Employing Source Code Information to Improve Question-Answering in Stack Overflow

Themistoklis Diamantopoulos, Andreas L. Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki Greece
Email: thdiaman@issel.ee.auth.gr, asymeon@eng.auth.gr

***Abstract*—Nowadays, software development has been greatly influenced by question-answering communities, such as Stack Overflow. A new problem-solving paradigm has emerged, as developers post problems they encounter that are then answered by the community. In this paper, we propose a methodology that allows searching for solutions in Stack Overflow, using the main elements of a question post, including not only its title, tags, and body, but also its source code snippets. We describe a similarity scheme for these elements and demonstrate how structural information can be extracted from source code snippets and compared to further improve the retrieval of questions. The results of our evaluation indicate that our methodology is effective on recommending similar question posts allowing community members to search without fully forming a question.**

***Index Terms*—Source Code Mining, Indexing, Search Engines**

I. Introduction

Lately, the process of developing software has been greatly influenced by the emergence of developer-centric question-answering (QA) communities, such as Stack Overflow. Nowadays, there are several scenarios where writing source code to solve an issue is preceded by searching in such a QA website in order to probe on how other users have handled a specific problem, or whether they have already found a solution. In the case of Stack Overflow, question posts may span from simple standard (or third-party) library issues (e.g. "How to convert a string to an integer?") to more complex scenarios (e.g. "How to place an image on a swing button?" or "Why iterating this list throws a ConcurrentModificationException?").

Submitting a new question post requires several steps; the developer has to form a question with a clear title, further explain the problem in the question body, isolate any relevant source code fragments, and possibly assign tags to the question (e.g. "swing" or "android") to draw the attention of community members that are familiar with the specific technologies/frameworks. In most cases, however, the first step before forming a question post is to find out whether the same question (or a similar one) is already posted. To do so, one can initially perform a search on question titles. Further improving the results requires including tags or question bodies. Probably the most effective query includes the title, the tags, and the body, i.e. the developer has to form a complete question post. However, this process may be difficult or time-consuming; for example the source code of the problem may not be easily isolated, or the developer may not know which tags to use.

In this paper, we explore the problem of finding similar question posts in Stack Overflow, given different types of information. Apart from titles, tags, or question bodies, we explore whether a developer could be able to search for similar questions using source code fragments, thus not requiring to fully form a question. Our main contribution is a question similarity scheme which can be used to recommend similar question posts to users of the community trying to find a solution to their problem. Since this scheme could be useful for identifying similar questions, it may also prove useful to community members or moderators so that they identify linked or duplicate questions, or even so that they try to find similar questions to answer and thus contribute to the community.

The rest of this paper is organized as follows. Section II describes how the Stack Overflow data is collected and preprocessed to allow extracting useful information. Our methodology for creating a question similarity scheme is presented in Section III, while our approach is evaluated in Section IV. Finally, Section V concludes this paper, by summarizing the main contributions and providing insight for further research.

II. Data Collection and Preprocessing

Our methodology is applied on the latest official data dump of Stack Overflow as of September 26, 2014, provided by [1]. We have used Elasticsearch [2] to store the data for our analysis, since it provides indexing and supports the execution of fast queries on large chunks of data. Storage in Elasticsearch is simple; any record of data is a *document*, documents are stored inside *collections*, and collections are stored in an *index*. We defined a *posts* collection, which contains the Java questions of Stack Overflow (determined by the "java" tag)[1].

A. Extracting Data from Questions

An example Stack Overflow question post is depicted in Figure 1. The main elements of the post are: the title (shown at the top of Figure 1), the body, and the tags (at the bottom of Figure 1). These three elements are stored and indexed for each question post in our Elasticsearch server to allow for retrieving the most relevant questions. The title and the body are stored as text, while the tags are stored as a list of keywords. Since the body is in html, we have also employed an html parser to extract the source code snippets for each question.

[1]Although our methodology is mostly language-agnostic, we have focused on Java Stack Overflow questions to demonstrate our proof of concept.

How to iterate through all buttons of grid layout?

I have a 2x2 grid of buttons

```
JFrame frame = new JFrame("myframe");
JPanel panel = new JPanel();
Container pane = frame.getContentPane();
GridLayout layout = new GridLayout(2,2);
panel.setLayout(layout);
panel.add(upperLeft);
panel.add(upperRight);
panel.add(lowerLeft);
panel.add(lowerRight);
pane.add(panel);
```

where upperLeft, upperRight, etc. are buttons. How can I iterate through all of the buttons?

java swing jbutton

Fig. 1. Example Stack Overflow question post

B. Storing and Indexing Data

This subsection describes how the elements of each post are indexed in Elasticsearch. Note that Elasticsearch not only stores the data but also creates and stores *analyzed* data for each field, so that the index can be searched efficiently.

The title of each question post is stored using the *standard analyzer* of Elasticsearch. This analyzer comprises a tokenizer that splits the text into tokens (i.e. lowercase words) and a token filter that removes certain common stopwords (in our case the common English stopwords). The tags are stored in an array field without further analysis, since they are already split and in lowercase. The body is analyzed using an *html analyzer*, which is similar to the standard analyzer, however it first removes the html tags using the html_strip character filter of Elasticsearch. Titles, tags, and question bodies are stored in the fields "Title", "Tags", and "Body" respectively.

