Empir Software Eng
DOI 10.1007/s10664-015-9397-1

Prompter

Turning the IDE into a self-confident programming assistant

Luca Ponzanelli[1] · Gabriele Bavota[2] · Massimiliano Di Penta[3] · Rocco Oliveto[4] · Michele Lanza[1]

Abstract Developers often require knowledge beyond the one they possess, which boils down to asking co-workers for help or consulting additional sources of information, such as Application Programming Interfaces (API) documentation, forums, and Q&A websites. However, it requires time and energy to formulate one's problem, peruse and process the results. We propose a novel approach that, given a context in the Integrated Development Environment (IDE), automatically retrieves pertinent discussions from Stack Overflow, evaluates their relevance using a multi-faceted ranking model, and, if a given confidence threshold is surpassed, notifies the developer. We have implemented our approach in PROMPTER, an Eclipse plug-in. PROMPTER was evaluated in two empirical studies. The first study was aimed at evaluatingPROMPTER's ranking model and involved 33 participants.

Communicated by: Sung Kim and Martin Pinzger

✉ Luca Ponzanelli
luca.ponzanelli@usi.ch

Gabriele Bavota
gabriele.bavota@unibz.it

Massimiliano Di Penta
dipenta@unisannio.it

Rocco Oliveto
rocco.oliveto@unimol.it

Michele Lanza
michele.lanza@usi.ch

[1] REVEAL @ Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland

[2] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy

[3] Department of Engineering, University of Sannio, Benevento, Italy

[4] CSSC Lab - Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy

Published online: 30 August 2015

The second study was conducted with 12 participants and aimed at evaluating PROMPTER's usefulness when supporting developers during development and maintenance tasks. Since PROMPTER uses "volatile information" crawled from the web, we also replicated *Study I* after one year to assess the impact of such a "volatility" on recommenders like PROMPTER. Our results indicate that (i) PROMPTER recommendations were positively evaluated in 74 % of the cases on average, (ii) PROMPTER significantly helps developers to improve the correctness of their tasks by 24 % on average, but also (iii) 78 % of the provided recommendations are "volatile" and can change at one year of distance. While PROMPTER revealed to be effective, our studies also point out issues when building recommenders based on information available on online forums.

Keywords Recommenders · Mining software repositories · Stack overflow · Empirical studies

1 Introduction

The myth of the lonely programmer is still lingering, in stark contrast with reality: *Software development, also due to the ever increasing complexity of modern systems, is tackled by collaborating teams of people*. A helping hand is often required, either by team mates (Ko et al. 2007), through pair programming sessions (Constantine 1995), or the perusal of vast amounts of knowledge available on the Internet (Umarji et al. 2008). While asking team mates is the preferred means to obtain help (LaToza et al. 2006), their availability may fall short. In this case, developers resort to electronically available information.

This comes with a number of problems, the main one being the absence of automation. Every time developers need to look for information, they interrupt their work flow, leave the IDE, and use a Web browser to perform and refine searches, and assess the results. Finally, they transfer the obtained knowledge to the problem context in the IDE. The information is retrieved from different sources, such as forums, mailing lists (Bacchelli et al. 2012), blogs, Q&A websites, bug trackers (Anvik et al. 2006), *etc.* A prominent example is Stack Overflow, popular among developers as a venue for sharing programming knowledge. Stack Overflow is vast. In 2010 it already had 300k users, and millions of questions, answers, and comments (Mamykina et al.). This makes finding the right piece of information cumbersome and challenging.

Recommender systems (Robillard et al. 2010) represent a possible solution to this problem. A recommender system gathers and analyzes data, identifies useful artifacts, and suggests them to the developer. Seminal tools, such as ERose (Zimmermann et al. 2004), HIPIKAT (Cubranic and Murphy 2003) and DEEPINTELLISENSE (Holmes and Begel 2008), suggest project artifacts in the IDE aiming at providing developers with additional information on specific parts of the system. They come however with a caveat: the developer must proactively invoke them, and, once invoked, they continuously display information, which may defeat their purpose, as they augment the complexity of what is displayed in the IDE. *Ideally, a recommender system should behave like a prompter in a theatre: ready to provide suggestions whenever the actor needs them, and ready to autonomously give suggestions if it feels something is going wrong*.

The interaction between the theatre prompter and the actor is similar to the interaction between two developers doing pair programming, working side by side to write code. These developers have different roles, *i.e.,* the driver, who is in charge of writing code, and

the observer, who observes the work of the driver (Williams 2001), tries to understand the context, and, if she has enough confidence, interrupts the driver by giving suggestions. In addition, the driver can consult the observer whenever she needs it, making the observer the programming prompter of the programming actor.

By following a metaphor similar to the interaction between the theatre prompter and the actor, we propose PROMPTER, a fully automated tool that retrieves and recommends, through push notifications, relevant Stack Overflow discussions to the developer. PROMPTER makes the IDE a programming prompter that silently observes and analyzes the code context in the

IDE, searches for Stack Overflow discussions on the Web, evaluates their relevance by taking into consideration *code aspects* (*e.g.,* code clones, type matching), *conceptual aspects* (*e.g.,* textual similarity), and Stack Overflow *community aspects* (*e.g.,* user reputation) to decide, given a certain amount of self-confidence (encoded in a threshold the user can change through a slider, to make the recommender quiet or talkative) when to suggest discussions.

We have evaluated PROMPTER through two studies. In the first one (*Study I*) we asked 33 developers to indicate to what extent the Stack Overflow discussions pushed by PROMPTER for 37 code snippets were actually related to them. Participants positively evaluated 76 % of the recommendations. In the second study (*Study II*) we assessed the usefulness of PROMPTER to developers performing development and maintenance tasks. We evaluated the performance (in terms of completeness of the assigned task) of 12 participants when performing the assigned tasks with and without PROMPTER support. The achieved results showed that the boost provided by the PROMPTER recommendations helped participants in improving their performance of 24 % on average.

In addition, we replicated *Study I* after one year with 18 additional participants. The goal of the replicated study was to assess the impact of the "volatility" of the information exploited by PROMPTER on its recommendations. Indeed, PROMPTER exploits information mined from Stack Overflow that are not guaranteed to be always available (*e.g.,* a discussion might be deleted or moved toward other Q&A websites) and that can change over time (*e.g.,* the content of a question in a Stack Overflow discussion can always be updated by the user who formulated the question). The results of our study showed that 78 % of the Stack Overflow discussions recommended by prompter in *Study I* changed after one year (*i.e.,* a different Stack Overflow discussion is pushed by PROMPTER starting from the same code context). This result highlights the irreproducibility of studies performed to evaluate recommenders mining information from "volatile" sources like Q&A websites. Also, it highlights the fact that the performances of recommenders based on information available on online Web forums or code search engines can vary over time. Despite these changes in the PROMPTER recommendations, our results show that the new Stack Overflow discussions pushed by PROMPTER are as well-evaluated by participants as the old ones.

Summarizing, this paper makes the following contributions:

- a novel ranking model to evaluate the relevance of a Stack Overflow discussion, given a code context in the IDE, by considering code, conceptual and community aspects;
- the implementation of our approach in the PROMPTER Eclipse plug-in;
- an empirical evaluation, *Study I*, aimed at validating the devised ranking model;
- a controlled experiment, *Study II*, aimed at evaluating the usefulness of PROMPTER during development and maintenance tasks;

– the replication of *Study I* after one year, to assess the impact of information volatility on the PROMPTER recommendations;
– a comprehensive replication package publicly available at http://prompter.inf.usi.ch/.

Structure of the Paper In Section 2 we present our approach and its implementation in PROMPTER. In Section 3 we present the results of the ranking model evaluation of PROMPTER (*Study I*). In Section 4 we present the evaluation of PROMPTER with developers (*Study II*), while the replication of *Study I* one year later is described in Section 5. We discuss threats to validity in Section 6 and review related work in Section 7. Section 8 concludes the paper and outlines directions for future work.

2 PROMPTER

We first introduce PROMPTER's user interface and architecture. We then discuss and describe the approach implemented in PROMPTER, especially focusing on the ranking model, its features, and the techniques that enable its self-confidence.

2.1 User Interface

Figure 1 shows the user interface of PROMPTER. It provides two views through which the user can (i) receive and track notifications, and (ii) read the suggested Stack Overflow discussions. The notification center (1) is the main view of PROMPTER and it is used to notify the developer whenever a relevant result is available. When PROMPTER considers a discussion as relevant for the current context (*i.e.,* for the code opened in the IDE), it opens the notification center and plays a sound. If a Stack Overflow discussion is notified more than once, it is pushed to the top of the list for visibility.

Figure 2 shows an example of notification. The developer is provided with some information regarding

(a) the title of the Stack Overflow discussion,
(b) the notification date and time, and

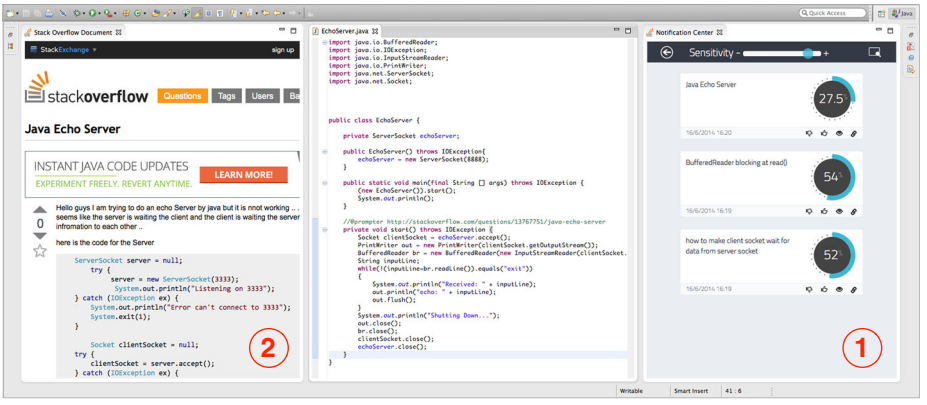


Fig. 1 The PROMPTER User Interface

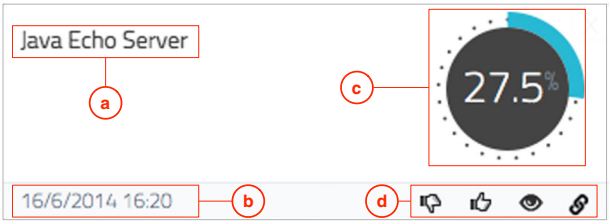


Fig. 2 PROMPTER notification details

(c) the confidence level of PROMPTER on the Stack Overflow discussion against the related code context,

Moreover, PROMPTER provides feedback, tracking and linking functionalities in the bottom-right corner. By clicking on the thumb up (down) icon, the developer can rate the discussion as useful (useless) with respect to the coding activity she is performing in the IDE. Currently, the feedbacks provided by PROMPTER's users are only stored in a database for possible future usages, including: (i) a better tuning of the PROMPTER ranking model, and (ii) the possibility to gather indications on the goodnesses of the PROMPTER's recommendations during case studies.

The other icons on the notification allow the developer of backtracking the code entity associated with a specific notification (eye icon), or to link the suggested discussion to its code entity (chain icon). If the developer clicks on the former, PROMPTER opens up a code editor and highlights the portion of code related to the notification. If the developer clicks on the latter, a simple annotation reporting the URL of the discussions is created in the code in form of a comment.

Whenever a developer clicks on a notification, a Stack Overflow document view (point 2 in Fig. 1) is opened, which shows the contents of the Stack Overflow discussion. At the top of the notification center, the developer can change the sensitivity of the notification system (point 2 in Fig. 3(a)): by sliding to the right PROMPTER is more talkative and produces more notifications, by sliding to the left it becomes more taciturn and requires a higher level of confidence to notify the developer. Moreover, by clicking on the arrow in the top-left corner (point 1 in Fig. 3(a)), the developer can access the full result set of Stack Overflow discussions related to the last notification (*i.e.,* the other Stack Overflow discussions retrieved by PROMPTER for the same code context but not pushed as having a lower confidence level).

(a) PROMPTER sensitivity bar

(b) PROMPTER manual search bar

Fig. 3 Notification center bars of PROMPTER

Explicit Query Writing Sometimes PROMPTER is not able to point out the right Stack Overflow discussion or probably it has not enough information to generate a notification. A similar situation could happen at the very beginning of the development, when there are just few lines of code (*e.g.,* a class stub) written in the IDE's code editor. To overcome these situations, we implemented an additional manual interaction where we provide the developer with the capability to perform manual searches. Whenever the developer wants to search for Stack Overflow discussions on her own, she can click on the manual search button at the top right corner (point 3 in Fig. 3(a)). The notification center disappears and a manual search bar becomes available (Fig. 3(b)). There, the developer can manually type a query (point 4 in Fig. 3(b)) and search for Stack Overflow discussions. As it will be clearer later, the first version of PROMPTER did not implement the search bar for manual query formulation. The need for such a feature has been highlighted by participants of our second study (see Section 4).

The results are presented in form of notification, where each one presents a confidence value according to the code context obtained from the code editor on top, that is, the active code editor. While the developer is interacting with the manual search view, she can continue modifying and writing code. If PROMPTER pushes a discussion in the meanwhile, the developer is notified anyway: a counter of the unseen notifications will popup on top of the notification center icon—point 5 in Fig. 3(b)—and it resets as soon as the developer accesses the notification center by clicking on the icon.

