On Evaluating Recommender Systems for API Usages

Marcel Bruch   Thorsten Schäfer   Mira Mezini
Software Technology Group, Darmstadt University of Technology
{bruch,schaefer,mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

To ease framework understanding, tools have been developed that analyze existing framework instantiations to extract API usage patterns and present them to the user. However, detailed quantitative evaluations of such recommender systems are lacking. In this paper we present an automated evaluation process which extracts queries and expected results from existing code bases. This enables the validation of recommendation systems with large test beds in an objective manner by means of precision and recall measures. We demonstrate the applicability of our approach by evaluating an improvement of an existing API recommender tool that takes into account the framework-method context for recommendations.

1. INTRODUCTION

Framework reuse provides several benefits such as reduced costs, higher quality, and shorter time to market. However, a number of issues that complicate the usage of a framework has been identified [3]. Most importantly, a developer needs an understanding of the framework, which often requires a lengthy learning process [12].

A promising approach to the framework understanding problem is to apply automated machine learning techniques such as association rule mining [2] to extract knowledge about the usage of a framework from existing instantiations. Tools like CodeWeb [10] and FrUiT [4] exploit these techniques to mine for API usage rules of the kind "In 80% of the cases where you extend from `Wizard`, you also instantiate a `WizardPage`; this pattern has been observed 91 times.".

First experiments [4, 9, 10, 11, 13, 15] indicate that the recommended API usage rules are meaningful. However, the evaluation techniques underlying these experiments are unsatisfactory. There are two main evaluation approaches. First, qualitative evaluation approaches apply the tool to sample tasks and analyze the results [4, 10, 13, 15], e.g., using domain knowledge or by comparing whether the suggestion made by the tool corresponds to the advice made in the framework documentation. Second, user studies are conducted to judge whether the use of recommender systems increase productivity or decrease the number of bugs [9, 11].

While qualitative evaluations and user studies are useful to judge e.g., the effect of tools on programmers productivity, we also need methods to precisely compare different algorithms. Current approaches are labor intensive and time consuming due to the manual analysis of the results. Furthermore, they are somewhat subjective, since deciding whether a recommendation that "a developer typically follows" is correct is often a matter of subjective judgment.

To address the above problems, we propose an automated quantitative evaluation approach for mining-based tools for recommending API usages. The proposed evaluation process "simulates" the usage of the tool by a human user in an automated manner. It uses existing code that uses an API to automatically extract both sample user queries on API usage patterns and common answers to these queries. These automatically extracted queries and answers to them serve as reference data against which tool results are compared. The proposal makes the following contributions to the state of the art of evaluating recommender systems for framework API usages: It enables to a) validate recommendation systems with large test bases, and b) perform evaluations in an objective manner in terms of precision and recall measures.

We provide evidence for the above claims by using the approach to evaluate two different versions of FrUiT: its original version [4] and a new version that exploits a different technique for preparing the input data and presenting the rules to the user. Existing approaches use the framework class which the user is extending as the context to filter relevant recommendations to be presented. In the new approach presented here the filtering is also based on framework methods. Our evaluation shows that this improves the quality of recommendations in general, and that it is most useful for white-box frameworks.

The remainder of this paper is structured as follows. In the next section we overview tools for frameworks usage assistance. Next, we describe the proposed evaluation process in Sec. 3. In Sec. 4 we illustrate the application of the evaluation process in the evaluation of the proposed improvement of FrUiT. The paper is summarized in Sec. 5.

2. RELATED TOOLS

This section overviews tools that use data mining techniques to assist programmers in using frameworks and the respective evaluation approaches.

Framework users often know what type they need to use, but not how to write code that retrieves an instance of that type. The Prospector tool [9] addresses this issue by modeling the reachability between types as a graph and searching for a path in the latter. XSnippet [11] and PARSEWeb [13] extend this approach by improving ranking heuristics, the query capabilities and the mining process. PARSEWeb also uses code search engines as example repositories which enable to collect code samples on demand.