Concerning the snippets extracted from the question bodies, we had to find a representation that describes not only their terms (which are already included in the "Body" field) but also their structure. Source code representation is an interesting problem for which several solutions can be proposed. Although complex representations, such as the snippet's abstract syntax tree (AST) or its program dependency graph, include a lot of information, they cannot be used in the case of incomplete snippets such as the one in Figure 1. For example, extracting structural information from the AST and using matching heuristics between classes and methods, as in [3], is not possible since we may not have information about classes and methods (e.g inheritance, declared variables, etc.).

Interesting representations can be traced in the API discovery problem, where several researchers have used sequences to represent the snippets [4], [5]. However, these approaches concern how to call an API function, thus they are not applicable for the broader problem of snippet similarity. Other interesting approaches include mining types from the snippet [6], [7] or tokenizing the code and treating it as a bag of words, perhaps also considering the entropy of any given term [8]. In any case, though, these representations result in loss of structural information. Thus, we had to find a representation that uses both the types and the structure of the snippets.

The representation we propose is a simple sequence, which is generated by three source code instruction types: assignments (`AM`), functions calls (`FC`), and class instantiations (`CI`). We deviate from complex representations, as in [9], since incomplete snippets may not contain all the declarations. For example, in the snippet of Figure 1, we do not really know anything about the "upperLeft" object; it could be a field or a class. Our method operates on the AST of each snippet, extracted using the Eclipse compiler. Upon parsing the AST and extracting all individual commands, we take two passes over them (which are adequate, since the code is sequential). In the first pass, we extract all the declarations from the source code (including classes, fields, methods, and variables) and create a lookup table. The lookup table for the snippet of Figure 1 is shown in Table I.

TABLE I
EXAMPLE LOOKUP TABLE FOR THE SNIPPET OF FIGURE 1

Variable	Type
frame	JFrame
panel	JPanel
pane	Container
layout	GridLayout

Upon extracting the types, the second pass creates a sequence of commands for the snippet. For example, the command "pane = frame.getContentPane()" provides an item `FC_getContentPane`. After that, we further refine this sequence item by substituting the name of the variable or function (in this case `getContentPane`) by its type (in this case `Container`) using also the lookup table when required. If no type can be determined, the type `void` is assigned. The sequence for the snippet of Figure 1 is shown in Figure 2.

```
CI_JFrame, CI_JPanel, FC_Container,
CI_GridLayout, FC_void, FC_void,
FC_void, FC_void, FC_void, FC_void
```

Fig. 2. Example sequence for the snippet of Figure 1

This representation is also added to our index in the "Snippets" field, as an ordered list.

III. METHODOLOGY

This section discusses our methodology, illustrating how text, tags, and snippets can be used to find similar questions. A query on the index may contain one or more of the "Title", "Tags, "Body", and "Snippets" fields. When searching for a question post, we calculate a score that is the average among the scores for each of the provided fields. The score for each field is normalized in the $[0, 1]$ range, so that the significance of all fields is equal. The following subsections illustrate how the scores are computed for each field.

A. Text Matching

The similarity between two segments of text, either titles or bodies, is computed using the *term frequency-inverse document frequency (tf-idf)*. Upon splitting the text/document into tokens/terms (which is already handled by Elasticsearch), we use the vector space model, where each term is the value of a dimension of the model. The frequency of each term in each document (tf) is computed as the square root of the number of times the term appears in the document, while the inverse document frequency (idf) is the logarithm of the total number of documents divided by the number of documents that contain the term. The final score between two documents is computed using the cosine similarity between them:

$$score(d_1, d_2) = \frac{d_1 \cdot d_2}{|d_1| \cdot |d_2|} = \frac{\sum_1^N w_{t_i,d_1} \cdot w_{t_i,d_2}}{\sum_1^N w_{t_i,d_1}^2 \cdot \sum_1^N w_{t_i,d_2}^2} \quad (1)$$

where d_1, d_2 are the two documents, and w_{t_i,d_j} is the tf-idf score of term t_i in the document d_j.

B. Tag Matching

Since the values of "Tags" are unique, we can compute the similarity between "Tags" values by handling the lists as sets. Thus, we define the similarity as the *Jaccard index* between the sets. Given two sets, T_1 and T_2, their Jaccard index is the size of their intersection divided by the size of their union:

$$J(T_1, T_2) = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} \quad (2)$$

Finally, note that we exclude the tag "java" since it is present in all sets, and so it is not useful for the scoring.

C. Snippet Matching

Upon having extracted sequences from snippets (see subsection II-B), we define a similarity metric between two sequences based on their *Longest Common Subsequence (LCS)*. Given two sequences S_1 and S_2, their LCS is defined as the longest subsequence that is common to both sequences. Note that a subsequence is not required to contain consecutive elements of any sequence. For example, given two sequences $S_1 = [A, B, D, C, B, D, C]$ and $S_2 = [B, D, C, A, B, C]$, their LCS is $LCS(S_1, S_2) = [B, D, C, B, C]$. The LCS problem for two sequences S_1 and S_2 can be solved using dynamic programming. The computational complexity of the solution is $O(m \times n)$, where m and n are the lengths of the two sequences [10], so the algorithm is quite fast.