Explicit Invocation A prompter in a theatre not only prompts the right sentence to the actors on the stage, but also provides support on demand. Indeed, an actor can always ask the prompter for a cue in order to go on with the show. In PROMPTER we implemented the same interaction: the developer can always ask PROMPTER to perform a search on a specific code entity (*i.e.*, method or class), by accessing the contextual menu in the code editor, or on

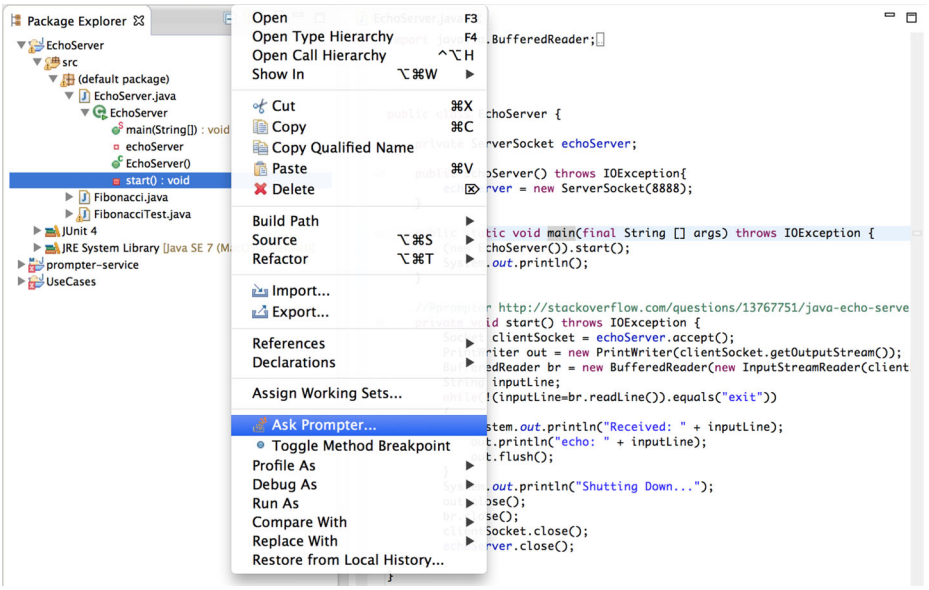


Fig. 4 Explicit invocation of PROMPTER via contextual menu

the package explorer. In the first case, PROMPTER searches discussions for the code entity identified by cursor in the editor, while in the second case it searches according to the code entity selected (see Fig. 4).

2.2 Architecture and Control Flow

Figure 5 depicts the interactions among all the components of PROMPTER when it searches, evaluates, and triggers a new notification to the developer.

PROMPTER tracks code contexts every time a change in the source code occurs. The extracted code context—code elements to formulate the query—is sent to the *Query Generation Service*, which formulates a query starting from the code context. It extracts a query and, according to a set of parameters described later, determines if a new search can be triggered. This information is sent back, with the query and the context, to the plug-in. Since the query is the basis of every search triggered by PROMPTER, the plug-in also considers the query when deciding to trigger a new search. PROMPTER submits a new search only if the query differs from the last one. The query and code context are sent to the *Search Service*, which acts as a proxy between the plug-in, the search engines to which the query is sent, and the *Stack Overflow API*. The query is sent to search engines (Google, Bing) to perform a Web search on the Stack Overflow website. The first 100 Stack Overflow discussions retrieved by each of the two search engines (a retrieved URL refers to a question from Stack Overflow if it matches the form *stackoverflow.com/questions/⟨id⟩/⟨title⟩*) are collected and merged in a single set, where duplicates are removed. Note that this set of retrieved Stack Overflow's discussions is not ranked in any way (*i.e.,* we ignore the ranking made by the search engine) since PROMPTER will evaluate the relevance of each of these discussions to

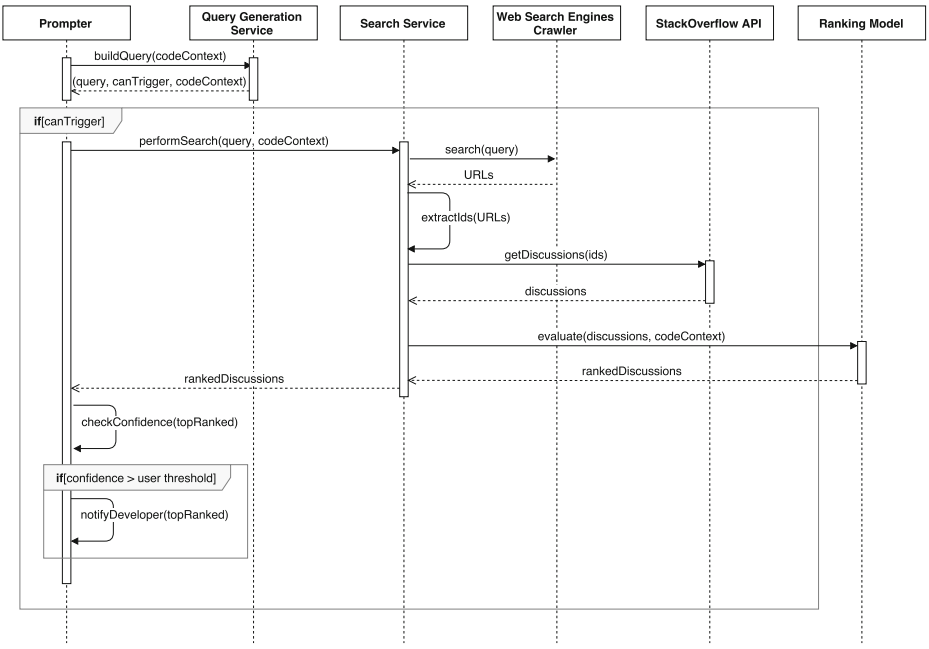


Fig. 5 The UML sequence diagram representing the background search phase performed by PROMPTER whenever the developer modifies a code entity

the code context by using its own ranking model. The search service uses the Stack Overflow question ID to retrieve the discussion via the Stack Overflow API. Every discussion is ranked according to the *Ranking Model* (see Section 2.3), that takes into account the developer's code context. The ranked list of URLs, along with the related confidence values given by the model, is sent back to PROMPTER. The plug-in takes the top-ranked discussion and evaluates its confidence level against the threshold set by the developer. In case the confidence surpasses the threshold, the top-ranked discussion is notified to the developer.

2.3 Retrieval Approach

Our approach is capable of (i) connecting different aspects of the code written by developers to the information contained either in the text or in the code of Stack Overflow discussions, and (ii) taking into consideration information about the quality of the discussions that Stack Overflow has available (*e.g.,* user reputation and questions/answers score). Our previous work only used text similarity to retrieve Stack Overflow discussions related to the actual code (Ponzanelli et al. 2013a). This led to errors in the identification of relevant discussions.

Tracking Code Contexts in the IDE PROMPTER is meant to be a silent observer "looking" at what a developer writes, with the aim of suggesting relevant Stack Overflow discussions. Whenever the developer types, PROMPTER waits until the developer stops writing for at least s seconds,[1] identifies the current code element (*i.e.*, method or class) that has been modified, and extracts the current context, which consists of: (i) a fully qualified name identifying the code element;[2] (ii) the source code of the modified element (*i.e.*, class or method); (iii) the types of the used API, taking into account only types outside the analyzed Eclipse project (*i.e.*, declared in external libraries or in the JDK); and (iv) the names of methods invoked in the API, again considering only external libraries and JDK only. The extracted information (*i.e.*, the context) is sent to the *Query Generation Service* (see Fig. 5) to generate a query.

Generating Queries from Code Context Since we want to automatize the triggering of searches for discussions on Stack Overflow, we have to devise a strategy to build a query describing the current code context in the IDE. A näive approach (Ponzanelli et al. 2013a) is to treat the code as a bag of words by: (i) splitting identifiers and removing stop words; (ii) ranking the obtained terms according to their frequency; and (iii) selecting the top-n most frequent terms. Using only the frequency value is not highly discriminating in selecting terms that appropriately describe the context: Words like *run* or *exception*, even if very frequent in source code, have a too general meaning in programming to discriminate the programming context. Our solution is to also consider the entropy of a given term t in Stack Overflow—previously used in the context of quality assessment and

[1] The s threshold is customizable. By default it is set to 5.

[2] Classes are identified by the unique id projectName.packageName.ClassName, methods are identified by projectName.packageName.ClassName.methodSignature

reformulation of queries for text retrieval in software engineering (Haiduc et al. 2013; Haiduc et al. 2012; Haiduc et al. 2012)—and computed as:

$$E_t = -\sum_{d \in D_t} p(d) \cdot \log_{\mu} p(d) \tag{1}$$

where D_t is the set of discussions in Stack Overflow containing the term t, μ is the number of discussions in Stack Overflow, and $p(d)$ represents the probability that the random variable (term) t is in the state (discussion) d. Such a probability is computed as the ratio between the number of occurrences of the term t in the discussion d over the total number of occurrences of the term t in all the discussions in Stack Overflow. The entropy has a value in the interval of [0, 1]. The higher the value, the lower the discriminating power of the term. We computed the entropy of all 105,439 terms present in Stack Overflow discussions by using the data dump of June 2013[3]. Frequent terms exhibit high levels of entropy (*e.g.,* for *run* the entropy was 0.75) compared to less frequent and more discriminative terms (*e.g.,* for *swt* the entropy was 0.25). Therefore, term entropy can be used to lower the prominence of frequent terms that do not sufficiently discriminate the context.

It is important to point out that the interpretation of term entropy is more similar to the interpretation of Gibbs' entropy from thermodynamics than to the Shannon's entropy (Shannon 1948). That is, words that are highly diffused across documents have high entropy, much alike particles in a gas, whereas words occurring only in few documents have a low entropy, much alike particles in a solid. Our definition of entropy may still be interpreted as a Shannon entropy, similarly to what done by Hassan (2009) to change entropy. That is, if a word is scattered in many files, you need more bits to keep track of where it is located (*e.g.,* the inverted index representation would allocate more memory for that word) than if it appears in few documents.

Last, but not least, it is important to point out that, while E_t converges to a *idf* (*i.e.,* Inverse Document Frequency) when a term is diffused, strictly speaking the definition of E_t is different from the *idf* definition. Indeed, previous studies that compared the *idf* and the entropy, concluded that "*despite the - log(P) form of the traditional IDF measure, any strong relationship between it and the ideas of Shannon's information theory is elusive.*" Robertson (2004).

The *Query Generation Service* ranks the terms in the context based on a term quality index (**TQI**):

$$\mathbf{TQI}_t = \nu_t \cdot (1 - E_t) \tag{2}$$

where t is the term, ν_t is frequency in the context, and E_t is its entropy value measured as described before.

Once the ranking is complete, the *Query Generation Service* selects the top n terms to devise the query, plus the word *java*. The query can exceed n terms in case two or more terms exhibit the same **TQI** value.

[3] http://www.clearbits.net/torrents/2141-jun-2013

To better understand this process, we show an example of query creation.

```
@Override
public List<String> filter(final List<String> tokens) {
    final List<String> stemmed = new ArrayList<String>();
    for(final String t : tokens){
        SnowballStemmer stemmer = new englishStemmer();
        stemmer.setCurrent(t);
        stemmer.stem();
        stemmed.add(stemmer.getCurrent());
    }
    return stemmed;
}
```

Listing 1 Example code entity from which the Query Service extracts a query

Listing 1 shows a Java method from which the *Query Service* has to extract a query. The method is making use of a library applying the *Snowball Stemmer*[4] on a set of tokens. By treating the code as bag of words, we tokenize the text on white spaces, split on case-change, symbols and numbers, lower the case, and remove English stop-words and Java keywords. Table 1 shows the resulting tokens with the respective frequency and entropy value. Tokens in bold are the one selected for the query.

We can notice how the entropy acts as dumping factor for the frequency, making high entropy value terms lose power. An example is the term *tokens* that has more priority than the term *list* even though it has half the frequency values of the other term. However, the term entropy approach has one drawback. We observed that terms with a very low entropy (thus good candidates to be part of a query) may be terms containing typos (*e.g.,* for *override* the entropy was 0.63, and for *overide* it was 0.05). They are present in very few Stack Overflow discussions and thus have a low entropy. To overcome this problem, before selecting the n terms to create the query, we use the *Levenshtein distance* (Levenshtein 1966) to check for terms with a very high textual similarity. If we detect two terms (say t_i and t_j) having *Levenshtein distance* = 1, the term having the lower frequency in the context (say t_i) is discarded and considered as a likely typo, and its frequency is added to the frequency of t_j. If the two terms have the same frequency, we discard the lower entropy term as a likely typo.

2.4 Prompter Ranking Model

The goal of the ranking model is to rank the retrieved Stack Overflow discussions, and assign them a value that measures their relevance to the query. It relies on 8 different features that capture relations between Stack Overflow discussions and source code.

1. **Textual Similarity:** The similarity of the code in the IDE to the textual part of a Stack Overflow discussion without code samples. The goal is to assess the similarity between the topics of the code and the topics of the discussion. We use APACHE LUCENE to create the index and preprocess the contents, by removing English stop words and Java language keywords, by splitting compound identifiers/token based on case change and presence of digits, and by applying the Snowball stemming. Finally, we compute the cosine similarity among the *tf-idf* vectors(Baeza-Yates and Ribeiro-Neto 1999; Manning et al. 2008).

[4] http://snowball.tartarus.org/

Table 1 Selected terms for the code entity in Listing 1

Term	Frequency	Entropy	TQI
stemmer	6	0.15	5.10
stemmed	3	0.15	2.55
tokens	2	0.45	1.10
list	4	0.74	1.04
snowball	1	0.11	0.89
stem	1	0.25	0.75
english	1	0.51	0.49
filter	1	0.58	0.42
array	1	0.72	0.28
set	1	0.80	0.20
add	1	0.84	0.16

2. **Code Similarity:** The percentage of lines of code in the IDE that are cloned in the Stack Overflow discussion. We use DUDE (Wettel and Marinescu 2005), a fast and lightweight line-based textual clone detector, to identify cloned statements among code and documents.
3. **API Types Similarity:** The percentage of API types used in the code that are also present in the Stack Overflow discussion. These are types that are not declared in the project, but in external libraries or in the JDK. The higher the usage of the same types in both discussions and code, the more the potential usefulness of the discussions. To identify the API types, we parse every code sample in the discussion with the Eclipse JDT parser. We are able to resolve types among different samples in the discussion as long as the fully qualified name (*e.g.,* imports) of the type is used in one them, or if the identified type is part of the standard JDK. In case of unresolved types, we match the identified simple name of the class with the simple name of the types used in the code.
4. **API Methods Similarity:** The percentage of API method invocations in the code present in the Stack Overflow discussion. Higher values suggest a similarity in API usage. We use the Eclipse JDT parser to identify method invocations that respect the Java grammar even if the type is not resolved. Since we can only identify the name and number of parameters without any signature, we only consider the name of the invoked method, which helps matching overloaded methods.
5. **Question Score:** The quality of the score of the question in the Stack Overflow discussion. Since the score is not bounded, we normalize the value in the range [0,1] using a sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{(\bar{x} - x)}} \tag{3}$$

where x is the score and $\bar{x}$ is the average of the scores of all the questions in Stack Overflow according to the data dump of June 2013. This index indicates the quality of the question, according to the Stack Overflow community.
6. **Accepted Answer Score:** The quality of the score of the accepted answers in the Stack Overflow discussion. In case no accepted answer is present, the score is set to zero. The score is normalized like the question score, using the related average score. This

index indicates the quality of the accepted answer, according to the Stack Overflow community.