PR-Miner [7] and DynaMine [8] try to find implicit programming rules to automate the detection of bugs, i.e., violations of the mined programming rules. The basic idea is that programming rules correspond to patterns that exhibit a very high confidence, while bugs are negative examples for these patterns. While PR-Miner [7] analyzes code, DynaMine [8] uses history information from revision control systems.

FrUiT [4] and CodeWeb [10] support understanding the framework instantiation process by recommending usages that should likely be made. They learn API usage patterns from existing instantiation code and recommend those API usages that were made in similar contexts. Other approaches not only find likely program elements to use, but also suggest the order in which methods should be called. For instance, Perracotta infers likely temporal properties of a program using dynamic analysis. Jadet [14] uses existing code bases to mine implicit object interaction protocols. MAPO [15] identifies frequent method invocation sequences using sequence pattern mining.

The mining-based approaches have been evaluated either qualitatively [4, 7, 8, 10, 14, 15], and/or by small-scale user studies [9, 11]. As already argued in the introduction, while such evaluations provide initial evidence that the tools are useful, they cannot deliver an objective quantification of the ability of the evaluated tools to discover relevant knowledge.

The evaluation process presented in this paper is applicable to all mining-based tools discussed above, except for those that aim at bug detection [7, 8]. In the remainder of this paper, we refer to this category of tools as API recommender tools. The approach is not applicable to mining-based bug detection tools since it is based on the automatic extraction of possible user queries and respective answers to be used as reference data against which to compare the output of a tool: knowledge about instantiation code that contains bugs cannot be extracted automatically.

3. EVALUATION PROCESS

We propose an automated evaluation process that measures the effectiveness of a recommender system in terms of *precision* and *recall*. In a nutshell, the evaluation process simulates the usage of the tool by a human user. When using a recommender system a user typically enters a *query* into the system, in response to which the system returns a set of *recommendations*. The user then follows those recommendations that are relevant for the task at hand. After changing the code, the recommendation system may be queried again to see whether new recommendations are made.

To simulate a human user, an automated evaluation process would iteratively execute a set of test queries and then compare the returned usage rules against a set of expected answers. The result of the latter comparison is mapped onto a meaningful measure of the effectiveness of the system. To make the sketched evaluation process operational, we need to answer the following questions:

1. **Queries and expected results**: Which queries to execute and which recommendations to expect?
2. **Performance measures**: How to measure tool performance?
3. **Ranking of recommendations**: How to take the ranking of recommendations into account when calculating performance measures?
4. **Data selection**: Which data to use as the input to the recommender system?

Given answers to these questions, which we elaborate on in the following, the overall process works as follows. First, each query is executed. The results returned by the recommender are compared to the expected recommendations. If an interactive usage of the tool should be simulated, the correct recommendations are added to the context, i.e., we pretend that the user followed them in the order given by the ranking, and the recommender is queried again until no new recommendations are made. Otherwise, the query is only executed once. Next, the effectiveness measure is computed for the query based on its results and the expected recommendations. Finally, the measures are averaged over all queries; this aggregated data expresses the overall performance of the system.

Queries and expected results.

Recommender systems may focus on different understanding questions. Hence, a first step in the evaluation is to define the *types of query* which should be evaluated. Clearly, those depend on the goals on the recommender system and thus need to be designed with respect to the supported understanding questions. For instance, for a recommender system that aims at recommending methods to call on given framework types in given contexts, the queries would specify framework types and the result of the system would be a list of methods to call.

Having defined the type of queries, concrete queries and the results we expect from them need to be defined. As a manual definition of queries and their corresponding relevant recommendations by domain experts is time-consuming and also subjective, we propose to derive queries and their expected results from existing code bases. The rationale behind this decision is that existing instantiation code already contains knowledge on how to use the framework in a particular situation; we consider usages in existing instantiations as correct and thus use them as a test oracle.

Given a system that recommends methods to call on given framework types in given contexts, the queries are extracted by searching for all contexts (i.e., framework subtypes or framework methods, depending on the context model) and all framework types used within these contexts. For each type, a single query is extracted comprised of the type itself and the context. The relevant recommendations are all methods called on the type within the context.

