since the creation of such queries is very subjective and not based on any input from actual developers, we consider these evaluations as artificial.

Some approaches conduct controlled experiments and capture all interactions of the subjects for later experiments. However, existing approaches either capture data that is not appropriate to build or evaluate source-based RSSE (e.g., [10,11]) or they do not perform automated evaluations and evaluate results manually (e.g., [16]). To the best of our knowledge, there have been no attempts to create a benchmark of developer interactions for an automated evaluation of RSSE before.

2.2 Problem Statement

Our survey of evaluation approaches suggests that artificial evaluations are more popular than real evaluations. This is not surprising, since real evaluations tend to be too expensive and time-consuming to be practical [13]. They usually also involve factors like privacy considerations and convincing developers to use a research prototype. While artificial evaluation strategies do not have the issues mentioned above, it is important to understand how close they are to real evaluations. To investigate this, the goal of this paper is to answer the two research questions posed in the introduction. Specifically, we evaluate different artificial evaluation techniques that employ different *query generation strategies* and compare them to a realistic evaluation. Our survey identified two factors that differentiate the artificial query generation strategies: the query scenario and the selection strategy.

Query Scenario. Any piece of code has *context information*. This includes structural context information such as the surrounding method or class, as well as information such as which methods have been called and in which order. The *query scenario* describes *how much context from the final code is kept* in the query. Most query-generation approaches focus on the target information they want to predict. For example, a method-call recommender focuses on the methods called in a particular context, while a parameter recommender focuses on the parameters of a particular method call. Given a final code snapshot of `M` items of target context information, the query-generation approaches range from removing all of this context (*0*|*M*) to "leave one out" (*M-1*|*M*), with shades in between, where the target context is partly preserved (*N*|*M*).

0|M In this case, all target context information is removed from the final state of the code, resulting in a minimal query. If the approach depends on specific information, e.g., the type of the variable on which code completion was triggered, this information is preserved. This strategy mimics the situation where developers are just starting to write code and may not know where to start. Creating such queries is straightforward since there are no ambiguities in what goes into a query.

N|M In this case, parts of the existing code is preserved. This mimics the typical development scenario, where developers implement some parts of a method, but potentially miss details for which they need the recommender's help.

M-1|M In this case, only one piece of information is removed. This mimics the case of developers who already implemented most functionality but only miss one part.

Selection Strategy. The *selection strategy* is the second differentiating factor, which determines *how partial information is selected* from the final context. It answers the question: Given a complete piece of code from a repository, which parts of it should be removed for querying? Several approaches assume a linear development of source code and remove the later parts of a method. Other approaches perform a random selection of the context or even repeat the random selection multiple times to cover different parts of the existing code.

Linear Assuming code is developed in a linear fashion greatly simplifies the evaluation and makes its implementation straightforward. However, there is no empirical evidence that developers actually code in this fashion and so it is unclear how realistic this assumption is.

Random For a thorough evaluation, several random sub-selections are made for a complete usage. The results are averaged to get one representative prediction-quality measure. The averaging adds an extra layer to the implementation. Heuristics may be needed to limit the number of selected queries, since the number of sub-selections can be quite large. Randomly selecting parts of the existing code to remove might also hide corner cases in which a recommender performs particularly well or badly.

Table 1 classifies the related work we presented in Section 2.1 along the two identified dimensions. Given the variability of artificial evaluation strategies, it is important to understand the impact of different choices, how they reflect real-life developer usage, and how they compare to a real evaluation

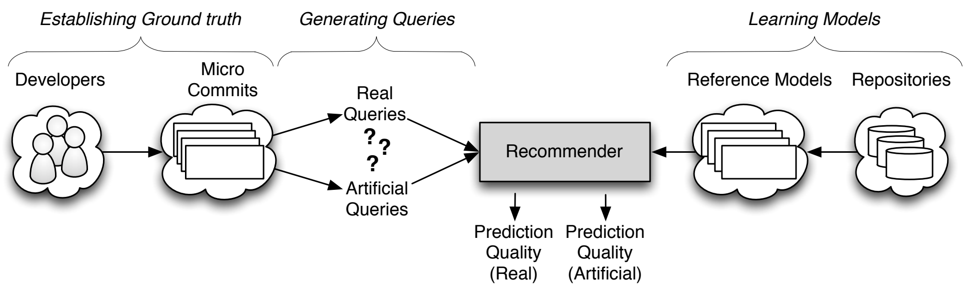

Figure 1: Overview of Evaluation-Comparison Strategy

strategy. Such knowledge is valuable in two ways: First, it helps to judge the validity of evaluation results reported in the literature. Second, it helps researchers make informed decisions about their evaluation strategies and how to interpret their evaluation results.

3. OVERVIEW OF EVALUATION SETUP

Figure 1 outlines the overall setup we use to compare the identified evaluation strategies. The idea is composed of two parts. First, we establish a ground truth, from which we generate different types of queries for the identified evaluation strategies. Second, we provide the different queries to a recommender and compare the quality of its proposals as a means to compare the different strategies.