7. **User Reputation:** The level of reputation of the person who posted the question. The value is normalized like the two previous features, using the related average value. Differently from the two previous indexes, this index evaluates the reliability of the person who asked the question on the Stack Overflow community.
8. **Tags Similarity:** The percentage of tags covered by keywords extracted from imports. Tags gets split on number and symbols to remove versions, and tokens are matched against the tokens obtained by splitting imports on dots and lowering the case, without considering the case change. For example the tag *apache-httpclient-4.x* becomes *[apache, httpclient]* and the import statement *org.apache.http.client.HttpClient* becomes *[org, **apache**, http, client, **httpclient**]* . In this case, there is a 100% coverage of the tags. The idea is to identify the topics or libraries used in the discussions even if there is no code in the discussion.

Ranking Model Definition These 8 features are linearly combined to define the ranking model. Each feature is assigned a weight that defines the impact of this specific feature on the overall score:

$$S = \sum_{i=1}^{n} w_i \cdot f_i \qquad \text{having} \qquad \sum_{i=1}^{n} w_i = 1 \tag{4}$$

where $f_i \in [0, 1]$ is a feature value and $w_i \in [0, 1]$ is the assigned weight. In doing so, the score S ranges in the interval [0,1] as well. The next step is to calibrate the weights of the PROMPTER features in (4).

Calibration of the ranking model We need a way to objectively measure the recommendation accuracy of a given PROMPTER configuration, a "gold standard" composed of code contexts each of which is linked to a set of "relevant" (useful to a developer working on a specific context) Stack Overflow discussions. With such a dataset, the recommendation accuracy of a specific PROMPTER configuration can be measured as the number of code contexts for which PROMPTER is able to retrieve a relevant Stack Overflow discussion in the first position. Since PROMPTER recommends only the top ranked document, we only need to evaluate the accuracy for that document.

To identify the best configuration we used an exhaustive combinatorial search. We measured the performance of all configurations obtained varying each weight between 0 and 1 with step size 0.01 where the weights total 1, as defined in (4). Although time-consuming, this avoids that a possible sub-optimal calibration affects the study results. If a faster calibration were required, a search-based approach could be used, as done for information retrieval (Lohar et al. 2013; Panichella et al. 2013) or clone detectors (Wang et al. 2013).

Such a calibration process might be highly biased by the choice of the dataset, *i.e.,* of the set of code contexts. We mitigate this threat by maximizing the dataset diversity, and its representativeness of various programming problems developers could encounter: We collected a set of problems encountered by industrial developers and Master and Bachelor students during laboratory and project activities. For each problem, we asked the subjects to provide a description and the code they produced before requesting or searching for solutions.

We collected 74 code contexts, 48 from academic contexts and 26 from industry. We randomly sampled half of them (37) for the calibration, and used the remaining 37 for the

first evaluation of PROMPTER described in Section 3. For each of the 37 contexts used for the calibration, we browsed Stack Overflow with the aim of finding pertinent, helpful, discussions. More than one discussion could be identified in this phase. The set of relevant documents manually identified represents our "gold standard" to measure the suggestion accuracy of a specific PROMPTER's configuration.

Table 2 reports the configuration that provides the best recommendation accuracy. The indices with value 0.00 have been discarded from the model after completing the calibration. We have used this configuration for the two evaluation studies. Having 74 code contexts available (along with manually identified relevant documents), and having calibrated the model using only 37 of them, we could have used the other 37 contexts as a test set to automatically evaluate the performance of the ranking model. However, such an evaluation would have been biased by our manual validation of the links between contexts and relevant documents. We do not have such a threat in our studies, because the relevance was evaluated by external participants (**Study I**), or where participants used PROMPTER in maintenance and development tasks (**Study II**).

2.5 Putting it Together

The result of the PROMPTERranking model is not sufficient to determine if a discussion is to be recommended or not. As we discussed in Section 2.1, the user can define the sensitivity of PROMPTER in notifying new discussions, and we showed how the *Query Service* determines if a new search is to be triggered or not. Triggering a new search and notifying a discussion relies on two thresholds: (i) *Query Entropy Threshold* and (ii) *Minimum Confidence Threshold*. The former is sent to the *Query Service* and defines the entropy level that should not be exceeded by the median (or mean, depending on the user preferences) of the terms of the query. If the value is below the threshold, a new search is triggered. The latter defines the minimum confidence level needed for a discussion to be recommended.

Both thresholds range in the interval [0, 1]. We limited the interval to [0.1, 0.9] to prevent PROMPTER from not being able to submit new searches or notify new discussions. Whenever one uses the sensitivity slider, these values are modified in an inverse proportional way. A complete slide to the right means a high-sensitive configuration with *Query Entropy Threshold* at 0.9 and *Minimum Confidence Threshold* at 0.1, and the opposite otherwise.

Table 2 PROMPTER Ranking Model: Best Configuration

Index	Weight
Textual Similarity	0.32
Code Similarity	0.00
API Types Similarity	0.00
API Methods Similarity	0.30
Question Score	0.07
Accepted Answer Score	0.00
User Reputation	0.13
Tags Similarity	0.18

3 Study I: Evaluating Prompter's Recommendation Accuracy

RQ$_1$:*To what extent are the Stack Overflow discussions identified by* PROMPTER *relevant?*

The goal of our first empirical study (*Study I*) is to evaluate, from a developer's *perspective*, the relevance of the Stack Overflow discussions identified by PROMPTER, *i.e.,* we are interested in understanding to what extent the retrieved discussion provides useful information to a developer working on a particular code snippet.

3.1 Study Design and Planning

The *context* of the study consists of *participants*, *i.e.,* various kinds of developers, among professionals and students, and *objects*, *i.e.,* source code snippets and its related Stack Overflow discussion as identified by PROMPTER. This study aims at answering the following research question:

We asked 55 people (industrial developers, academics, and students) to complete a questionnaire aimed at evaluating the relevance of the Stack Overflow discussions identified by PROMPTER, by analyzing a specific code snippet. 33 participants filled in the questionnaire by answering the questionnaire through a Web application. They received the URL of the questionnaire, along with instructions, via email. Before accessing the questionnaire,

Table 3 Study I Answers Questionnaire Summary. Percentages for Q3 and Q4 are calculated on the total number for subjects

Question	Answer	Total	Percentage
Job	Industrial Developers	13	39 %
	PhD Students	9	27 %
	Master Students	7	21 %
	Bachelor Students	2	6 %
	Faculty	2	6 %
Q1 : Have you ever worked in industry?	$<$ 3 years	21	64 %
If yes, how long?	3-5 years	3	9 %
	$>$ 5 years	2	6 %
	Never	7	21 %
Q2 : How long have you been	$<$ 3 years	5	15 %
programming in Java?	3-5 years	5	15 %
	$>$ 5 years	22	67 %
	Never	1	3 %
Q3 : What kind of traditional	Javadoc	22	67 %
documentation do you usually use?	Official API Documentation	28	85 %
	Books	9	27%
Q4 : What kind of additional	StackOverflow	26	79 %
resources do you usually use?	Forums	24	73 %
	Mailing List	7	21 %
	Others	7	21 %

participants were required to create an account, with login credentials, and to fill in a pre-questionnaire aimed at gathering information on their background. The answers to this pre-questionnaire are reported in Table 3.

The majority of the participants are industrial developers (39 %) while 79 % of participants declared to have spent some years in industry. Only 18 % of participants have less than three years of experience in Java programming while 67 % have more than five years. Most of participants use Javadoc and API Documentation as traditional documentation, while they mostly rely Stack Overflow and Forums as additional resources. Note that the different background of participants is a *requirement* for this study, since PROMPTER should be able to support developers having different skills, programming knowledge, and experience.

Once the participants answered the pre-questionnaire, they had to perform (up to) 37 tasks where the Web application showed a Java class and a discussion from Stack Overflow that PROMPTER suggested as top-1 ranked discussion among the results retrieved when analyzing that class. Even though participants had the chance of skipping tasks, we obtained at least 30 answers for each task. In the context of this study, we used the remaining 37 code snippets manually collected as explained in the previous section. Participants expressed their level of agreement to the claim *"The code and the Stack Overflow discussion are related"*, providing a score on a five points Likert scale (Oppenheim 1992): 1 (strongly disagree), 2 (disagree), 3 (neutral), 4 (agree), and 5 (strongly agree). In other words, the participants had to indicate to what extent the discussion could help them in completing the implementation task in the showed class.

Figure 6 shows an example of task from our survey. After submitting the score, participants were asked to write an optional comment to explain the rationale for their evaluation. We gave participants four weeks to complete the questionnaire. The participants were neither aware of the experimented technique (*i.e.,* PROMPTER) nor how the Stack Overflow discussions were selected. The Web questionnaire was also designed to (i) show the 37 tasks to participants in random order to limit learning and tiredness effects, and (ii) measure the time spent by each subject in answering each question. Response time was collected to detect participants who provided answers in less than 10 seconds, *i.e.,* without carefully reading code and the Stack Overflow discussion. This was not the case for any participant.

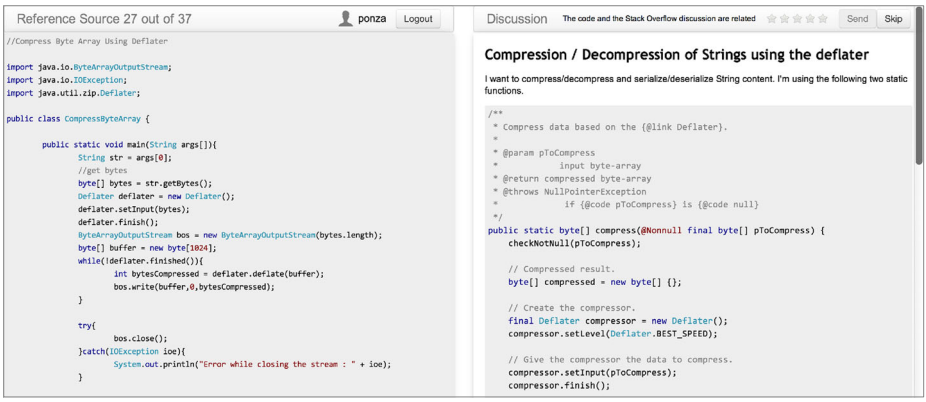


Fig. 6 An Example Question from the Questionnaire Assessing Discussions Retrieved by PROMPTER

3.2 Analysis of the Results

We quantitatively analyzed participants' answers through violin-plots (Hintze and Nelson 1998) to assess the ability of PROMPTER in identifying relevant Stack Overflow discussions given a piece of code. Violin plots combine box-plots and kernel density functions, thus providing a better indication of the shape of a distribution. The dot inside a violin plot represents the median. A thick line is drawn between the lower and upper quartiles, while a thin line is drawn between the lower and upper tails.

Figure 7 shows the violin-plots of scores provided by participants of our experiment to each of the 37 questions composing our questionnaire (*i.e.,* their level of agreement to the claim *"the code and the Stack Overflow discussion are related"*). To understand whether

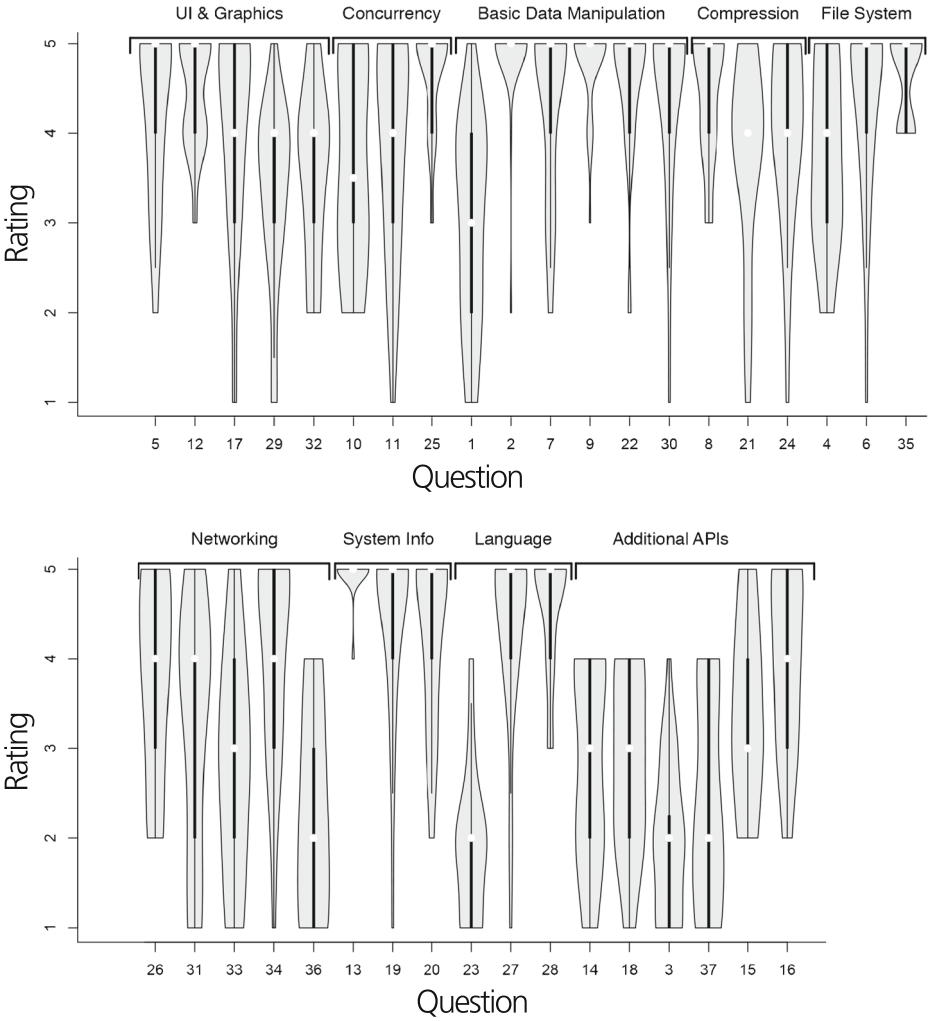


Fig. 7 Violin Plots of Scores Assigned by Participants to the Evaluated Stack Overflow Discussions

PROMPTER excels in particular domains, we grouped the 37 tasks based on the topic/piece of technology they are related to, instead of ordering the tasks by their number.