For instance, consider the sample instantiation code snippet in List. 1.

```
1  // code within a method declared by the framework
2  Text textfield = new Text(container, SWT.NONE);
3  textfield.setText("Input data");
4  textfield.setLayoutData(...);
5  textfield.setFont(Font.ARIAL);
```

Listing 1: Existing instantiation code

Based on this code, we can automatically derive the following query: the developer wants to use an instance of a `Text` (cf. line 1) and needs usage recommendations. We can also derive the set of expected recommendations. The developer of this instantiation called the methods `new Text()`, `setText()`, `setLayoutData()`, and `setFont()`; a recommender system should ideally return exactly those four recommendations for the given situation.

Deriving queries and the expected results from existing code bases has several advantages. First, it is an automated process and does not require a large manual effort by domain experts. Second, often large code bases are available, which enables us to create very large test beds. Third, we do not need subjective judgments about the correctness of recommendations; the instantiation code already contains the "correct" recommendations in the respective situation.

Performance measures.

To assess the effectiveness of a recommender system for a given query we use recall and precision measures that express the ability

of the system to a) find all relevant recommendations, and b) filter recommendations not relevant for the given query. Recall is defined as the ratio between the relevant recommendations made by the system for the given query and the total number of recommendations that should have been made. Precision is defined as the ratio between relevant recommendations made and the total number of recommendations made by the system for a particular query.

Let us illustrate the calculation of precision and recall measures for the example code in List. 1. Consider the recommender suggests to make calls shown in Tab. 1. The expected recommendations are four method calls, all of which are actually suggested by the recommender (results 1, 2, 4, and 5); hence, the recall for this query is $4/4 = 100\%$. The recommender returns in total seven recommendations, four of which are relevant; hence, the precision is $4/7 = 57.1\%$.

The use of precision and recall measures enables us to make exact and objective statements about a) the number of false positives a recommender system returns and b) the number of the usages that a developer should actually add to the code at hand recommended by the system.

Ranking of recommendations.

We have shown that, given a set of recommendations and a set of expected results, precision and recall can be calculated. However, typical recommender systems do not only provide a recommendation, but also a measure for their certainty about the relevance of this recommendation.

Such measures are used to rank the recommendations. The most likely relevant recommendations are shown at the top of the list, while the less likely relevant ones are at the bottom. Users start browsing the recommendations from the top of the list and stop after all relevant recommendations were found or too many irrelevant recommendations are presented. Hence, it is crucial that the quality of recommendations at the top is high; the recommendations at the bottom are less relevant, as they are rarely investigated by users .

To incorporate the quality of the recommendation ranking into the assessment of recommender systems, the set-based precision and recall measures cannot be used directly. Instead precision and recall are computed for each position in the ranking individually as follows. For a position k, recall and precision are computed based on all recommendations at position 1 to k; recommendations at positions below k are ignored.

Consider the example presented in Tab. 1. It shows a ranked recommendation set a code recommender could have made for using a text widget. The first column shows the the rank, column two the recommendation made, column three states whether the given recommendation was relevant or not and the remaining two columns show the recall and precision measure for each position. Consider for illustration the entry at position four. Up to this position three of four relevant recommendations are made, which results in a recall of 75% at this position. The precision is also 75% since one irrelevant recommendation is made up to this position.

Typically, we only have a few data points that represent precision and recall pairs. When drawn in a diagram and connecting the data points with a line, a sawtooth-shaped curve as illustrated by line 1 in Fig. 2 is gained. Interpreting such a curve is difficult: if we do not have a data point for a recall level, we cannot determine the respective precision value. The peaks in line 1 may lead to large changes in the precision when the recall level only changes slightly; it is not a good approximation of the real performance of the system.

Therefore, we propose to use interpolated precision [1], a procedure that is widely used in the evaluation of information retrieval systems. Typically the precision for eleven recall levels varying from 0% to 100% is interpolated. For a recall level r, the maximum precision obtained for the query for any recall greater than or equal to the current recall level r is used: $Precision_{interp}(r) = \max_{r' \geq r}\{Precision(r')\}$

The resulting interpolated precision value for each recall level is then shown by line 2 in Fig. 2. The interpolated precision can be used to combine results from multiple queries. Therefore, the interpolated precision recall curves are averaged over all queries; this allows to assess the overall ranking performance of a recommender system. The resulting diagrams are called *11-point average interpolated precision/recall curves*.