To ensure fairness and comparability, we use the same recommender system for all evaluation strategies. The choice of the particular subject recommender system is less important, because the effect of any weakness or strength of the recommender will be equal across all evaluation strategies. We use the *Pattern-based Bayesian Network* (PBN) method-call recommender [22], because it provided us with a complete open-source evaluation pipeline. The core data structure used in PBN is an *object usage*, which contains information about the context in which the usage of a particular API type was observed (e.g., the enclosing method or the way the object is initialized), as well as the set of methods that have been invoked on that instance of the type.

To build the recommender, we create reference models for various API types (the *Learning Models* step in Figure 1). We used an open-source dataset of C# projects as input training data to create the reference models [21]. The dataset was created from 360 open source GitHub repositories and contains usage data for more than 560 unique APIs, each of which includes various types. We filter the dataset to the 407 types that appear in our ground truth and use the collected usages to build reference models for these types.

Since establishing the ground truth and generating the queries are the main steps in setting up our evaluation comparison experiments, we respectively dedicate Section 4 and Section 5 to the details of each of these steps.

4. ESTABLISHING THE GROUND TRUTH

To ensure a realistic and fair comparison, we first establish the ground truth about what queries developers actually perform and what the code looks like at that point. Thus, we are interested in capturing a development history of how method calls are added to the code under edit over time. This allows us to replay the development using the actual state of the code at query time, i.e., to simulate a controlled experiment with real queries. It also allows a comparison of the results for different recommenders.

The commit history obtained from a project's source control repository has been commonly used to obtain such a development history (e.g., Hassan and Holt [6]). However, it has also been shown that, on average, commits are created on every third day, with a high variance between users and the type of changes [12]. It has also been shown that version-control commits shadow many of the intermediate code changes [17]. This means that the version history found in public repositories of Open Source Software is too coarse grained for our purposes. Instead, we choose to capture more fine-grained code changes directly from the developer's Integrated Development Environment (IDE), similar to the idea previously proposed by Robbes and Lanza [24]. While both Hassan and Holt [6] and Robbes and Lanza [25] used some form of development history to improve recommenders, our work is different in that we use this development history to *compare different evaluation strategies* rather than use it to improve the recommender itself.

4.1 Creating the Tooling

To get more fine-grained code changes, we want to capture snapshots of the code under edit every time a change occurs. Such information is best captured directly from within the developers' IDE. Additionally, we want to capture interactions of the developers with the IDE's code-completion tool and store which method was selected from the list of proposals, if any. We combine the code snapshot and the timestamp, as well as the optional selection of a method proposal, if available. We call the collection of this information an *enriched micro commit.* We can use these micro commits to create real queries and to replay the recorded development history, including code completion, after the fact.

To collect such information, we extended FeedBaG, an open source instrumentation of Visual Studio that collects interactions of C# developers with their IDE [1]. We extended the instrumentation of the code completion and added a static analysis that extracts context information from the code under edit. Each time code completion is triggered by the developer (or when it pops up automatically), we create a snapshot of the source code under edit. Snapshots are stored in the form of *simplified syntax trees* [21], a lightweight format that also includes typing information and which supports markers for code completion trigger points.

4.2 Gathering the Data

FeedBaG's sources are publicly available, and the tool can be installed from within Visual Studio. Once a user has the tool installed, their interactions and micro commits are automatically captured. Users can then upload this captured data to our servers at any time through a provided dialog.

We first deployed our modified version of the tracking tool

Table 2: Contributed Events per Developer

Id	Type	# Days	# Queries	%
0*	Researcher	89	4888	20.6
1	Student	66	4625	19.5
2	Student	52	3162	13.3
3	Hobby Programmer	48	2096	8.8
4	Student	28	900	3.8
5	Student	32	771	3.2
...				
45	Unknown	4	10	<0.1
...				
55	Professional	3	2	<0.1
total		753	23,746	100.0

with *Company X* (cannot be named for privacy reasons) that develops tax and accounting-related software as well as in-house software for 50 years. It employs more than 1,600 developers, out of which more than 400 write programs in C#. Development projects span from small training examples to core-business applications. In addition, we advertised the project in several social media channels and during various conferences to widen our user base. Even before our active recruitment efforts, several open-source developers independently installed our tool after seeing it in Visual Studio's public plug-in repository. We also had several students install FeedBaG while they were developing different systems, including a game, web applications in ASP.NET, or their own Masters thesis project. Finally, one of the authors of this paper along with student assistants also had FeedBaG installed while they working on the tools used in this paper.

Our final data set, therefore, contains queries from a variety of users and projects, including industrial developers, open-source developers, researchers, and students. The individual participant contributions are listed in Table 2. Note that the table is cropped for brevity, but the complete list is available on the artifact page. The asterisk in the table marks the contributions of the involved author of this paper.

In total, we received submissions of captured data from 56 users. Out of these, 27 were industrial developers that provided 13% of our queries. The remaining users (with percentage contributions shown in parenthesis) were 8 students (45.9%), 4 researchers (21.5%), and 2 hobby programmers (9.6%). The remaining 15 users (10.0%) decided not to fill our (optional) profile information. The submissions cover 753 days and span over a period of 13 months, but not all users participated the whole time.