Overall, the analyzed Stack Overflow discussions have been considered related to the showed Java code snippet. Specifically, 28 out of the 37 analyzed discussions (76 %) received a median score greater or equal than 4. This indicates that participants agreed or strongly agreed to the above reported claim. Among the remaining 9 discussions, 5 (14 %) achieved 3 as median, meaning that participants were generally undecided about their relevance to the code context, and four (10 %) were mostly marked as not relevant achieving a median score of 2 (*i.e.,* disagree). In the following, we discuss two examples in which PROMPTER performed well, and a scenario in which we show its limitations.

Example 1 The question reported in Fig. 6 (question 8 in Fig. 7) is an example where the achieved median score is 5. The class `CompressByteArray`—implementing the compression of a `byte` array using the `Deflater` class—has been linked by PROMPTER to the Stack Overflow discussion **Compression/Decompression of Strings using the deflater**[5]. Among the comments left by developers to their votes, one explained her "strongly agree" vote with the following sentence: *it is a good discussion if working on the `CompressByteArray` class, since it talks about compression with deflater, decompression, but also about problems that could be experienced and possible solutions*.

Example 2 Another Stack Overflow discussion felt by developers as strongly related to the companion Java class was the one entitled **Java regex email**[6] (question 9 in Fig. 7) and associated to the `Utility` class in Listing 2:

```
import java.util.regex.*;

public class Utility {

  public static boolean isValidEmailAddress(String email) {
  //regex to match an e-mail address
     String EMAIL_REGEX = "^[\\w-_\\.+]*[\\w-_\\.]\@([\\w]+\\.)+[\\w]";
     Pattern emailPattern = Pattern.compile(EMAIL_REGEX);
     Matcher emailMatcher = emailPattern.matcher(email);
     return emailMatcher.matches();
  }
}
```

Listing 2 Utility class

The `Utility` class emulates a developer experiencing troubles in writing the method `is-Valid-Email-Address`, aimed at validating through the Java `regex` mechanism an email address provided as parameter. The regular expression stored in variable `EMAIL_REGEX` is wrong, and for this reason `is-Valid-Email-Address` is incorrect.

In the Stack Overflow discussion retrieved by PROMPTER as the most related one to the `Utility` class, a user is asking help since she is experiencing a similar problem when trying to validate an email address using Java `regex`. The top answer in this discussion contains the solution to the problem in method `is-Valid-Email-Address`, *i.e.,* the correct regular expression to validate email addresses. This explains why almost all subjects involved in our study (26 out of 32) assigned a score equal to 5 to this discussion.

[5] http://stackoverflow.com/questions/9542987

[6] http://stackoverflow.com/questions/8204680

Example 3 Developers did not consider particularly useful the discussion **Invoke only a method of a servlet class not the whole servlet**[7] related to the ShoppingCartViewerCookie servlet class (question 36 in Fig. 7). The reason why PROMPTER linked ShoppingCartViewerCookie to this discussion is because it is about servlets, but not about the particular problem the developer wants to solve (*i.e.,* managing cookies). Instead, the discussion explains how to invoke a single method of a servlet. This was also confirmed by one of the participants: *"the SO discussion does not mention how to use cookies"*. This example shows the limits of PROMPTER: It correctly captures the general context of the code (a developer is working on a servlet class), but it fails to identify the problem she is experiencing when trying to implement a specific feature. The same happened in the few cases where our approach obtained low scores (questions 3, 23, and 37 in Fig. 7).

Summary of RQ$_1$. *The Stack Overflow discussions identified by* PROMPTER *are, from a developer's point-of-view, generally considered related to the source code. 76% of the discussions where considered related (median 4) or strongly related (median 5) by developers, while only 10% was considered as unrelated.*

4 Study II: Evaluating Prompter with Developers

The *goal* of this study is to evaluate to what extent the use of PROMPTER can be useful to developers during a development or maintenance task. The *quality focus* is the completeness (and correctness) of the task a developer can perform in a limited time frame, *e.g.,* because of a hard deadline. The *context* consists of *objects*, *i.e.,* participants have to perform two tasks with/without the availability of PROMPTER. We had 12 *participants* (3 BSc and 3 MSc CS students, and 6 industrial developers). Before the study, we screened the participants by using a pre-study questionnaire, asking them about their experience in programming and Java (the study tasks were in Java). The experience was measured in terms of (i) the number of years of Java programming, and (ii) a self-assessment based on a five-points Likert scale (Oppenheim 1992) going from 1 (very low experience) to 5 (very high experience). Also, we asked participants which sources of documentation they generally exploit when programming.

Subjects Analysis All participants have at least 3 years of experience in programming, with a maximum of 12 reached by an industrial developer and a median of 6.5. They have a median of 4 years of Java programming experience. Participants claimed to have a good experience in programming and Java programming with a median of four (high experience) in both cases. Only two BSc students assessed their experience at 3 (medium), while all the others declared a high experience (4). The sources of information mostly exploited by participants when programming are: Stack Overflow (10 participants), Forums (8), Javadoc (8), and Books (6).

Tasks The tasks participants have to perform are one maintenance task and one greenfield development task (*i.e.,* from scratch). The choice of tasks was performed taking into account that, being the study executed within a lab, the tasks could not be too long nor complicated.

[7] http://stackoverflow.com/questions/13509291

On the other side, the tasks could not be too simple, to avoid a "ceiling" effect, *i.e.,* that all participants correctly completed the tasks without problems, regardless of the use of PROMPTER.

Maintenance Task (MT). This task required the implementation of new features in a Java 2D arcade game, where the player controls a spaceship to destroy an attacking alien enemy fleet. In its original implementation, the game directly starts when its `Main` class is executed. We asked participants to perform the following changes:

- **Change 1.** When starting the game, present to the player a home screen containing two buttons named "Start Game" and "Show Best Scores".
- **Change 2.** By clicking on "Start Game", the player can fill in a form composed of a text box labeled with "Specify a Nickname" and a "Go!" button, that allows the user to start the game.
- **Change 3.** When the game is over, the score (*i.e.,* the number of aliens destroyed) must be stored together with the user's nickname in a file named `scores.xml`.
- **Change 4.** By clicking on "Show Best Scores", the player can view the ranking of the top 10 scores achieved ever. This data must be loaded from the previously described file `scores.xml`.

Development Task (DT). For this task the participants had to create from scratch a Java program that, given the URL of a Web page and an e-mail address, converts the HTML page into a PDF and then send it via email to the specified address. The task consisted in three sub-tasks aimed at implementing the following features:

- **Feature 1.** The program shows a form with the following input fields: (i) the URL of an HTML Web page, (ii) an e-mail address, and (iii) the "object field" for the e-mail to be sent.
- **Feature 2.** The program converts the HTML web page at the provided address in PDF, using the three following conversion rules: (i) the content of the HTML tag `<title>` must become the PDF title, using Arial font with a 16 pt size; (ii) the images in the HTML tags `<img>` must be shown center-aligned in the PDF; and (iii) the content of the HTML tag `<p>` must become the PDF body, by adopting as font Arial 12 pt.
- **Feature 3.** Once the PDF is created, it has to be sent as attachment in an e-mail to the specified address, with the specified object.

We did not provide to participants any indication about the strategy to follow in the implementation of the two tasks.

4.1 Research Questions and Variables

The study aims at addressing the following research question:

RQ$_2$: *Does* PROMPTER *helpdeveloperstocompletetheirtaskcorrectly*?

We investigate if the use of PROMPTER helps developers when performing coding activities and in particular to what extent—within the available time frame, and when working with or without PROMPTER—participants are able to correctly complete the task (or part of it).

The dependent variable aimed at addressing RQ_2 is the task completeness. Since it is difficult to automatically evaluate task completeness, we asked two independent industrial developers to act as "evaluators" and to assess task completeness by performing code review

on each task implemented by participants. The evaluators did not know the goal of the study nor which tasks were performed with (without) PROMPTER's support. To help them in the assessment, we provided a checklist aimed at assigning a fixed completeness score to each of the sub-tasks correctly implemented by participants when working on MT and DT. The checklist for the maintenance task was the following:

1. 15 %: The home screen containing the buttons "Start Game" and "Show Best Scores" has been implemented.
2. 25 %: The "Start Game" button correctly works, allowing the player to insert her nickname. Also, by clicking on "Go!" the game correctly starts.
3. 35 %: When the game is over the player score is correctly stored in the `scores.xml` file.
4. 25 %: The "Show Best Scores" button works, by showing the top 10 scores.

The percentage of completeness assigned for each sub-task was proportional to its difficulty and complexity. The evaluators were independent, and conducted a discussion in case of diverging scores. This happened on four out of the 24 evaluated tasks and the divergence was quickly solved by evaluators performing an additional code inspection.

The main factor and independent variable of this study is the presence or absence of PROMPTER. Specifically, such a factor has two levels, *i.e.,*the availability of PROMPTER(PR) or not (NOPR). Other factors that could influence the results are (i) the (possible) different difficulty of the two tasks MT and DT, (ii) the participants' (self-assessed) *Skills* and (iii) *Experience* in Java development, and (iv) the years of *Industrial Experience* (if any) they may have.

4.2 Study Design and Procedure

The study design—shown in Table 4—is a classical paired design for experiments with one factor and two treatments. The design is conceived in a way that:

- each participant worked both with and without PROMPTER's support,
- each participant had to perform different tasks (MT and DT) across the two sessions to avoid learning effect,
- different participants worked with and without PROMPTER in different ordering, as well as on the two different tasks MT and DT.

Overall, this means partitioning participants into four groups, receiving different treatments in the two laboratory sessions. When assigning participants to the four groups, we made sure that their level of experience was (roughly) uniformly distributed across groups.

We carried out a pre-laboratory briefing, in which participants were trained on the use of PROMPTER, and the laboratory procedure was illustrated in details. However, in doing so, we made sure not to reveal the study research questions. In addition, the training was performed on tasks not related to MT and DT to not bias the experiment.

Table 4 Study II: Design

Session	Group A	Group B	Group C	Group D
1	MT-PR	MT-NOPR	DT-PR	DT-NOPR
2	DT-NOPR	DT-PR	MT-NOPR	MT-PR

Participants had to perform the study in two sessions of 90 minutes each (*i.e.,* participants had a maximum of 90 minutes to complete each of the required tasks) interleaved by a 60-minute break to avoid fatigue effects[8]. At the end of each session, each participant provided the code she implemented.

To simulate a real development context, participants were allowed to use whatever they want to complete the tasks including any material available on the Internet. After the study, we collected qualitative information by (i) using a post-study questionnaire and afterwards, by (ii) conducting focus-group interviews.

The post-study questionnaire was composed of: (i) three questions asking if participants used Internet, the suggestions by PROMPTER, and their own knowledge during implementation. To answer these three questions participants used a four points scale, choosing between *absolutely yes*, *more yes than no*, *more no than yes*, and *absolutely no*; and (ii) a question asking participants to evaluate the relevance of the suggestions generated by PROMPTER. In this case, we adopted a five-points Likert scale (Oppenheim 1992) going from 1 (totally irrelevant) to 5 (very relevant).

During the focus-group interview, two of the authors and all participants discussed together about PROMPTER, trying to point out its weaknesses and strengths. This interview lasted 45 minutes.

4.3 Analysis Method

In the following, we describe all the statistical procedures used to analyze data of this study and address $\mathbf{RQ}_2$. Analyses have been performed using the *R* statistical environment (Core Team 2012). For all the used statistical tests, we consider a significance level $\alpha = 0.05$.

First, we provide an overview of the distribution of task completeness values for the two treatments PR and NOPR using box plots. In this study box plots are preferred over violin plots since they allow to better focus on the comparison between the completeness achieved by participants with the two treatments by not including visual details that violin plots provide. Then, we statistically compare results achieved with the two treatments. Given the chosen (paired) design, we test the null hypothesis:

H_0: there is no statistically significant difference between the completeness achieved with and without PROMPTER*'s support.*

using the non-parametric Wilcoxon signed-rank test (Sheskin 2007). This is a paired test, to be used when we need to compare related samples, as in our case where we need to compare the completeness achieved with and without PROMPTER. Since we do not know a priori in which direction the difference should be observed, we use a two-tailed test.

Besides testing the presence of a significant difference, we also assess the magnitude of the observed difference using the Cliff's delta (d) effect size (Grissom and Kim 2005) which is an effect size measure suitable for non-parametric data. The Cliff's d is defined as the probability that a randomly-selected member of one sample has a higher response than a randomly selected member of a second sample, minus the reverse probability. Cliff's d ranges in the interval $[-1, 1]$ and it is considered small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

To investigate whether PROMPTER helps differently for maintenance or development tasks, we pairwise compare results achieved for different tasks using the Mann-Whitney U test (equivalent to the Wilcoxon Rank Sum test) (Sheskin 2007). In this case we use an

[8]During the break participants did not have the chance to exchange information among them.

unpaired test, because we cannot compare related samples, since each participant performed a development task with PROMPTER and a maintenance task without PROMPTER, or *vice versa*. Since here multiple tests have been performed, p-values have been adjusted using Holm's correction (Holm 1979). This procedure sorts the p-values resulting from n tests in ascending order of values, multiplying the smallest by n, the next by $n - 1$, and so on.