One important characteristic of the interpolation process warrants further discussion. In our example, computing standard precision using all recommendations made would give an overall non-interpolated precision of 57%, but computing the worst precision using the interpolated precision approach for ranked results gives an 80% precision. This is because the interpolation process cuts off the recommendations after the last relevant recommendation was found. When discussing the performance of a recommender system, this has to be kept in mind.

Data selection.

An obvious requirement on the data selection is that the data used to train the recommender may not be used to evaluate it. Otherwise, the evaluation would only test the memory capabilities of the recommender system but does not allow to assess whether the system provides useful recommendations in general.

Therefore, the *holdout validation* mechanism [6] is often used, which splits the available data randomly into a *training* and a *validation set*. The training set is used as input to the learning algorithm of the recommender whereas the validation set is used to derive test data for the evaluation. Normally, less than a third of the data is used for validation. Unfortunately, in some cases the number of available instantiation code for a framework is limited. Splitting this code into two sets may lead to small training and validation sets; hence, only few data is available for the recommender to learn from and also only a small number of queries can be derived.

To address this issue, we propose to use *10-fold cross-validation* [6]: the available instantiation code base is split randomly into ten equal-sized partitions; nine partitions are used to train the recommender and the tenth partition is used for validation. The evaluation process is executed ten times, with each partition used exactly once as the validation data. The measures computed for each run are subsequently averaged to produce a single estimation of the performance. The average and the maximum deviations of the measures, i.e., the best and the worst precision and recall values, allow an objective conclusion about the performance of the system and the stability of the underlying model learned.

4. CASE STUDY: USAGE CONTEXTS

In this section, we use the evaluation approach presented in Sec. 3 to compare the performance of recommender systems using three different models of usage context: a) no context at all (i.e., the recommender simply recommends the most frequently used method invocations; it is used for a baseline estimation), b) the framework super-type of the currently edited class (class-based context model), and c) the framework method being overridden within an instantiation class (method-based context). While the first two context models are already used by existing tools [4, 10], the method-based context model is a new contribution. We illustrate this feature using FrUiT as a representative of the tool category mining-based API recommenders.

Pos.	Recommendation	Rel.	Recall	Precision
1	new Text()	x	1/4	1/1
2	Text.setText()	x	2/4	2/2
3	Test.addListener()			2/3
4	Text.setLayoutData()	x	3/4	3/4
5	Text.setFont()	x	4/4	4/5
6	Text.getText()			4/6
7	Text.setBackground()			4/7

Figure 1: Ranked Recommendations for Text

Figure 2: Precision/Recall Curves

4.1 Motivation

In the current version of FrUiT [4] the class in the current Eclipse editor constitutes the usage context used to filter relevant recommendations. This course-grained context model may be too imprecise. For illustration, consider the example code depicted in List. 2.

```java
public class MyPreferencePage extends PreferencePage {
    private Text textfield;
    private Button button;
    private IPreferenceStore store;

    protected Control createContents(Composite parent){
        Composite container = ...
        textfield = new Text(container, SWT.NONE);
        textfield.setText(store.getString(KEY1));
        button = new Button(container, SWT.CHECK);
        button.setSelection(store.getBoolean(KEY2));
        ...
    }

    public boolean performOk(){
        store.setString(KEY1,textfield.getText());
        store.setBoolean(KEY2,button.getSelection());
        ...
    }
}
```

Listing 2: Example Framework Extension

The class `PreferencePage` being extended in line 1 is part of the JFace framework. A preference page is a UI element used to display and to modify the properties of an Eclipse plugin. Instantiations may subclass `PreferencePage` to implement special preference pages by reimplementing its `createContents` and `performOk` methods (cf. lines 6-13, respectively lines 15-19). The responsibility of `createContents` is to create a set of graphical widgets for gathering user input. The `performOk` method is invoked by the framework when the preference dialog is closed; the instantiation class implements it to read out the user input from the widgets and to store it into the preference store.