4.3 Post-processing the Data

There is still a gap between our collected ground truth and the input data required for PBN. The collected micro commits are file-oriented snapshots whose contents reflect a complete type declaration in a file (e.g., a class with all its methods and the corresponding method bodies), while the input data for PBN are object usages. To bridge this gap, we first sort the micro commits by time and declared type. As a result, we get the development history of a file. After this, we extract object usages for all types used in each micro commit. Since a micro commit represents a whole class, we extract object usages for several types in this step. Finally, we merge the resulting usages from all micro commits of the same user, group them by type and by enclosing method, and preserve the order to create a complete usage history.

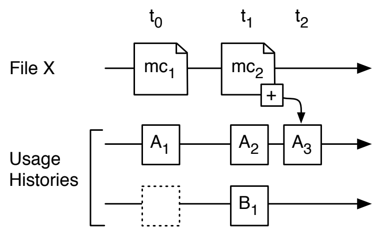

Figure 2: From Micro Commits to Usage Histories

Figure 2 illustrates an example. The file icons depict micro commits, while the squares represent object usages. The character in each square shows the type of that specific object usage; the index is only added for easier reference.

Assume that we have created two micro commits for file X that were captured at times t_0 and t_1. For the micro commit at t_1, we captured the information that a specific method was selected from the code completion proposals (depicted by the "+'). Assume that at time t_0, the code only contained usages of type A, while at time t_1, usages of type B were also added. Our usage extraction therefore finds a single object usage for type A in the first micro commit and two object usages for the types A and B in the second one.

In order to use all information that is contained in the usage history, we developed two strategies that transform implicit knowledge into explicit states in the usage history. Given all extracted usages, we can identify types that are used in a specific context, but for which we were not able to extract usages for all micro commits. In the example, we can derive that B was not yet used in the first micro commit, but added to the second one. To make sure that we include such usages that are created *from-scratch* (i.e., usages where we only know the object type and surrounding context but do not have any called methods yet) in our evaluation, we add an empty usage to B's history at time t_0.

Another corner case that needs to be handled is if the last micro commit in a usage history contains a selection. In this case, we will not actually see the effect of the selection in subsequent micro commits, because none exist. In order to preserve the selection for our evaluation, we create an artificial usage in which we merge the usage on which code completion was triggered with the selection result. Referring to the example in Figure 2, a method selection took place for the micro commit captured at time t_1. Assuming the completion trigger would have taken place on the object usage of type A and that $t + 1$ is the last micro commit in the history, we would create an object usage A_3 at time $t + 2$, which contains the selected method too.

After the extraction, our usage history may contain subsequent occurrences of the same object usage. This duplication may be due to several reasons. For example, consider the situation in which a developer invokes code completion, but then cancels it. This would result in two equal usage snapshots in the history. Another example is that the developer uses the types A and B in the same context. She adds multiple method calls to the object of type B, but leaves the object of type A untouched for a while. This would add several unchanged usage snapshots to A's history. We post-process the histories and remove such duplicates.

After applying the described transformations, our final ground truth set that is used in our experiments consists of 6,189 usage histories. On average, each usage history contains 4.6 snapshots, but a few outliers exist with a length of more than 700 usages. We manually inspected these cases and all inspected cases were examples in which the developer spent time in a specific context implementing an algorithm and working with the same type over and over again. Frequent additions and removals of the same methods bloat up the histories for these usages, e.g., adding and removing a log statement. However, outliers with more than 17 steps represent less than 2% of our collected data, so we did not introduce special handling for these cases.

Our collected usage histories cover 407 types used in 5,834 different method contexts. The dataset we used to train the recommender contains a total of 650,340 object usages for these types. Note that PBN models do not contain ordering information. While object usages only contain a *set* of method invocations, our extraction implementation guarantees that the order in which invocations are entered reflects the order in the source code. While this is irrelevant for object usages used in PBN models, it is important for generating order-dependent queries for some of the evaluation strategies we compare.

5. GENERATING QUERIES

In this section, we discuss how we use our collected ground-truth data set to generate queries that can be used to compare different evaluation strategies. As discussed in Section 2, artificial evaluations are based on released code found in a repository. This version is treated as the final state of the code and considered correct and complete. Artificial evaluations apply heuristics to approximate past states of this version, which are then used to generate queries. The final state serves as expectations to judge the quality of an RSSE. The evaluation, thereby, measures the recommender's ability to lead the developer towards that final state.

Since artificial evaluation strategies are much cheaper than real evaluations, we expect that toolsmiths and researchers will continue to use them. The goal of our evaluation is to analyze the different heuristics that are used to approximate queries. We do this by comparing an evaluation based on these heuristics to an evaluation based on real queries, which we obtain from our object-usage histories. This evaluation comparison uncovers qualities and problems of the artificial strategies and we identify guidelines for future evaluations.

5.1 Obtaining Real Queries

The usage histories from our dataset mimic the real development history and reflect changes to the files under edit in a very fine-grained manner. In terms of evaluating a recommender, a *query* has an input state and an expected output state. We assume that the last snapshot of a history represents the outcome of a development task. We, therefore, use this final snapshot to formulate our expectation on the evaluated recommender's proposals, similar to how artificial evaluations use the code from a repository. However, the difference between an artificial query and a real query lies in which code state is used for the query input.

We extract 23,746 queries from our usage histories by combining pairs from each snapshot in the history with the final state. After filtering 6,218 pure removals (i.e., queries in which calls were removed, but no calls were added) and 10,371 queries that contained equal start and end states, we ended up with 7,157 real queries for the evaluation.