Finally, we used permutation test (Baker 1995) to check, from a statistical standpoint, the influence of the various co-factors and their interaction with the main factor treatment. The permutation test is a non-parametric alternative to the two-way Analysis of Variance (ANOVA); differently from ANOVA, it does not require data to be normally distributed. The general idea behind such a test is that the data distributions are built and compared by computing all possible values of the statistical test while rearranging the labels (representing the various factors being considered) of the data points. We used an implementation available in the *lmPerm R* package. We have set the number of iterations of the permutation test procedure to 500,000. Since the permutation test samples permutations of combination of factor levels, multiple runs of the test may produce different results. We made sure to

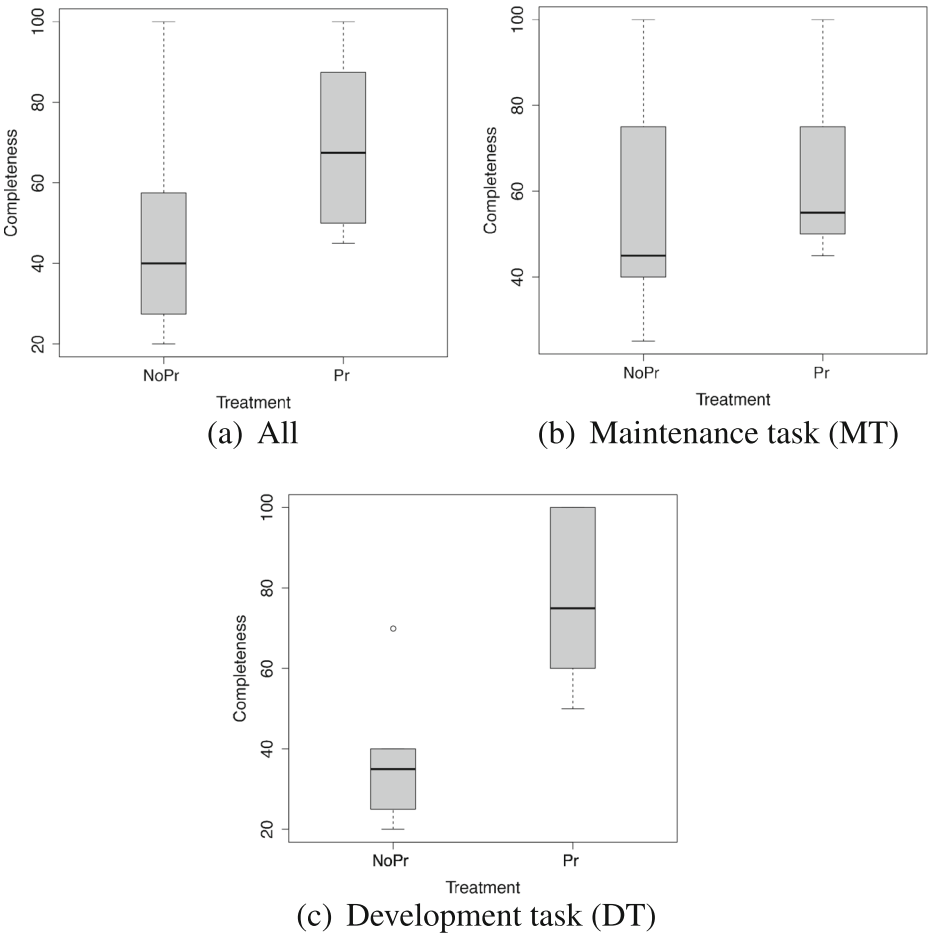


Fig. 8 Boxplots of Completeness achieved by Participants with (Pr) and without (NoPr) PROMPTER

choose a high number of iterations such that results did not vary over multiple executions of the procedure.

4.4 Quantitative Analysis of the Results

Figure 8(a) shows boxplots of completeness achieved by participants when using (PR) and when not using (NOPR) PROMPTER. As it can be noticed, participants using PROMPTER were able to achieve a level of completeness greater than those not using it. The PR median is 68% (mean 70%) against the 40% median (mean 46 %) of NOPR. In other words, PROMPTER allowed participants to achieve a median additional correctness of 28 % (mean of 24 %). The Wilcoxon paired test indicates the presence of a statistically significant difference, with a p-value lower than 0.01, hence rejecting H_0. The Cliff's d is 0.65, indicating a *large* effect size.

Figure 8b and c show boxplots of completeness when focusing on results achieved for MT and DT, respectively, to better understand where PROMPTER results particularly helpful. PROMPTER helped participants in both MT and DT, increasing the median completeness achieved for MT of 10 %, and for DT of 40 %. The results of the Mann-Whitney unpaired two-tailed test indicates that for MT the difference is not significant (p-value=0.23) and the effect size is 0.38 (*medium*), while there is statistically significant difference for DT (p-value=0.03), with a *large* effect size (0.88). PROMPTER produced much more benefits for DT, where participants implemented from scratch and where they had to use several libraries, *e.g.,* to parse the HTML page, to convert it in PDF, to send an e-mail. In such a circumstance, PROMPTER provided an effective support by pointing to Stack Overflow discussions concerning the correct usage of such libraries.

Concerning the effect of all other co-factors, the permutation test results, reported in Table 5, indicate that:

- *Java Skills* and *Experience* have a significant effect on the participants' performance, although they do not interact with the main factor. In other words, people with higher skills and experience perform better, independently of PROMPTER;
- There is no effect nor interaction of the *Industry Experience*;
- *Task* has no direct effect on the observed results. It marginally interacts with the main factor: As we have also explained above, and as one could have expected from Figs. 8b and c, PROMPTER resulted more helpful for DT than for MT.

4.5 Qualitative Analysis of the Results

Results from the post-questionnaire provided us with interesting observations. First, participants generally used Internet during the implementation of the required tasks. When being asked, six of them answered *absolutely yes*, five *more yes than no*, and one *more no than yes*. This is expected and consistent with the answers they provided to the pre-study questionnaire. Second, most participants felt to have used their knowledge in the tasks implementation, with four of them answering *absolutely yes*, six *more yes than no*, and two *more no than yes*.

As for the question related to the use of PROMPTER's recommendations, most of participants answered positively. Three of them answered *absolutely yes*, eight *more yes than no*, and two *more no than yes*. The latter participants explained that they received very few PROMPTER's recommendations, due to the fact that they spent much time on the Internet,

Table 5 Effect of co-factors and their interaction with the main factor: results of permutation test

		Java Skills			
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	0.01
Java Skills	1	2566.87	2566.87	500,000	0.02
Treatment:Java Skills	1	1.88	1.88	58,583	0.94
Residuals	20	8487.50	424.38		
		Java Experience			
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	0.01
Java Experience	1	2276.87	2276.87	500,000	0.03
Treatment:Java Experience	1	72.70	72.70	462,783	0.68
Residuals	20	8706.68	435.33		
		Industry Experience			
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	0.02
Industry Experience	1	192.30	192.30	500,000	0.55
Treatment:Industry Experience	1	35.32	35.32	248,801	0.80
Residuals	20	10828.63	541.43		
		Task			
	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	1	3384.38	3384.38	500,000	0.02
Task	1	26.04	26.04	225,011	0.82
Treatment:Task	1	1426.04	1426.04	500,000	0.10
Residuals	20	9604.17	480.21		

trying to figure out how to implement the required tasks. This resulted in wasted time during which the code they were working on was untouched, leading PROMPTER to wait in vain for their moves to produce suggestions. Still, these two participants agreed that the few received recommendations were actually relevant to what they were implementing in the IDE.

The goodness of the PROMPTER's recommendations perceived by participants is evident when analyzing the answers to the question concerning the relevance of the Stack Overflow discussions pushed by PROMPTER in the IDE. Among the twelve participants, two of them classified the suggestions as *very relevant* (5), and the remaining ten as *relevant* (4).

We gained useful insights from the focused group interview. Participants agreed that PROMPTER is very useful when working on tasks in which the developer has poor experience, since the information brought in the IDE by PROMPTER helps the developer in enriching her knowledge about the task to be performed. For instance, one of the participants was experiencing problems with the repaint function provided in the `JFrame` by the `updateUI` method. PROMPTER pushed in his IDE a Stack Overflow discussion[9] exactly related to what he was trying to implement, solving his problem. Another participant, when

[9] http://stackoverflow.com/questions/11640494/

starting to work on DT, observed the push notification of PROMPTER about a Stack Overflow discussion[10] providing guidelines on how to choose the HTML parser library to use. After reading the discussion, his choice was targeted on `jsoup`. Summarizing, participants identified the following PROMPTER strengths:

- the accuracy of the suggestions and the relevance of the suggested Stack Overflow discussions;
- the user interface: clean, clear, and not invasive;
- the ease of use, and minimal training required;
- the possibility to tune the sensitivity of PROMPTER, increasing or reducing the rate of suggestions.

Besides identifying PROMPTER strengths, the focused group interview we conducted allowed us to also identify PROMPTER's limitations. Specifically, participants would like to see the following improvements in future PROMPTER releases:

- the possibility to exploit information coming not only from Stack Overflow, but also from forums and programming tutorials available online; the current ranking model implemented in PROMPTER considers features that are typical of Stack Overflow discussions (*e.g.,* question score, accepted answer score, *etc*), and thus it cannot be generalized to other sources of information as it is. However, the PROMPTER architecture allows to easily include additional ranking models customized to exploit useful data from specific sources of information (*e.g.,* other Q&A websites).
- a way to force PROMPTER to look for specific types of discussions on Stack Overflow. For example, participants would like the possibility to specify some key terms that should always be considered by PROMPTER when searching for discussions on Stack Overflow;
- the possibility to have a search field. Indeed, most participants agreed on the fact that PROMPTER loses its usefulness if the developer has no idea on how to start coding. In such a situation, the developer is forced to leave the IDE and surf the Web. Participants suggested the addition of a search field in the PROMPTER user interface that allows one to explicitly formulate and execute a query without leaving the IDE. As explained in Section 2.1, as a result of participants' feedback, the current version of PROMPTER already implements a search field for manual queries.

Summary of RQ$_2$. *Quantitative results indicated that overall* PROMPTER *allowed participants to achieve a significantly better completeness of the assigned tasks. The collected feedbacks suggested that participants perceived the tool as usable, the suggestions accurate and not invasive. Eleven out of the twelve participants involved in our study claimed that they would like to use* PROMPTER *in their daily development activities.*

5 PROMPTER: one Year Later

One of the main challenges when dealing with recommenders like PROMPTER is the variability of the information available. PROMPTER is mainly based on a Q&A website and

[10] http://stackoverflow.com/questions/3152138/

search engines. Thus, given the continuous growth of the web and its contents, our goal is to investigate issues that pertain to the persistence of such information and the subsequent replicability of the evaluation of approaches and tools based on information available on online forum.

In Section 3 we presented a study aimed at validating the usefulness of the ranking model exploited by PROMPTER (see Section 2.4). In such a study the PROMPTER's ranking model was used to recommend Stack Overflow's discussions for 37 development tasks (*i.e.,* code snippets). Then, the relevance of the top-ranked discussion to its related task was judged by the involved participants. After one year, we replicated *Study I* using the information available online at the time of writing. In particular, *Study I* as described in Section 3 has been carried out in July 2013, while its replication (described in this section) has been performed in July 2014.

The *goal* is to replicate the retrieval of top recommendations for the 37 tasks considered in Section 3 using PROMPTER, with the *purpose* of investigating (i) the replicability of the study we performed and, above all (ii) to what extent the PROMPTER's recommendations—and consequently, its performance—may vary over time. The *quality focus* is the performance variability over time of recommender systems relying on online resources, and in particular of PROMPTER which relies on Stack Overflow and on search engines. The *context* consists of the same 37 development tasks considered in Section 3, and a pool of 18 people (industrial developers, academics, and students) evaluating the relevance of the Stack Overflow discussions retrieved by PROMPTER in July 2013 and July 2014.

Research questions The research questions this study aims to answer are:

- **RQ$_3$:** *To what extent are the Stack Overflow discussions identified by* PROMPTER *in July 2013 still relevant in July 2014?*
- **RQ$_4$:** *How is the developers' assessment of the new recommendations compared to those identified one year before?*

The first research question (**RQ$_3$**) aims at posing the premises of this study, *i.e.,* to investigate whether starting from the same 37 code snippets used one year ago, PROMPTER recommends a different Stack Overflow discussion for some of them (which of course may represent a better or a worse recommendation). The results of **RQ$_3$** show that among the 37 tasks which we re-run PROMPTER on, some resulted in a different recommendation as compared to the recommendation obtained one year before. Thus, **RQ$_4$** aims at comparing the assessments provided by participants to both recommendations, *i.e.,* the one provided by PROMPTER in July 2013 and in July 2014.

5.1 Study Design and Analysis Method

The first step to answer **RQ$_3$** is to re-ask PROMPTER to retrieve and rank Stack Overflow discussions for each of the 37 tasks. This results in a ranked list of Stack Overflow discussions for each of the 37 tasks.

To verify the replicability of *Study I*, we check for each of the 37 tasks in which position of its related ranked list has been retrieved the Stack Overflow discussion recommended by PROMPTER one year before as the most relevant (*i.e.,* first in the ranked list). For matter of fairness, the ranking model was tuned exactly as in *Study I*. Also, we exploited the same entropy information we computed one year before on the Stack Overflow dump of June 2013. Our choice was dictated by the fact that the number of Stack Overflow discussions

present in that dump was already huge (5,016,480), including a total of 105,439 different terms.

In a perfect scenario, where *Study I* is fully replicable, the top Stack Overflow discussion retrieved by PROMPTER in July 2013 for each of the 37 tasks should be still ranked in first position in July 2014. In other words, the study should be time independent.

As for $\mathbf{RQ}_4$, we need to compare human-based assessment of the new recommendations with the assessment provided to the old recommendations. Such a comparison is performed only for the tasks on which the top-ranked Stack Overflow discussion has changed after one year. This is because we cannot just ask the study participants to assess the new recommendations and compare such assessments with those obtained in the previous study. This is because the assessment of the old recommendations has been performed by different people. Thus, the results could be influenced by subjectiveness or personal levels of experience/skills. To limit this threat, we asked the participants of this study to evaluate both the old and new recommendations. Based on the results of $\mathbf{RQ}_3$, this study is limited to those tasks (29 in total) for which the top-ranked Stack Overflow discussion provided by PROMPTER (*i.e.,* the recommended one) changed between *Study I* and this replication. Overall, study participants assessed 58 recommendations. We adopted the same instrumentation and set-up described in Section 3 by reusing the same web application to collect participants evaluations of the PROMPTER's recommendations. Again participants were required to create an account and to fill in a pre-questionnaire aimed at gathering information on their background. Of the 30 invited people, 18 completed our questionnaire. Their answers are reported in Table 6.

Table 6 Replication Study Answers Summary

Question	Answer	Total	Percentage
Job	Industrial Developers	5	28 %
	PhD Students	3	17 %
	Master Students	6	33 %
	Bachelor Students	3	17 %
	Faculty	1	6 %
Q1 : Have you ever worked in industry?	< 3 years	5	28 %
If yes, how long?	3-5 years	2	11 %
	> 5 years	1	6 %
	Never	10	56 %
Q2 : How long have you been	< 3 years	5	28%
programming in Java	3-5 years	7	39%
	> 5 years	6	33%
	Never	0	0%
Q3 : What kind of traditional	Javadoc	14	78%
documentation do you usually use?	Official API Documentaiton	14	78 %
	Books	5	28 %
Q4 : What kind of additional	StackOverflow	18	100 %
resources do you usually use?	Forums	14	78 %
	Mailing List	0	0 %
	Others	1	6 %

Five of the involved participants are industrial developers, while eight declared to have some years of industrial experience (Q1). Only five participants have less than three years of experience in Java programming (Q2) and most of them rely on Javadoc and API documentation as standard sources of documentation (Q3) and on Stack Overflow and forums as additional resources (Q4).