Note that the two overridden framework methods invoke a distinct set of methods on the text and the button widgets. For instance, in `createContents` a text field is constructed and set up with some text taken from the preference store. In `performOk` the user input is read from the text field using the `getText` method. Generalizing from the particularities of this example, we observe that "different methods within an instantiation class may exhibit different usage patterns of a particular framework type". In such cases, using a more fine-grained method-based context is expected to lead to better performance. By not considering the fine-grained method context, the tool recommends a large number of usage patterns which are not applicable in given method, and thus provides false recommendations.

Solution.

Given a framework type T, our method-based approach to determine context is as follows.

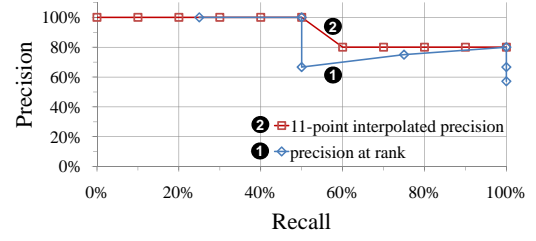**Step 1** computes the overall probability distribution p_o for call-ing methods on T. For each method n in T, we compute the relative probability by dividing the number of methods calling $T.n$ by the number of methods that have a reference to T. The whole instantiation code is taken into consideration when calculating p_o.

Step 2 computes for each method m defined by the framework and redefined in instantiation code its specific probability distribution p_m for calling methods on T; here, only usages that have been observed in any implementation of m are considered.

Step 3 compares p_m with p_o. The null hypothesis is that p_m is similar to the overall distribution p_o, i.e., the likelihood of calling a method within m is similar to calling it within any other method. We test this hypothesis with the Kolmogorov-Smirnov test [5]. If this test fails, i.e., the method-specific distribution p_m significantly differs from the overall distribution, we use p_m to predict likely methods to call for that particular method. Otherwise, the overall model p_o is used.

4.2 Evaluation

Setup.

Two UI frameworks, SWT and JFace, were used as test data. This selection enables us to investigate whether the type of framework, white-box (JFace) versus black-box (SWT), affects the performance of the recommender systems being evaluated. The typical usage scenario we examined here is as follows: *Given an instance of framework type T within framework method m. Which methods should be invoked on the instance of T?* Given that typical usage scenario, the queries are derived from the validation set as follows. First, all methods in the validation data that are declared by the framework are selected. For each of those methods m and for any framework type T referenced within m a query is formulated consisting of T and the context model supported by the recommender system being evaluated (none, class-based, or method-based). The expected results correspond to the actual usages of T within method m. Finally, the recommender system is queried once and precision and recall are computed. Tab. 1 lists number of queries that could be derived by using the method-based versus the class-based context model. The numbers show that given a sublass of a framework type, the method-based context model could be used in 1/6 of the cases for SWT, respectively 1/3 of the cases for JFace.

Experiment	Number of queries	
	Class-based	Method-based
JFace	37414	11572
SWT	6261	1022

Table 1: Size Characteristics

Results.

Fig. 3 & 4 summarize the performance of the three recommenders. The lines show the average precision recall curve, while the data
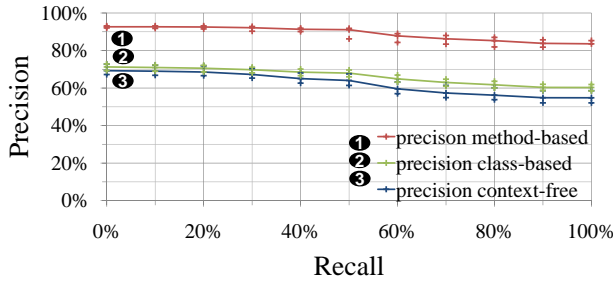
Figure 3: Precision/recall curve for JFace

Figure 4: Precision/recall curve for SWT

points above and below show the maximum/minimum value gained in the 10 runs. Line 3 depicts the precision of the context-free recommender. The line shows that this recommender performs well, even though it does not use any context information. For JFace approximately 2/3 of the recommendations presented in the upper half of the ranking (left side of the 50% recall level) represent actual relevant recommendations and decreases slightly for the lower positions in the ranking. For the SWT framework the performance is a little lower. Here an average precision of 40% is reached. From a user point of view, this means that out of 10 recommendations, 4 recommendations are actually relevant.