Start

```
public void M() {
  T t = new T();
  t.m1();
  t.mX();
  t.mY();
}
```

End

```
public void M() {
  T t = T.Create();
  t.m2();
  t.m1();
}
```

Strategy	Definition Site	Calls in Query
Linear	`T.Create`	`m2`
Random	`T.Create`	`m1` and `m2` in multiple queries
Real*	`T.Create`	`m1`
Real	`new T()`	`m1`, `mX`, `mY`

Figure 3: Example of a $3-2+1$ query case (labeled as 1|2) and the queries created for each strategy (Yellow is change, red is removal, and green is addition).

5.2 Generating PBN Queries

At this point, we have the set of real queries obtained from the usage history. To compare the results of a real evaluation to an artificial approach, it is necessary to emulate the heuristics that build an artificial start state from the real end state. Recall from Section 2.2 that there are two dimensions used to automatically generate artificial queries: *query scenario* (0|M, N|M, and M-1|M) and *selection strategy* (Linear, Random, and Real). To create the artificial queries, we first identify the query scenario for each real query and then apply the different selection strategies on that query to create an artificial one.

Query Scenario. We first categorize the collected queries by the type of performed change. We assign each query a label that reflects the number of calls added or removed. A label `n-r+a` means that the query contained `n` calls in the input and that `r` calls were removed, while `a` calls were added, to come to the final state. Consider the query in Figure 3 as an example. The start state contains three calls (`m1`, `mX`, and `mY`). For the final state, `mX` and `mY` were removed, while `m2` was added. Thus, this is an example of a `3-2+1` query.

Since PBN can only suggest method-call additions, and not removals, we needed to adapt query labels accordingly. We do so by dropping the removals from the labels used for categorization. For example, for the query in Figure 3, even though the removals `mX` and `mY` are used in the real query, we do not include them in the final categorization label. Instead, we label the query as a 1|2 change to indicate that the query already contained one out of the two final calls. We assign the queries to the three query scenarios based on this label.

Selection Strategy. Once a query is assigned to a query scenario, we next generate the actual queries for each selection strategy as follows. We use the query in Figure 3 to explain the difference between strategies.

Linear In this case, the query is taken only from the end state. The method calls to be included in the query are selected top-down from the list of existing method calls in the end state. For our example query, which was classified as a 1|2 query according to the above query scenario classification, we could technically generate both 0|2 and 1|2 queries. Yet, we only generate a 1|2 query to have the real query to compare it to. The linear approach would select `m2` for the query, because it is the first method call that exists in the end-state code. However, the order of appearance of calls in source code at the end state might not be the order in which they were added during development, as can be seen from the example.

Random The random strategy also selects the information to include in the query from the end state, but instead of selecting method calls linearly, it selects them randomly. In the end state of Figure 3, there are only two methods. To generate the 1|2 query using the random strategy, we randomly pick one call and use it in the query. To make sure all scenarios are covered, the approach repeats this random selection until all possible method calls are covered and the results are averaged. In our example, two possible queries will be generated where the first includes only `m1` as input and expects `m2` and the second includes only `m2` as input and expects `m1`. An average of the prediction quality of both results is then taken to reflect the prediction quality of this whole query.

Real Based on the unique opportunity of having detailed development information available in our collected usage histories, we introduce the real selection strategy to reproduce what would actually happen during development or during a controlled experiment with subject developers. We only use the information that would be available to a recommender in a real-life scenario where the query was placed during development. This means that only the start state is used to query the recommender. The fine-grained history reflects the *evolving context* over time and includes all information that do not show in the end state, because they were missing, changed, or got removed in the usage history. For example, the query does not only include method `m1`, but also the removed methods `mX` and `mY`, as well as the original definition site `new T()` that was changed during development to the static call `T.Create()`.

Real Without Noise (Real*) To understand the effect of the *evolving context* in real queries, we add a fourth strategy that we call *real**. The only difference between real and real* is that the latter would not include any evolving context in the query. To create the query, real* uses the context of the end state and selects all methods from the start state that have not been removed during development. In the example, the 1|2 query would include only `m1`, because it existed before. In addition, `T.Create()` would be selected as the definition site. We consider *real** to be an artificial approach, because the selection of the methods for the query and the inclusion of the correct definition site can only happen *after the fact.*

The input of each generated query is used to request proposals from the recommender, which is built from the reference models. We measure the prediction quality by comparing the set of proposals with the expected outcome, which is the set of methods that are missing in the query input, but that exist in the end state. The similarity is calculated through the F1 measure (i.e., the combination of recall and precision). A detailed overview of the number of queries we captured in each query scenario is shown in Table 3.

6. DO ARTIFICIAL QUERIES WORK?

In this section, we empirically compare the different selection strategies described in the previous section. We follow the evaluation comparison strategy that has been outlined in Figure 1 and explained in Section 3 in order to answer the two research questions posed in the introduction.