Following the same setup used in *Study I*, we make use of violin plots to summarize the results. Also, we compare the assessment distribution for each task between the old and the new recommended Stack Overflow discussion by using Wilcoxon paired tests (two-tailed). In addition to that, we report the Cliff's d (paired) effect size measures of such comparisons. Note that, by design, this study requires a paired analysis, because for each task each participant evaluates both the old and new recommendation generated by PROMPTER.

5.2 RQ_3: To what Extent are the Stack Overflow Discussions Identified by Prompter in July 2013 Still Relevant in July 2014?

Table 7 reports in column PROMPTER the rank assigned by PROMPTER in July 2014 to the top-ranked discussion retrieved in July 2013 (from now on, *old top-discussion*) for each of the 37 tasks object of our study. In particular:

- if the value in column PROMPTER is "1", this means that the top-retrieved Stack Overflow discussion for the specific code snippet (task) did not change after one year;
- if the value in column PROMPTER is x with $x > 1$, this means that the top-retrieved Stack Overflow discussion for the specific code snippet changed after one year, and the *old top-discussion* is now ranked in a lower position (x);
- if the value in column PROMPTER is "-", this means that the *old top-discussion* is not retrieved at all by PROMPTER one year later (*i.e.,* it is not present in the ranked list of Stack Overflow discussions generated by PROMPTER).

Table 7 also reports the rank of the *old top-discussion* in the Google and Bing search engines in July 2014 (*i.e.,* one year after *Study I*) to better analyze the cases where PROMPTER was not able at all to retrieve the *old top-discussion* ("-" in column **Prompter**). As explained in Section 2, Google and Bing are exploited by PROMPTER to retrieve the Stack Overflow discussions to rank. Thus, if Google and Bing are not able to retrieve the *old top-discussion* for a task one year later, as a consequence also PROMPTER will not be able to retrieve it We consider a Stack Overflow discussion as not retrieved by Google and Bing if it does not appear in the first 100 retrieved Stack Overflow discussions.

Table 7 Top-rated Stack Overflow discussions re-ranked by PROMPTER one year later

Task	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Google	-	1	-	-	-	-	10	-	1	-	-	4	6	9	43	-	-	4	-
Bing	3	-	-	-	1	-	-	10	-	-	1	-	-	-	-	5	-	-	2
Prompter	1	1	-	-	9	-	15	3	1	-	11	5	1	3	28	3	-	3	2

Task	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
Google	25	12	-	-	-	7	-	3	-	-	53	-	-	-	-	1	-	-
Bing	-	-	1	-	-	-	-	-	9	-	-	1	-	-	-	-	-	1
Prompter	26	1	1	-	-	1	-	2	4	-	2	59	-	-	-	1	-	4

The numbers shown in Table 7 highlights as only for eight of the 37 tasks PROMPTER retrieves the same top-ranked discussion as one year before (*i.e.,* tasks 1, 2, 9, 13, 21, 22, 25, 35). This means that after one year, PROMPTER recommends a different Stack Overflow discussion for 78 % of the tasks, highlighting a low replicability of *Study I* just one year after.

Analyzing the 29 tasks where the recommendation provided by PROMPTER changed, it emerges that in 13 of them (45 % of cases) PROMPTER was not able to retrieve the *old top-discussion* because the employed search engines did not retrieve it anymore (*i.e.,* tasks 3, 4, 6, 10, 17, 23, 24, 26, 29, 32, 33, 34, 36—see Table 7). This could be due to several reasons, such as (i) the presence after one year of more related Stack Overflow discussions for the specific task, (ii) the deletion of the *old top-discussion* from Stack Overflow, or (iii) changes in the ranking algorithm exploited by the search engines. Despite the underlying reason(s) for such a result, it is clear that the recommendations produced by tools relying on search engines like PROMPTER are strongly influenced by changes in the output of such engines, undermining the replicability of any type of evaluation.

Concerning the remaining 16 tasks where the PROMPTER recommendation changed, in six cases (*i.e.,* tasks 8,14,16,18,19,27) the *old top-discussion* is still ranked in the top-three positions, even if it is no more the top-discussion. While for other kinds of recommender this result might show some sort of stability in the recommender's behavior (*e.g.,* tools using Stack Overflow discussions to document the source code (Vassallo et al. 2014)), given the *push mechanism* implemented in PROMPTER (*i.e.,* PROMPTER just pushes in the IDE the top-ranked Stack Overflow's discussion) also these cases represent a total different behavior by our tool at one year of distance. The situation is even more marked on the remaining ten tasks where the *old top-discussion* is ranked, in July 2014, in a much lower position.

Summary of RQ$_3$. *The recommendations provided by* PROMPTER *starting from the same 37 tasks exploited in* Study I *changed in 78% of cases when replicating the study one year later. This clearly highlights that (i) the performance of recommenders relying on volatile information mined from the Web can strongly change over time and (ii) empirical studies performed to evaluate such tools are almost not replicable. The results of RQ$_3$ pave the way to our RQ$_4$, where we investigate if the 78% tasks for which* PROMPTER *recommends a different Stack Overflow discussion results in an improvement or in a worsening of* PROMPTER*'s performance.*

5.3 RQ$_4$: How is the Developers' Assessment of the new Recommendations Compared to Those Identified one Year Before?

Figure 9 reports violin plots related to the assessment provided by the study participants to the old (red) and new (blue) recommendations. For the *old top-ranked* discussions (red violin plots), 18 out of 29 (62 %) of the proposed discussions received a median score greater or equal than 4, that is, people agreed (31 %) or strongly agreed (31 %) on the statement *"The code and the Stack Overflow discussion are related"*. Of the remaining 11 discussions, 17 % received a rating between 2 and 4, meaning that people were generally undecided, while 21 % received a rating lower or equal than 2.

For the *new* recommendations proposed by PROMPTER(blue violin plots), the results obtained are slightly worse than the previous one, but they seem to follow the same trend. Indeed, 15 out of 29 (52 %) of the proposed discussions received a median score greater or equal than 4 (of those, 34 % received a median of 5, while 17 % a median of 4). For the

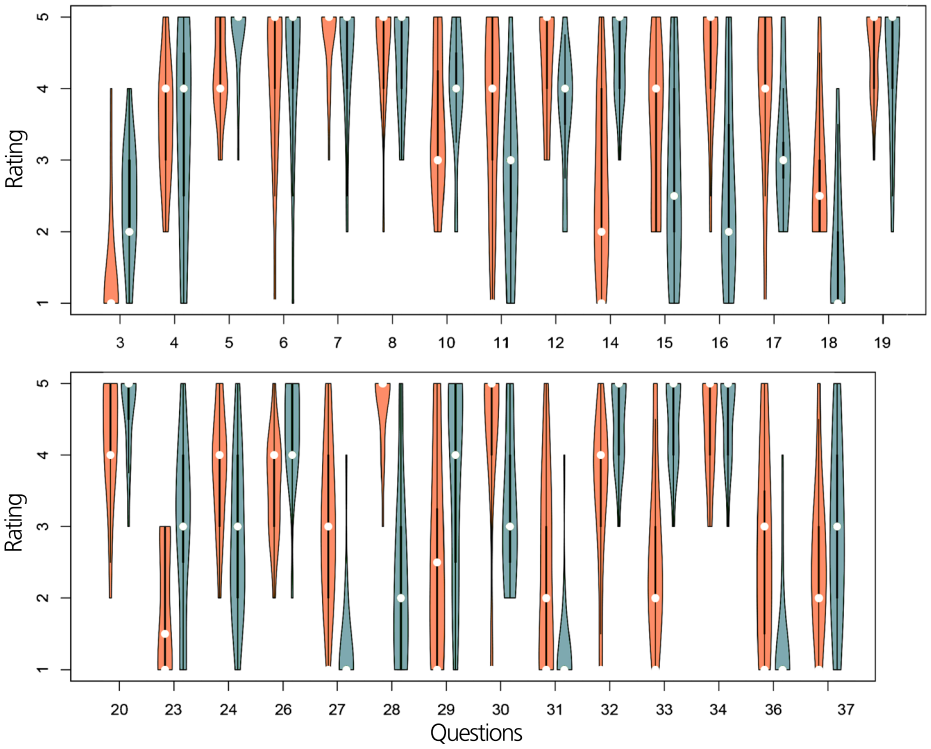


Fig. 9 Violin Plots of Scores Assigned by Participants to the Old (*red*) and New (*blue*) Top-ranked Stack Overflow discussion

remaining discussions, they equally (24 %) received a median score either between 2 and 4 (participants undecided on the quality of the pushed Stack Overflow discussion), and lower or equal than 2.

Table 8 reports the results of the Wilcoxon tests (p-values) and Cliff's d effect size measures when comparing the ratings assigned by participants to the old and the new top-ranked discussions for each of the 29 object tasks.

Based on the results of the Wilcoxon test, we divided the tasks in three categories:

(i) *Improved*, seven tasks for which the new top-ranked Stack Overflow discussion achieves a statistically significant better user evaluation;
(ii) *Neutral*, eleven tasks where none of the two top-ranked discussions is significantly better than the other as assessed by participants; and
(iii) *Worse*, eleven tasks where the old recommendation provided by PROMPTER in *Study I* achieved a better user evaluation compared to the new one.

In the following we discuss the results achieved organizing the discussion by considering the above "group of tasks" (*i.e.,* Improved, Neutral, and Worse) trying to understand what happened in one year, what affected the model and produced different recommendations given the same task (*i.e.,* code snippet).

Improved This category includes all the tasks for which the new recommendation achieved a better participants' assessment as compared to the old one. As reported in

Table 8 Mann-Whitney test (p-value) and Cliff's delta (*d*). The recommendation achieving the better user' evaluations is reported in the second column: new (new recommendation), old (old recommendation), tie (not statistically significant difference)

Status	ID	Best	p-value	d
	3	new	0.0050	0.71
	5	new	0.0147	0.42
	14	new	0.0036	0.78
	23	new	0.0261	0.50
	26	new	0.0312	0.56
	32	new	0.0085	0.57
Improved	33	new	0.0028	0.75
	4	tie	0.7485	0.00
	6	tie	0.9319	0.05
	7	tie	0.1736	0.14
	8	tie	0.3796	0.20
	10	tie	0.1581	0.24
	19	tie	0.3053	0.16
	20	tie	0.3429	0.21
	29	tie	0.0685	0.41
	31	tie	0.0719	0.48
	34	tie	0.5653	0.08
Neutral	37	tie	0.3586	0.19
	11	old	0.0334	0.44
	12	old	0.0070	0.59
	15	old	0.0146	0.31
	16	old	0.0036	0.69
	17	old	0.0282	0.51
	18	old	0.0017	0.59
	24	old	0.0332	0.48
	27	old	0.0020	0.78
	28	old	0.0014	0.82
	30	old	0.0031	0.73
Worse	36	old	0.0054	0.64

Table 8, for six of the tasks belonging to this category (all but task 5) the new recommendations were highly preferred by participants with respect to the old recommendations (the effect size is greater than 0.474, *i.e.,* a large effect size).

We manually analyzed each of the new recommended Stack Overflow discussions in this set to understand why the recommended Stack Overflow discussion changed after one year. A very simple explanation can be given for tasks 32 and 33 where the new recommended Stack Overflow discussions have been added on Stack Overflow in November 2013 and August 2013 respectively, *i.e.,* after the dataset construction for *Study I* was already completed. Thus, the new top-ranked discussions for these two tasks simply did not exist when *Study I* was carried out in July 2013.

For the remaining five tasks of this set, the *old top-ranked* Stack Overflow discussions have not been modified since the construction of the data set for *Study I* and the *new top-ranked* discussion already existed at the time *Study I* was performed. However, in all these cases PROMPTER assigns (in July 2014) a higher confidence level to the *new top-ranked* discussion. This is because in July 2013, when *Study I* was performed, the *new top-ranked* discussions were not retrieved by the search engines exploited by PROMPTER (otherwise the *new top-ranked* discussions would have been pushed by PROMPTER also in July 2013). This is likely due to changes in the ranking algorithms of the exploited search engines.

Neutral This category includes all tasks for which there is no statistically significant difference in the assessment provided by participants to the old and the new PROMPTER's recommendations. The manual analysis of the old and new retrieved discussions highlights as for six of the tasks in this group (*i.e.,* tasks 6, 7, 8, 20, 34, 37) the reason for the change in recommendation is the same discussed above for the "Improved" group, *i.e.,*, in July 2013 the search engines exploited by PROMPTER did not retrieve the *new top-ranked* discussion.

The change in recommendation for tasks 4, 10, and 31 have a similar reason: the *old top-ranked* discussion exhibits a higher confidence level than the *new* one as assessed by the PROMPTER's ranking model. However, for these three tasks the search engines used by PROMPTER do not retrieve anymore the *old top-ranked* discussion in July 2014. Despite this, as assessed by participants, there is no clear difference in the quality of the old and the new recommendations.

The last two tasks in this group are those numbered with 19 and 29. For both of them the *new top-ranked* discussion already existed and was also retrieved by the search engines at the time of *Study I* (*i.e.,* July 2013). However, the PROMPTER's ranking model assigned a higher confidence level to the *old top-ranked* with respect to the *new* one at that time. The situation changed at one year of distance due to modifications applied in this time period to the *new top-ranked* discussion that pushed up its confidence level. To better understand, Table 9 reports the model dump for task 19 for (i) the *new top-ranked* in July 2014 (column "New"), (ii) the *old top-ranked* in July 2013 as well as in July 2014, since its confidence level was unchanged during the year (column "Old"), and (iii) the *new top-ranked* in July 2013 (column "Original New").