Lines 1 and 2 show the performance of the recommenders with the method-based, respectively the class-based context model. For both frameworks, the recommender with the method-based context model performs best with a precision between 80 and 92%. The class-based recommender (line 2) performs better compared to the recommender that uses no context information, but worse than the method-based recommender.

An interesting observation is that for SWT, the class-based context model improves the precision significantly as compared to the context-free model; the method-based context model does not improve the prediction quality much further. For JFace, however, class-based context is not sufficient; here, the recommender with the method-based context model outperforms the other two systems significantly. The reason for this differences in performance is that the subject frameworks are of different types. SWT is a black-box framework and provides only few classes that can be extended and those are typically simple listener or adapter classes with a single method to be implemented. Hence, the class-based context provides sufficient information to predict the methods that should be invoked on a framework type. In contrast, JFace is a white-box framework and most of its classes provide several hook methods to be overridden by instantiations. Each of these hook methods has its own specific responsibilities and hence exhibits its own specific usage patterns. Consequently, recommenders with a method-based context model are much more effective for white-box frameworks.

Summarizing, using the method-based context to predict method calls affects the order of the proposed recommendations: those calls that are most likely relevant within method m are at the top of the ranking. This results in less false recommendations and thus improves the prediction quality. The evaluation has shown that the approach makes sense, as it improves the predictive quality for both frameworks. In case of white-box frameworks, this approach is particularly beneficial as it outperforms existing recommenders by far.

5. SUMMARY

In this paper, we presented an approach to evaluate recommender systems for framework API usages in a quantitative manner. By automa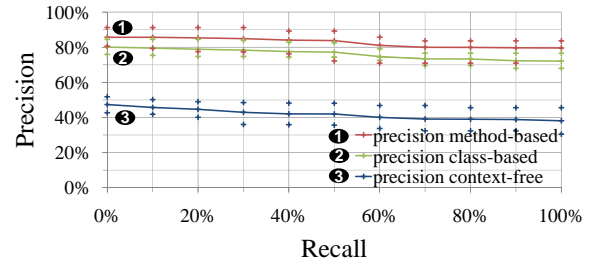ting the process, evaluations with a large test base can be performed and the recommender system's quality is measured in terms of precision and recall. Furthermore, we demonstrated the usage of this evaluation process in a case study.

6. REFERENCES

[1] *Overview of the Third Text REtrieval Conference (TREC-3)*, Gaithersburg, MD, USA, 1990. NIST.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Int'l Conf. Management of Data*, pages 207–216. ACM, 1993.

[3] J. Bosch, P. Molin, M. Mattsson, and P. Bengtsson. Object-oriented framework-based software development: Problems and experiences. *ACM Computing Surveys*, 32(1):3–8, 2000.

[4] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE support for framework understanding. In *OOPSLA Workshop Eclipse Technology Exchange*, pages 55–59. ACM, 2006.

[5] W. T. Eadie, D. Drijard, and F. E. James. *Statistical methods in experimental physics*. Amsterdam: North-Holland, 1971.

[6] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1137–1145, 1995.

[7] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, pages 306–315. ACM, 2005.

[8] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *FSE*, pages 296–305. ACM, 2005.

[9] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61. ACM, 2005.

[10] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE*, pages 167–176. ACM, 2000.

[11] N. Sahavechaphan and K. Claypool. Xsnippet: Mining for sample code. In *OOPSLA*, pages 413–430. ACM, 2006.

[12] F. Shull, F. Lanubile, and V. R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE TSE*, 26(11):1101–1118, 2000.

[13] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *ASE*, pages 204–213. ACM, 2007.

[14] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *FSE*, pages 35–44. ACM, 2007.

[15] T. Xie and J. Pei. MAPO: mining api usages from open source repositories. In *MSR*, pages 54–57. ACM, 2006.