Table 3: Available queries for N|M scenarios

N|M	1	2	3	4	5	6+	Σ	
0	4327	703	592	82	13	30	5747	(80.3%)
1	-	741	109	60	3	6	919	(12.8%)
2	-	-	252	52	15	11	330	(4.6%)
3	-	-	-	95	19	6	120	(1.7%)
4	-	-	-	-	19	5	24	(0.3%)
5+	-	-	-	-	-	17	17	(0.2%)
Σ	4327	1444	953	289	69	75	7157	

6.1 Do Artificial Queries Affect the Measured Prediction Quality of a Recommender?

To compare the evaluation strategies, we feed our reference recommender with the generated queries. The quality is measured by comparing the proposals to the expected additions available in the end state of the query case. Table 4 shows the quality obtained for each selection strategy and corresponding query scenario. We explain these results by going through each query scenario (columns) and comparing the selection strategies (rows). We present the results of each query strategy and provide an interpretation of the findings.

0|M. Queries in this category contain no method calls as part of their input. This means that none of the method calls in the expected state appear in the input state. This is the most common case and 80.3% of our data falls into this category. We factored out NEW as a special subset of 0|M that reflects the case in which the developer did not write any code so far. In this case, no information is available in the code context, apart from the enclosing method and the type of the usage. Another special category is 0|1 queries, which can be assigned to both 0|M and M-1|M. We decided to assign it to the 0|M category but show it as a separate column in the table for better examination of the results.

The results in Table 4 show that for such queries, no difference exists between the artificial strategies. This is not surprising, because all artificial strategies end up with the same query created from the same end state. One observation is that it seems that the more missing calls exist, the harder it is for the recommender to find them. Another observation is a notable difference in the results of real queries. While the difference is already noticeable for NEW queries, it gets worse for 0|1 and the recommender seems to be unable to process 0|2+ queries (4.9%). The only difference here between real and the artificial strategies is missing or changing context information. While the definition site might change for all 0|M queries, 0|1 and 0|2+ might additionally contain calls that are about to be removed.

N|M. Such queries contain some existing calls and reflect the use case in which the developer has already started to write some code and then asks for help. This is the case for 14.5% of the queries in our dataset. We factored out 1|2 queries for the same reasons as the 0|1 queries.

We find that the F1-values for this query scenario differ greatly for 1|2 queries. The random approach reports the highest quality for the recommender (29.9%). The other approaches result in a much lower quality, 15.0% for linear and 19.1% for real*. The real evaluation, which includes *evolving context* in the query, yields the lowest quality with 12.1%). The results for N|3+ differ slightly between the different selection strategies and are around 43-45%. The only approach that sticks out is random, which reports a

Table 4: Different query scenarios [F1/%]

	0|M			N|M		
	NEW	0|1	0|2+	1|2	N|3+	**M-1|M**
LINEAR	60.2	38.2	34.0	15.0	45.4	34.8
RANDOM	60.2	38.2	34.0	29.9	50.0	30.9
REAL*	60.2	38.2	34.0	19.1	45.1	24.0
REAL	53.0	15.1	4.9	12.0	43.3	23.7
# Queries	3612	1445	690	741	294	375
	(50.5%)	(20.2%)	(9.6%)	(10.4%)	(4.1%)	(5.2%)

much higher quality of 50%. Overall, the random results are consistently higher in the N|M query scenario than the other selection strategies. The real evaluation again reports worse quality than the remaining artificial strategies.

M-1|M. This is the extreme case of N|M: only the last call in the method is missing, while the remaining methods are given as part of the input. Only 5.2% of the queries in our dataset fall into this category. The results show that the quality of the real and the real* strategy are comparable. However, the reported quality of random is much higher (30.9%) and even exceeded by the linear strategy (34.8%).

The selection of methods in the query is the only difference between the three artificial approaches, yet we see different results. The linear selection strategy reports the highest quality (34.8%), while real* is close to the real result and reports 24.0%. The random approach mixes the different extremal values and reports a quality in between (30.9%). It seems that some missing methods are harder to predict than others and that developers select these methods last.

Interpretation of Results. For the 0|M queries, no difference can be seen between the artificial approaches. On the other hand, the real queries perform worse, because the definition site is unknown. The more calls that need to be predicted, the more problematic this seems to be. In addition, a direct comparison between real* and real shows that evolving context in real queries reduces the reported quality. This is true for all query scenarios. We look at this more closely in Section 6.2 where we examine the effect of evolving context information.

When compared to the random selection strategy, the linear strategy seems to be better in some cases (e.g., M-1|M), while worse in others (e.g., N|M). The randomized generation of multiple queries and averaging of the results seems to cause a smoothing effect that creates more robust results. We averaged the results over all queries (not shown in the table) to see if this is a general effect. However, we found that all artificial selection strategies achieve results that are comparable to each other (linear 46.6%, random 48.2%, and real* 46.6%).

Although all artificial strategies create queries from the same set of methods found in the end state of a micro commit, they select the methods to include differently. This seems to have a real effect on prediction quality as can be seen for both N|M and M-1|M query scenarios. Such a difference suggests that some methods are harder to predict than others. A possible explanation is that these calls might be used very rarely and the recommender favors common method calls.