In the time period going between *Study I* and its replication, both the *new* and the *old top-ranked* discussion received up votes. Note that the up votes affect the parameter *Question Score* exploited by the PROMPTER's ranking model. The *old top-ranked* discussion received six up votes, while two new up votes were assigned to the *new top-ranked* between July 2013 and July 2014. Given the normalization used, the increment in up votes for the *old top-ranked* discussion did not affect its overall confidence level. Indeed, the value of the

Table 9 Model Dump for Task 19

Metric	New	Old	Original New
API Methods Similarity	1.00	1.00	1.00
User Reputation	0.00	0.00	0.00
Tags Similarity	0.75	0.67	0.75
Question Score	0.61	1.00	0.17
Textual Similarity	0.30	0.22	0.30
Confidence	57.52 %	56.04%	54.47 %

Table 10 Model Dump for Task 15

Metric	New	Old	Original New
API Methods Similarity	1.00	0.00	1.00
User Reputation	0.00	0.00	0.00
Tags Similarity	0.67	0.67	0.00
Question Score	0.17	1.00	0.17
Textual Similarity	0.25	0.25	0.25
Confidence	51.18 %	26.85 %	39.19 %

Question Score parameter for the *old top-ranked* discussion was already equal to 1.00 (*i.e.,* the maximum score) when *Study I* was carried out (see Table 9). On the other hand, the change in the *Question Score* for the new top-ranked discussion allows its confidence value to increase from 54.47 % up to 57.52 %. This resulted in the overtaking of the *new top-ranked* discussion over the *old* one in July 2014. However, as highlighted by the very similar confidence levels and as also confirmed by participants, the two discussions have a very similar relevance for their related task.

Worse This category includes all tasks for which the *old top-ranked* discussion was better assessed by participants as compared to the *new* one. For all these tasks but number 15, the Cliff's delta obtained in the comparison is greater than 0.474, *i.e.,* a large effect size.

Also in this case, we looked into the different tasks to understand what driven the change in the PROMPTER's recommendation. Task 27 is the only one where PROMPTER recommends in July 2014 a Stack Overflow's discussion that did not exist at the time of *Study I* (one year before). For task 31 a change in the tags of the *old top-ranked* discussion happened in the year going from July 2013 to July 2014 has caused a decrease of its confidence level (and in particular of the *Tags Similarity* parameter), consequently resulting in the recommendation of the *new top-ranked*.

The case of task 15 is particularly interesting. Indeed, between *Study I* and its replication both the *old* and the *new top-ranked* discussions have been modified. In particular, the *old* discussion received 8 up votes, causing an equivalent increment in the overall score, while the *new* discussion has been modified in the tags and in the body, thus altering the metrics *Tags Similarity* and *Textual similarity*.

Table 10 reports a dump of the model values for task 15. As before, column "New" shows the model values for the *new top-ranked* discussion in July 2014; column "Original New" reports the model values for the *new top-ranked* discussion when *Study I* was performed; and column "Old" shows the model values for the *old top-ranked* discussion.

Despite the changes applied to the *old top-ranked* discussion its model remained stable between *Study I* and its replication since its *Question Score* was already equals to 1.00 (the maximum) in July 2013.

The values reported in Table 10 clearly show as already in July 2013 the confidence level for the *new top-ranked* discussion was higher than the confidence level for the *old* one (49.22 % *vs* 26.85 %). This means that the *new top-ranked* discussion was simply not retrieved by the exploited search engines at the time of *Study I*.

Finally, discussions related to tasks 11,12,16,17,18, 24, 28, 30, and 36, have not been modified since July 2013. Thus, also in this case the PROMPTER recommendations have

been strongly influenced by the different results generated by the search engines in the two different time periods.

Summary of RQ$_4$. *While the recommendations provided by* PROMPTER *one year later from Study I changed in 78% of cases, its performance did not show strong deviations, with just 24% (against the old 21%) of the new recommended Stack Overflow's discussions classified by participants as not related to the task at hand. Manual analysis suggests that the changes in the search engines together with the volatility of the information exploited by* PROMPTER*, represent the main reasons for 78% of different recommendations after just one year.*

6 Threats to Validity

Construct Validity Threats to *construct validity* are related to the relationship between theory and observation. In *Study I* and in its replication, such threats are mainly due to (i) the fact that we mimic the code being written by a user by providing with PROMPTER a partially-complete class, and (ii) by letting the users provide evaluations using a Likert scale. Concerning the former, we made sure such classes were not too detailed nor too empty, to represent realistic situations where PROMPTER could be used. Concerning the latter, this is a standardized evaluation scale used to collect participants' feedbacks. Having said that, *Study II* overcomes the limitations of *Study I* mentioned above. In *Study II* threats to construct validity are due to how we measured the task completeness. Certainly, we could have used a test suite to measure the completeness in a objective manner. Conversely, code inspection allows to evaluate partial implementations. In addition, the use of a checklist and multiple independent evaluators limited the bias and subjectiveness.

Internal Validity Threats to *internal validity* are related to factors that could have influenced the results. For *Study I* one factor to be considered is the knowledge of the participants—not known a-priori—of the APIs being used in the particular task. The availability of multiple participants with different degree of experience mitigates this threat. Also, note that students taking part in our evaluation were not evaluated based on the task outcome, and we asked participants not to use other sources of information during the task, *e.g.,* to use them as a comparative source to the provided discussion. In *Study II*, to limit the effect of participants' skills and experience, we have pre-assessed them and used this information assigning them to the four groups. We also analyze to what extent the usefulness of PROMPTER depends on the particular task.

For the replication of *Study I*, there could be confounding factors that could have influenced results of both **RQ**$_3$ (different recommendation rankings) and **RQ**$_4$ (developers' assessment). Specifically, for what concern the ranking, we cannot exclude that the different position (or the total disappear) of a question from the search-engine rankings may depend on changes/optimization in the search engines themselves. Nevertheless, we believe this can be one of the factors that affect the volatility of recommenders' results, and that one cannot control.

For what concerns **RQ**$_4$, it is possible that for the same recommendation (already assessed in *Study I*) different subjects gave a completely different evaluation. Hence, such a recommendation could have been judged as very relevant in *Study I*, and not relevant at all in the replication, or vice versa. This can happen especially because of the large difference of experience they have (*Study I* participants are more expert and they might require

more advanced suggestions, whereas participants to the replication might prefer basic ones). To verify whether such a situation could have occurred, we statistically compared—using Mann-Whitney tests (two-tailed)—the ratings provided to the 29 recommendations by participants to *Study I* and by participants to *Study I replication*). Results indicate the presence of a significant difference only for tasks 11 (p-value=0.03, median old study=4, median new study=3), 27 (p-value=0.0001, median old study=5, median new study=3), and 31 (p-value=0.02, median old study=4, median new study=2).

Conclusion Validity For *Study I* we mainly report descriptive statistics and violin plots of the collected results, along with participants' feedbacks, while for its replication, whenever possible, we use appropriate statistical procedures, namely Wilcoxon paired tests and Cliff's d effect size measures. For *Study II*, we used distribution-free tests (Wilcoxon, Mann-Whitney, and permutation test) and effect size (Cliff's d) measures, suitable for limited data sets as in our study. Also, whenever multiple tests are used on the same data, we apply p-value adjustment using the Holm's procedure (Holm 1979).

External Validity Threats to *external validity* concern the generalizability of our findings. In terms of participants, the study involved both professionals and students, with different degree of experiences. Therefore, we claim the study provides a good coverage of the potential categories of PROMPTER users, although further studies with more participants are desirable. In terms of objects, we selected 37 tasks being different in terms of nature and required technical knowledge. However, we cannot exclude that our results depend on the particular choice of the tasks.

For *Study II*, although we selected, as participants, both students and industrial developers, it is worthwhile to replicate the study with a larger number of participants. Furthermore, PROMPTER was only evaluated with two tasks that, although different, are not representative enough for tasks that developers would perform. We believe that *Study I* achieves a better *external validity* whereas *Study II* a better *construct validity*. Finally, concerning the *Study I replication*, it is possible that the different ranking and evaluation obtained for the recommendations pertinent to the 29 tasks depend on these particular cases. In other words, there might be tasks—*e.g.,* related to emerging technology—for which recommendations can be more "volatile", while other tasks—*e.g.,* related to the usage of consolidated programming practices—-such as Java SDK—can be relatively more stable. Therefore, further studies can be needed to confirm or contradict the results obtained in this study.

7 Related Work

PROMPTER is a recommender system that mixes different software engineering fields, namely code search and recommender systems. In this section we go through the research relating to those fields and to PROMPTER.

Semantic Code Search and Code Search Engines The main usage of such search engines is to retrieve code samples and reusable open source code from the Web. Different researches (Bajracharya et al. 2006; Reiss 2009; Thummalapenta 2007; Thummalapenta and Xie 2007) tackled this problem and provided the developers with the capability of searching, ranking and adapting open source code. The mining of open source repositories has also been used to identify API and framework usage and to find highly relevant applications to be reused (McMillan et al. 2012; McMillan et al. 2011; Thummalapenta

and Xie 2008). Other studies analyzed the usage and the habits of the developers in performing researches with code search engines (*e.g.,* Koders) (Bajracharya and Lopes 2009; 2012; Linstead et al. 2007; Umarji et al. 2008), and how general purpose search engines (*e.g.,* Google) outperform code search engines when retrieving code samples from the Web (Sim et al. 2011). In our work we follow an approach based on general-purpose search engines. Differently from the work done so far on code search, we do not target open source repositories to provide code samples and reusable code, or to understand the usage of APIs; instead, we target the crowd knowledge provided by the discussions in Stack Overflow as alternative source. This is because we want to provide developers with code examples with explanations, rather than just with reusable code components/snippets.

Recommender Systems Different typologies of recommender systems to recovery traceability links, suggest relevant project artifacts, and suggest relevant code elements in a project has been presented. Well-known examples are HIPIKAT (Cubranic and Murphy 2003), DEEPINTELLISENCE (Holmes and Begel 2008), and EROSE (Zimmermann et al. 2004). Other work focused on suggesting relevant documents, discussions and code samples from the web to fill the gap between the IDE and the Web browser. Examples are CODETRAIL (Goldman and Miller 2009), MICA (Stylos and Myers 2006), FISHTAIL (Sawadsky and Murphy 2011), and DORA (Kononenko et al. 2012). Subramanian et al. (2014) presented an approach to link different webpages of different nature (*e.g.,* javadoc, source code, Stack Overflow) by harnessing code identifiers. They recommend related webpages augment webpages by injecting code that modifies the original web page.

Among the various sources available on the Web, Q&A Websites and in particular Stack Overflow, have been the target of many recommender systems. Other tools used Stack Overflow as main source for recommendation systems to suggest, within the IDE, code samples and discussions to the developer (Cordeiro et al. 2012; Rigby and Robillard 2013; Takuya and Masuhara 2011). In our previous work we presented SEAHAWK (Ponzanelli et al. 2013a; 2013b), a prototype tool to link Stack Overflow discussions to the source code in the IDE. The work presented here differs in several important ways, as our initial work did not rely on a ranking model, did not feature confidence-based push notifications, but merely tried to establish links between discussions and source code based on information retrieval techniques (*tf-idf*).

The way PROMPTER interacts with search engines to retrieve discussions concerns code context analysis and matching. Recommender systems have to identify relevant code elements to be used as source of information for the current context. Kersten and Murphy (2006) presented MYLYN, a plugin for the Eclipse IDE that identifies relevant code elements (*e.g.,* classes) in a project for a given task. MYLYN tracks events (*e.g.,* edits, commands, selection) and use them to compute the Degree Of Interest (DOI) of a code element for a specific task. The DOI is used, for example, to filter and reduce the number of code elements shown in the package explorer. PROMPTER focuses on the element under current modification, thus simply requiring edit events tracking.

The automation and generation of queries from code is another aspect that relates with PROMPTER. Holmes et al. (2005); Holmes et al. (2006); Reid and Murphy (2005) presented STRATHCONA, a tool to recommend relevant code fragments that automatically extracts queries from structural context of the code. STRATHCONA uses different structural heuristics (*e.g.,* target calls, inheritance, type usage) to match a code sample that is

then presented to the developer in the IDE. In out work we take a wider approach. We consider either the structural aspects (*i.e.,* API Methods), textual aspects (*e.g.,* textual similarity), and the crowd-related aspects (*e.g.,* user reputation) to evaluate the relevance of Stack Overflow discussions.

Mandelin et al. (2005) devised their own query language for code fragments, introduced the concept of *jungloid*, and presented PROSPECTOR, a plugin for the Eclipse IDE that automatically infers queries from the code context, by mining signature graphs generated from API specifications and jungloid graphs representing code. In our work, we focus on the current element being modified by the developer. We take advantage of our own definition of context, and we apply an entropy-based approach to generate the query.

Concerning automation, a remarkable example is the *Microsoft Office assistant* CLIPPY. At its foundation in the *Lumière* project (Horvitz et al. 1998), Horvitz *et al.* developed a bayesian user model to infer user's needs and goal, to estimate user profiles and understand when the assistant should intervene. Similarly to PROMPTER, users can modify the frequency of intervention of the virtual assistant by moving a sensitivity bar. In our work, we focus on similarity between the code entity at hand and Stack Overflow discussions, where the bar sets a threshold on the minimum similarity between the discussion and the code entity needed to push the discussion in the IDE.

Overall, we argue PROMPTER is different from the concept of recommender system proposed so far. PROMPTER is able to automatically and silently retrieve and rank Stack Overflow discussions relevant for the current code context. Then, it uses a configurable "self-confidence" mechanism to push suggestions, yet providing the developer with the possibility of consulting further relevant discussions whenever needed.

8 Conclusion and Future Work

We have presented a novel approach to turn an Integrated Development Environment (IDE) into the developer's programming prompter. The approach is based on (1) automatically capturing the code context in the IDE, (2) retrieving documents from Stack Overflow, (3) ranking the discussions according to a novel ranking model, and (4) suggesting them to the developer when (and only if) it has enough self-confidence. We implemented our approach in PROMPTER, a tool embodying the ideal behavior a recommender should have: a silent observer of the developer, that only intervenes when it deems itself to have a relevant enough suggestion, and that does not force the developer to invoke it but is always available in case the developer needs it. Through a quantitative study (*Study I*), performed via an online survey, we showed how the PROMPTER ranking model resulted to be effective in identifying the right discussions given a code snippet to analyze.

In a second study (*Study II*) we evaluated PROMPTER during maintenance and development tasks. We showed how, from a quantitative point of view, PROMPTER revealed to significantly help developers in completing the experiment tasks and how, from a qualitative point of view, the developer highly appreciated its features and usability.