To explain this intuition, let us go back to the example in Figure 3 and assume that `m2` is not a very typical method, which makes it harder to predict. In the linear case, the recommender is lucky, because it gets `m2` as input and has to predict `m1`. In the random case, two queries are provided to the recommender. It does really well in the easy query (i.e., when the input is `m2`) and really bad in the other one (i.e., when the input is `m1`). Since the results of both queries are averaged, the extreme effect is smoothed out a little. However, for the real* strategy, the recommender gets `m1` and has to predict the hard method `m2`. It, thus, follows that it would have a lower prediction quality.

As opposed to artificial strategies, queries in the real strategy only include the methods that were actually included by the developer first. Since the prediction quality of the real query is even lower, this also suggests that developers add such hard-to-predict methods later to their code rather than earlier. Thus, it seems that for such corner cases, artificial approaches tend to give a higher prediction quality than it would actually be the case in a real setting.

Table 5: Effects of Evolving Context [F1/%]

	NEW	**0**	**–M**	**ΔD**	**–M+ΔD**
REAL*	60.2	21.4	29.0	38.8	40.9
REAL	53.0	21.4	25.9	24.1	1.9
# Queries	3612	794	1173	249	1329
	(50.5%)	(11.1%)	(16.4%)	(3.5%)	(18.6%)

6.2 Do Real Queries Have Properties Not Reflected in Artificial Queries?

Released code is a static picture of the development activities. Any intermediate changes or removals cannot be identified by artificial evaluations, because they are part of the development process and do not show up in the released code. Our data set provides a unique opportunity to explore the impact of intermediate code changes and code removals on recommender evaluations. To explore this, we compare the *real* and *real** strategies in more detail. The input to real queries might miss context information or might contain context information that is changed in the end state. For PBN, this includes both method calls that have been removed during development and changes to the context of an object usage. The context is defined by the enclosing method and the definition site of an object. By construction, the enclosing method is fixed for all query generation strategies, because object usages are always bound to a specific method. If the enclosing method is changed, this would count as a different object usage. On the other hand, the definition site, which describes how an object instance was created, may change during development (e.g., see the definition site in Figure 3). If there are no removals or changes in the definition site, the query will not contain any artifacts of an *evolving context*, i.e., changed or removed information. Artificial selection strategies are unaware of such changes, and are thus not sidetracked by them. This results in a higher prediction quality than for a real query. We examine the effect of evolving context in more detail in Table 5. In addition to queries in an *unchanged context*, the context may evolve in four different ways that are applicable here. Moving method calls to other enclosing methods is a fifth kind. However, distinguishing these moves from removals is outside the scope of our work and not considered here.

Newly Created Usage (NEW) The most represented category in our dataset is usages that are added from scratch. The values are equal to Table 4 and only included for easier comparison. The difference between real and real* here can be explained by a changed definition site.

Unchanged Context (0) We categorize queries into this category that do no include any changes in the surrounding context. By design, no difference exists for real and real* queries in this category.

Retired Method Calls $(-M)$ This category refers to the case in which the query has calls in the input that are removed during development and no longer contained in the end state. For example, this can happen when the developer chooses a more appropriate method call to use. We find that this is the case for 16.4% of the queries in our dataset. Our results suggest that these extra methods that appear in the input seem to confuse the recommender. The quality decreases from 29% for real* to 25.9% for real.

Changed Definition (ΔD) A small part of our dataset (3.5%) includes queries in which the definition site changes between the input and end state. This particular context change has a big impact on real evaluations. Table 5 shows a quality drop from 38.8% for real* to 24.1% for real.

Combination $(-\mathrm{M} + \Delta D)$ The second largest category in our dataset are queries that combine both a change in definition site as well as the removal of method calls. It prevents any meaningful proposal in our experiment for the real approach and the quality drops to 1.9%.

We find that real evaluations are sensitive to context changes. While retired method calls have a minor impact on the result of a real evaluation, the change or absence of the definition site leads to a large difference in the result. Such an impact is not covered in the artificial evaluation strategies.

While the specific context information we discussed in this section (removed calls and changed definition sites) may be specific to PBN, other recommender systems that use context information will suffer from the same problem when this information evolves (e.g., changing the implemented interfaces [9], removing method calls on related objects [28], re-ordering call sequences [23], etc.). The general problem is that context information can change during development. Artificial evaluations usually do not mimic context evolution, which results in “unrealistic” quality reports.

7. DISCUSSION

Our results indicate that the focus of an evaluation is an important factor when determining which query scenario and selection strategy to use. In the following, we discuss broader implication of our results and implications for future work.

7.1 Implications of Results

Choice of Query Scenario. Depending on the type of users the recommender is meant to support, researchers can decide on which query scenario they use in their evaluation. While 0|M represents green-field projects or novices that request help from the recommender before actually starting to write code, N|M reflects advanced programmers that write something before triggering code completion, or maintenance tasks, in which the programmer typically starts to edit existing methods. The M-1|M query scenario can be used to demonstrate support for corner cases or that even experts that just miss “the last bit” can get valuable support.

Choice of Selection Strategy. Our ground truth reflects developers’ typical code completion usage. In our dataset, 0|M queries are the common case for which no difference exists for the different selection strategies. On the other hand, we find great differences between the selection strategies for the uncommon cases that include the N|M and M-1|M query scenarios. We observed for these query scenarios that some methods seem to be harder to predict than others. Depending on the methods that are contained in the query, the reported quality may vary. The random selection strategy leads to a smoothing effect that can overcome this effect to some extent, but it leads to a reported quality that is generally higher than for the other artificial approaches.

Overall, by comparing the average results over all queries, we could measure only minor differences between the different artificial selection strategies. We could not find a single artificial evaluation that is more realistic than the others. When compared to the result of real queries, we found that all artificial evaluations report a higher quality.

Considering the Effect of an Evolving Context. Missing or changing context information affected many query cases. We found that in real queries, the evolving context has a negative effect on the recommendation quality. The more changed context features the recommender engine takes into consideration, the bigger this negative effect is, because more “confusing” information is passed to the recommender. Researchers should be aware of this effect in their evaluations.

While this does not necessarily invalidate the results of artificial evaluations, it results in more positive results than what would actually be measured from real developers. Existing artificial evaluations remove only the target information (i.e., the method calls). However, to make an artificial evaluation more realistic, toolsmiths have to check their assumptions about the context information used in the query. They have to identify what context information may change in reality and they should mimic that in their automatically generated queries. For example, they should remove additional context information (e.g., definition sites in our case) from the input of a specific fraction of the queries or set it to a random, but valid, value (e.g., set an an arbitrary definition site stored in the model for a fraction of the queries).

7.2 Implications for Future Work

Further Comparisons. We designed our experiments to use a single recommender to compare artificial and real evaluation techniques. Future experiments should compare artificial and real evaluation techniques (using the same dataset) across multiple recommenders. This allows investigating which recommenders are more resilient to evolving context, for example.

Improving Recommenders. We found that 26.2% of the snapshots in our dataset were pure removals of method calls. While it could be the case that these are artifacts of maintenance tasks, we hypothesize that this could also be caused by going through a learning process on how to use a given API to solve a task at hand. These removals are not considered in any RSSE so far, probably because they cannot be observed by statically analyzing code repositories. Future work should investigate these removals. It seems that they contain information that could be leveraged to further improve existing approaches or to create a new kind of RSSE that points the developer to methods that should be removed.

Evaluation Style. Our dataset consists of a series of source code snapshots as illustrated in Figure 4. The evaluation design we used in this paper follows what we refer to as a *goal*

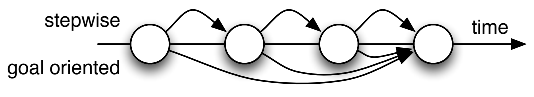

Figure 4: Identifying Expected Query Results

oriented style, in which the validation queries are created from the intermediate states and the proposals are validated on the end state. This follows the intuition that this final state is the desired outcome of a development task and that the recommender should lead the developer towards this outcome. It might be that a recommender that suggests such an end state increases productivity, because it points the developer to methods she must use in the end. However, it might also be that the proposal hinders the learning process, because it might be unexpected at that exact point in time.

We observed in our dataset that the path towards the final state is rarely straight. Therefore, a viable alternative could be to use a *step-wise* evaluation style instead, in which each subsequent change would be evaluated individually. For example, each snapshot could be used as a query and the subsequent snapshot formulates the respective expectations. Such an approach follows the intuition that an RSSE is expected to propose what the developer thought was right at the time of query. Such a tool might support the learning process, which could be beneficial, especially for novice developers. However, experienced developers that know the API in question might be disturbed by incorrect proposals.

It is not clear which of these should recommenders be designed to do and accordingly, which of the evaluation strategies is better or more realistic. To investigate this, we conducted a preliminary experiment that used each usage snapshot as a query and compared the proposals to both the next snapshot and to the final snapshots. We calculated the F1 measure for both scenarios and, over all queries, the stepwise approach resulted in an average of 38.4% and the goal oriented style in an average of 41.4%. A Mann Whitney U test shows that the difference between these two evaluation approaches is statistically significant (p-value=0.001). Therefore, future work should investigate assumptions on the expectations used to evaluate recommender systems. In other words, we need to analyze whether RSSEs should propose the *correct answer* or the *expected answer*.

8. THREATS TO VALIDITY

This section discusses the potential threats to the validity of our work, as well as our mitigation strategies.

Construct validity. The prediction quality range reported by PBN for this dataset is lower than prediction qualities usually obtained in the literature, including our previous experience with PBN on Java SWT [22]. We plan to investigate this more qualitatively to find out if this is a factor of language differences (C# vs. Java) or a factor of the API types included in the ground truth data set. However, our goal is not to promote a particular recommender, but rather to compare different evaluation strategies. Thus, we do not believe that this impacts the validity of our results since any difference between evaluation strategies would still be observed. Additionally, to avoid confounding factors that may affect our comparison, we ensure that we have enough object usages to build reference models for each API type we are interested in. This ensures that all API types have a fair chance of getting good predictions. That said, even if the model for one API type does not have enough data, the effect will be the same across all evaluation strategies, keeping our comparisons fair.

Internal validity. To decrease the possibility of implementation bugs as much possible, we used an existing recommender system with its existing evaluation pipeline. We implemented only those parts necessary for the new experiments and thoroughly tested them. Our code is publicly available on our artifact page. Our results also depend on the quality of the static analysis of the C# code we use to extract the micro commits. C# is a real-world programming language, and the analyzed programs contain very complex expressions. While we excessively tested the analysis in an extensive test suite, it is possible that we may have missed corner cases.