We also replicated *Study I* after one year from the original experiment. Surprisingly, the results showed that starting from the same code snippets PROMPTER's recommendations changed in 78 % of cases due to the volatility of the information it mines from the web. Despite this, the new recommendations still showed to be related to the task at hand in most of cases. However, the results of the replication clearly highlighted that recommenders built

on top of information mined from the web may experience strong changes in their behavior during time. As a consequence, the replication of empirical studies aimed at evaluating such tools and techniques could be unfeasible.

Our future research agenda focus on performing further evaluations, especially in the context of long-term usage of PROMPTER. Also, we plan to (i) better assess the influence of the "time factor" on the PROMPTER's recommendations (*i.e.,* what is the percentage of recommendations that change after one, three, and six months?), and (ii) study actions to limit the impact of the information volatility on PROMPTER's recommendations (*e.g.,* by increasing the number of exploited search engines).

Acknowledgements Luca Ponzanelli and Michele Lanza thank the Swiss National Science foundation for the financial support through SNF Project "ESSENTIALS", No. 153129.

References

Anvik J, Hiew L, Murphy G (2006) Who should fix this bug? In: Proceedings of ICSE 2006, 361–370. ACM

Bacchelli A, dal Sasso T, D'Ambros M, Lanza M (2012) Content classification of development emails. In: Proceedings of ICSE 2012, 375–385

Baeza-Yates R, Ribeiro-Neto B (1999) Modern information retrieval. Addison-Wesley

Bajracharya S, Lopes C (2009) Mining search topics from a code search engine usage log. In: Proceedings of MSR 2009, 111–120

Bajracharya S, Lopes C (2012) Analyzing and mining a code search engine usage log. Empir Softw Eng 17(4-5):424–466

Bajracharya S, Ngo T, Linstead E, Rigor P, Dou Y, Baldi P, Lopes C (2006) Sourcerer: A search engine for open source code supporting structure-based search. In: Proceedings of OOPSLA 2006, 25–26

Baker RD (1995) Modern permutation test software. In: Randomization Tests. Marcel Decker

Constantine L (1995) Constantine on Peopleware. Yourdon

Cordeiro J, Antunes B, Gomes P (2012) Context-based recommendation to support problem solving in software development. In: Proceedings of RSSE 2012, 85–89. IEEE Press

Cubranic D, Murphy G (2003) Hipikat: recommending pertinent software development artifacts. In: Proceedings of ICSE 2003, 408–418. IEEE Press

Goldman M, Miller R (2009) Codetrail: Connecting source code and web resources. Journal of Visual Languages & Computing

Grissom RJ, Kim JJ (2005) Effect sizes for research: A broad practical approach. Lawrence Associates

Haiduc S, Bavota G, Marcus A, Oliveto R, De Lucia A, Menzies T (2013) Automatic query reformulations for text retrieval in software engineering. In: 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013, 842–851. http://dl.acm.org/citation.cfm?id=2486898

Haiduc S, Bavota G, Oliveto R, De Lucia A, Marcus A (2012) Automatic query performance assessment during the retrieval of software artifacts. In: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, 90–99, DOI doi:10.1145/2351676.2351690, (to appear in print)

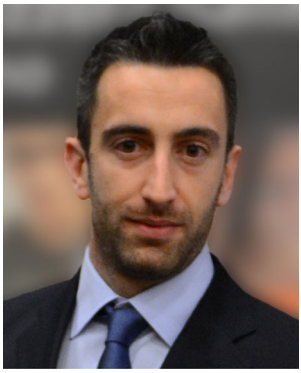
Haiduc S, Bavota G, Oliveto R, Marcus A, De Lucia A (2012) Evaluating the specificity of text retrieval queries to support software engineering tasks. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, 1273–1276, DOI doi:10.1109/ICSE.2012.6227101, (to appear in print)

Hassan AE (2009) Predicting faults using the complexity of code changes. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, 78–88, DOI doi:10.1109/ICSE.2009.5070510, (to appear in print)

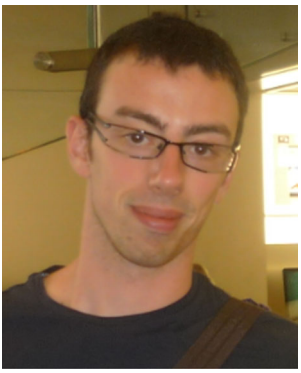
Hintze JL, Nelson RD (1998) Violin plots: A box plot-density trace synergism. Am Stat 52 (2):181–184

Holm S (1979) A simple sequentially rejective Bonferroni test procedure. Scand J Stat 6:65–70
Holmes R, Begel A (2008) Deep intellisense: a tool for rehydrating evaporated information. In: Proceedings of MSR 2008, 23–26. ACM
Holmes R, Walker R, Murphy G (2005) Strathcona example recommendation tool. SIGSOFT Software Engineering Notes 30:237–240
Holmes R, Walker R, Murphy G (2006) Approximate structural context matching: An approach to recommend relevant examples. IEEE TSE 32(12):952–970
Horvitz E, Breese J, Heckerman D, Hovel D, Rommelse K (1998) The lumière project: Bayesian user modeling for inferring the goals and needs of software users. In: Proceedings of UAI 1998 (14th Conference on Uncertainty in Artificial Intelligence), 256–265. Morgan Kaufmann Publishers Inc
Kersten M, Murphy G (2006) Using task context to improve programmer productivity. In: Proceedings of FSE-14, 1–11. ACM Press
Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: Proceedings of ICSE 2007, 344–353. IEEE CS Press
Kononenko O, Dietrich D, Sharma R, Holmes R (2012) Automatically locating relevant programming help online. In: Proceedings of VL/HCC 2012, 127–134
LaToza TD, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: Proceedings of ICSE 2006, 492–501. ACM
Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. Cybern Control Theory 10:707–710
Linstead E, Rigor P, Bajracharya S, Lopes C, Baldi P (2007) Mining internet-scale software repositories. In: In Proceedings of NIPS 2007. MIT Press
Lohar S, Amornborvornwong S, Zisman A, Cleland-Huang J (2013) Improving trace accuracy through data-driven configuration and composition of tracing features. In: Proceedings of ESEC/FSE 2013, 378–388. ACM
Mamykina L, Manoim B, Mittal M, Hripcsak G, Hartmann B Design lessons from the fastest q&a site in the west. In: Proceedings of CHI 2011, 2857–2866. ACM
Mandelin D, Xu L, Bodík R, Kimelman D (2005) Jungloid mining: Helping to navigate the api jungle. In: Proceedings of PLDI 2005 (16th ACM SIGPLAN Conference on Programming Language Design and Implementation), 48–61. ACM
Manning C, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press
McMillan C, Grechanik M, Poshyvanyk D, Fu C, Xie Q (2012) A source code search engine for finding highly relevant applications. IEEE TSE 38(5):1069–1087
McMillan C, Grechanik M, Poshyvanyk D, Xie Q, Fu C (2011) Portfolio: finding relevant functions and their usage. In: Proceedings of ICSE 2011, 111–120. ACM
Oppenheim AN (1992) Questionnaire design, interviewing and attitude measurement. Pinter, London
Panichella A, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A (2013) How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of ICSE 2013, 522–531. ACM/IEEE
Ponzanelli L, Bacchelli A, Lanza M (2013) Leveraging crowd knowledge for software comprehension and development. In: Proceedings of CSMR 2013, 59–66
Ponzanelli L, Bacchelli A, Lanza M (2013) Seahawk: Stack overflow in the ide. In: Proceedings of ICSE 2013, Tool Demo Track, 1295–1298. IEEE
Core Team R (2012) R: a language and environment for statistical computing. Vienna, Austria. http://www.R-project.org. ISBN 3-900051-07-0
Reid RH, Murphy GC (2005) Using structural context to recommend source code examples. In: Proceedings of ICSE 2005, 117–125. ACM
Reiss S (2009) Semantics-based code search. In: Proceedings of ICSE 2009, 243–253. IEEE
Rigby P, Robillard M (2013) Discovering essential code elements in informal documentation. In: Proceedings of ICSE 2013, 832–841
Robertson S (2004) Understanding inverse document frequency: On theoretical arguments for IDF. J Doc 60:2004
Robillard M, Walker R, Zimmermann T (2010) Recommendation systems for software engineering. IEEE Software

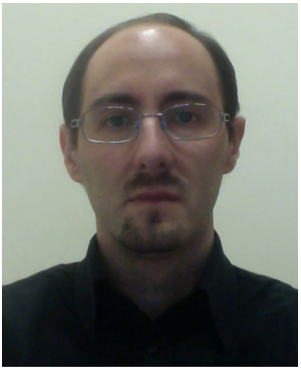
Sawadsky N, Murphy G (2011) Fishtail: from task context to source code examples. In: Proceedings of TOPI 2011, 48–51. ACM
Shannon CE (1948) A mathematical theory of communication. Bell Syst Tech J 27:379–423. 625–56
Sheskin DJ (2007) Handbook of parametric and nonparametric statistical procedures (fourth edition). Chapman & All
Sim S, Umarji M, Ratanotayanon S, Lopes C (2011) How well do search engines support code retrieval on the web ACM TOSEM:1–25
Stylos J, Myers BA (2006) Mica: A web-search tool for finding api components and examples. In: Proceedings of VL/HCC 2006, 195–202
Subramanian S, Inozemtseva L, Holmes R (2014) Live api documentation. In: Proceedings of ICSE 2014 (36th International Conference on Software Engineering), ICSE 2014, 643–652. ACM
Takuya W, Masuhara H (2011) A spontaneous code recommendation tool based on associative search. In: Proceedings of SUITE 2011, pp. 17–20. ACM
Thummalapenta S (2007) Exploiting code search engines to improve programmer productivity. In: Proceedings of OOPSLA 2007, 921–922. ACM
Thummalapenta S, Xie T (2007) Parseweb: a programmer assistant for reusing open source code on the web. In: Proceedings of ASE 2007, 204–213. ACM
Thummalapenta S, Xie T (2008) Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In: Proceedings of ASE 2008, 327–336. IEEE
Umarji M, Sim S, Lopes C (2008) Archetypal internet-scale source code searching. In: Proceedings of OSS 2008, 257–263
Vassallo C, Panichella S, Di Penta M, Canfora G (2014) Codes: mining source code descriptions from developers discussions. In: 22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014, 106–109
Wang T, Harman M, Jia Y, Krinke J (2013) Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of ESEC/FSE 2013, 455–465. ACM
Wettel R, Marinescu R (2005) Archeology of code duplication: recovering duplication chains from small duplication fragments. In: Proceedings of SYNASC 2005, 63–70
Williams L (2001) Integrating pair programming into a software development process. In: Proceedings of CSEET 2001, 27–36. IEEE
Zimmermann T, Weißgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings of ICSE 2004, 563–572. IEEE



Luca Ponzanelli is a PhD student at the University of Lugano, Switzerland since September 2012. He is currently working in the REVEAL research group under the supervision of Prof. Dr. Michele Lanza. He received his master's degree from the University of Lugano, in 2012, and he received his bachelor's degree from the University of Milano-Bicocca in 2010. His research interests include mining software repositories, software maintenance, and recommender systems for software engineering. He serves as web chair for MSR'16 and SANER'16.



Gabriele Bavota is Assistant Professor at the Free University of Bozen-Bolzano, Italy. He received the PhD degree in computer science from the University of Salerno, Italy, in 2013. From January 2013 to October 2014 he has been research fellow at the University of Sannio, Italy. His research interests include software maintenance, empirical software engineering, mining software repository, refactoring of software systems, and information retrieval. He is author of over 60 papers appeared in international journals, conferences and workshops. He serves as a Program Co-Chair for ICPC'16 and SCAM'16. He also serves and has served as organizing and program committee member of international conferences in the field of software engineering, such as ICSME, MSR, ICPC, SANER, SCAM, and others. He is a member of IEEE Computer Society and ACM.



Massimiliano Di Penta is associate professor at the University of Sannio, Italy since December 2011. Between Before that, he was assistant professor in the same University since December 2004. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is author of over 200 papers appeared in international journals, conferences and workshops. He serves and has served in the organizing and program committees of over 100 conferences such as ICSE, FSE, ASE, ICSM, ICPC, GECCO, MSR WCRE, and others. He has been general co-chair of various events, including the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010), the 2nd International Symposium on Search-Based Software Engineering (SSBSE 2010), and the 15th Working Conference on Reverse Engineering (WCRE 2008). Also, he has been program chair of events such as the 28th IEEE International Conference on Software Maintenance (ICSM 2012), the 21st IEEE International Conference on Program Comprehension (ICPC 2013), the 9th and 10th Working Conference on Mining Software Repository (MSR 2013 and 2012), the 13th and 14th Working Conference on Reverse Engineering (WCRE 2006 and 2007), the 1st International Symposium on Search-Based Software Engineering (SSBSE 2009), and other workshops.

He is currently member of the steering committee of ICSME, MSR, SSBSE, and PROMISE. Previously, he has been steering committee member of other conferences, including ICPC, SCAM, and WCRE. He is in the editorial board of IEEE Transactions on Software Engineering, the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley.

Rocco Oliveto is an Associate Professor at University of Molise (Italy), where he is also the Chair of the Computer Science program and the Director of the Laboratory of Computer Science and Scientific Computation (CSSC Lab).

He co-authored over 100 papers on topics related to software traceability, software maintenance and evolution, search-based software engineering, and empirical software engineering. His activities span various international software engineering research communities. He has served as organizing and program committee member of several international conferences in the field of software engineering. He was program co-chair of ICPC 2015, TEFSE 2015 and 2009, SCAM 2014, WCRE 2013 and 2012. He was also keynote speaker at MUD 2012.

Michele Lanza is a full professor at the faculty of informatics of the University of Lugano, where he founded the REVEAL research group in 2004.

He co-authored over 150 journal and conference publications and the book Object-Oriented Metrics in Practice.

His activities span various international software engineering research communities. He has served on the program committees of ICSE, FSE, ICSME, ICPC, MSR and many other conferences, and as program co-chair of ICSM 2010, VISSOFT 2009, MSR 2008, IWPSE 2007, and MSR 2007. He was keynote speaker at MSR 2010, CBSOFT/SBES 2011, BENEVOL 2011, and CSMR 2013. He is a board member of CHOOSE (Swiss Group for Object-Oriented Systems and Environments), and vice-president of the Moose association. He is a steering committee member of VISSOFT (International Workshop on Visualizing Software for Understanding and Analysis), and of the ERCIM working group on software evolution.