External validity. Our results are based on the comparison of the different evaluation strategies for only one recommendation system for single-object patterns (PBN). Our results might not generalize to other recommenders that have a different notion of context, that deal with multi-object patterns, or that propose complete code snippets. Since we use the same recommender across all evaluation strategies, our results are valid and ensure non-biased comparisons. Since this is a limitation of the recommender rather than our ground truth data set, the same comparison can be repeated with additional recommenders. However, this might entail some engineering effort to adapt each recommender to use the ground truth data set.

We cover various kinds of developers by collecting data from diverse groups. We have data from professional developers, open-source developers, researchers, and students. However, we do not currently have information about the project types these developers worked on. In the future, we will capture more information about the type of the project (e.g., green field or maintenance) and the role of the developer (e.g., developer, tester, or integrator) to get a better picture about the task the developer was working on.

9. CONCLUSION

In this paper, we surveyed related work to identify the current state of the art of evaluation strategies. We found that many existing approaches are based on artificial evaluations. Our goal was to analyze whether these evaluations reflect real-life usages. We presented a concept for the comparison of different evaluation strategies and collected a ground truth data set that allowed us to conduct the comparison. We analyzed how the results of a real evaluation relate to the artificial results. We showed that artificial approaches report a misleading quality for the evaluation if they do not consider evolving context information and, therefore, provide more information in the query than what would be available in a real scenario. We believe that artificial evaluations can still work if context evolution is carefully emulated in the evaluation.

10. ACKNOWLEDGMENTS

The work presented in this paper was partially funded by the German Federal Ministry of Education and Research (BMBF) with grant no. 01IS12054 and by the German Science Foundation (DFG) in the context of the CROSSING Collaborative Research Center (SFB #1119, project E1). The authors assume responsibility for the content.

11. REFERENCES

[1] S. Amann, S. Proksch, and S. Nadi. FeedBaG: An Interaction Tracker for Visual Studio. In *Proceedings of the 24th International Conference on Program Comprehension Tool Track*, 2016.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from Examples to Improve Code Completion Systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, 2009.

[3] R. DeLine, M. Czerwinski, and G. Robertson. Easing Program Comprehension by Sharing Navigation Data. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005.

[4] Evaluating the Evaluations of Code Recommender Systems: A Reality Check – Online Artifact. http://www.st.informatik.tu-darmstadt.de/artifacts/evaleval/.

[5] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete Completion Using Types and Weights. In *Proceedings of the 34th Conference on Programming Language Design and Implementation.* ACM, 2013.

[6] A. E. Hassan and R. C. Holt. Replaying Development History to Assess the Effectiveness of Change Propagation Tools. *Empirical Software Engineering*, 11(3), 2006.

[7] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel. Identifier-based Context-dependent API Method Recommendation. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering.* IEEE, 2012.

[8] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE, 2012.

[9] R. Holmes and G. C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering.* ACM, 2005.

[10] M. Kersten and G. C. Murphy. Mylar: A Degree-of-interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05. ACM, 2005.

[11] M. Kersten and G. C. Murphy. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering*, SIGSOFT'06/ FSE-14. ACM, 2006.

[12] C. Kolassa, D. Riehle, and M. A. Salim. The Empirical Commit Frequency Distribution of Open Source Projects. In *Proceedings of the 9th International Symposium on Open Collaboration.* ACM, 2013.

[13] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 26th Conference on Programming Language Design and Implementation.* ACM, 2005.

[14] R. Minelli, A. Mocci, and M. Lanza. The Plague Doctor: A Promising Cure for the Window Plague. In *Proceedings of the 23rd International Conference on Program Comprehension.* IEEE, 2015.

[15] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus. How Can I Use This Method? In *Proceedings of the 37th International Conference on Software Engineering.* IEEE, 2015.

[16] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014. ACM, 2014.

[17] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. *Proceedings of the 26th European Conference on Object-Oriented Programming*, chapter Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? Springer Berlin Heidelberg, 2012.

[18] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Grapacc: A Graph-based Pattern-oriented, Context-sensitive Code Completion Tool. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE, 2012.

[19] A. T. Nguyen and T. N. Nguyen. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering.* IEEE, 2015.

[20] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[21] S. Proksch, S. Amann, S. Nadi, and M. Mezini. A Dataset of Simplified Syntax Trees for C#. In *Proceedings of the 13th International Conference on Mining Software Repositories, Data Showcase*, 2016.

[22] S. Proksch, J. Lerch, and M. Mezini. Intelligent Code Completion with Bayesian Networks. In *Transactions on Software Engineering and Methodology.* ACM, 2015.

[23] V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th Conference on Programming Language Design and Implementation.* ACM, 2014.

[24] R. Robbes and M. Lanza. How Program History can Improve Code Completion. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008.

[25] R. Robbes and M. Lanza. Improving Code Completion with Program History. *Automated Software Engineering*, 17(2), 2010.

[26] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering*, 2013.

[27] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation Systems in Software Engineering.* Springer, 2014.

[28] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou. Automatic Parameter Recommendation for Practical API Usage. In *Proceedings of the 34th International Conference on Software Engineering.* IEEE, 2012.

[29] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming.* Springer, 2009.