Upon having found the LCS between two snippet sequences S_1 and S_2, the final score for the similarity between them is computed by the following equation:

$$score(S_1, S_2) = 2 \cdot \frac{|LCS(S_1, S_2)|}{|S_1| + |S_2|} \quad (3)$$

Since the length of the LCS is always smaller than the length of the smallest sequence, the fraction of it divided by the sum of sequence lengths is in the range $[0, 0.5]$. Hence. the score of the above equation lies in the range $[0, 1]$.

For example, if we calculate the score between the sequence of Figure 2 and that of Figure 3, the length of the LCS is 5, and the lengths of the two sequences are 10 and 6. Thus, the score between the two sequences, using equation (3), is 0.625.

```
CI_JFrame, FC_Container, AM_float,
CI_GridLayout, FC_void, FC_void
```

Fig. 3. Example sequence extracted from a source code snippet

IV. Evaluation

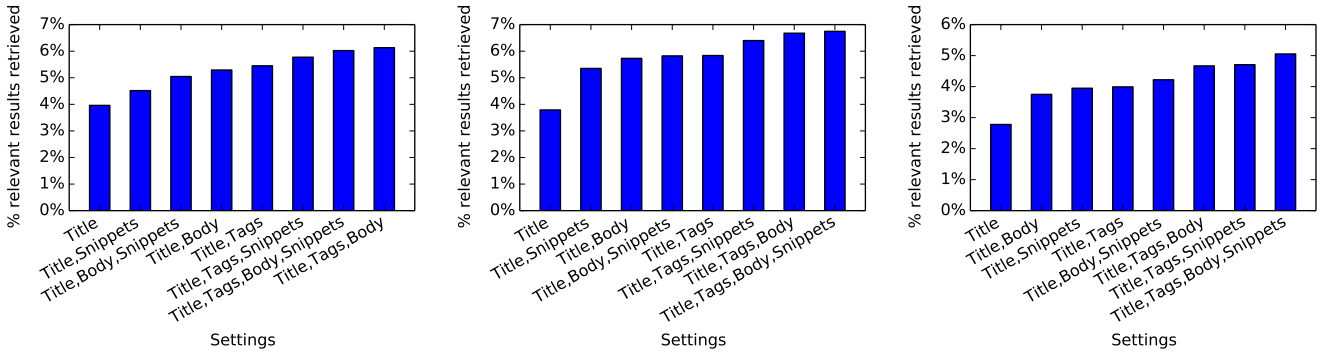
Evaluating the similarity of question posts in Stack Overflow is difficult, since the data are not annotated for this type of tests. We use the presence of a link between two questions (also given by [1]) as an indicator of whether the questions are similar. Although this assumption seems rational, the opposite assumption, i.e. that any non-linked questions are not similar, is not necessarily true. In fact, the Stack Overflow data dump has approximately 700000 java question posts, out of which roughly 300000 have snippets that result in sequences (i.e. snippets with at least one assignment, function call, or class instantiation), and on average each of these 300000 posts has 0.98 links. In our scenario, however, the problem is formed as a search problem, so the main issue is to detect whether the linked (and thus similar) questions can indeed be retrieved.

In our evaluation, we have included only questions with at least 5 links (including duplicates which are also links according to the Stack Overflow data schema)[2]. For performance reasons, all queries are at first performed on the question title, and then the first 1000 results are searched again using both title and any other fields, all in an Elasticsearch *bool* query. We used 8 settings (given the title is always provided, the settings are all the combinations of tags, body, and snippets) to evaluate our methodology. For each question we keep the first 20 results, assuming this is the maximum a developer would examine (using other values had similar results).

We summarize the results of our analysis in the diagrams of Figure 4. The diagram of Figure 4a depicts the average percentage of relevant results (compared to the total number of relevant links for each question) in the first 20 results of a query, regardless of the ranking. These results involve all questions of the dataset that have source code snippets (approximately 300000), even if the sequence extracted from them has length equal to 1 (having sequence length equal to 0 means no comparison can be made). Note that although the results for all settings in Figure 4a are below 10%, this is actually a shortcoming of the dataset. Many more of the retrieved results may be relevant, however they are not linked.

As shown in Figure 4a, when snippets are used in accordance with titles and/or tags, the retrieved questions are more relevant. However, when the body of a post is used, the use of

[2]Our methodology supports all questions, regardless of the number of links. However, in the context of our evaluation, we may assume that questions with fewer links may be too localized and/or may not have similar question posts.



(a) (b) (c)

Fig. 4. Percentage of relevant results (compared to the number of links of each question) in the first 20 results of each query. The tested settings are combinations of title, tags, body, and snippets and they are evaluated for questions with snippet sequence length larger than or equal to (a) 1, (b) 3, and (c) 5.

snippets is probably redundant or even non-preferable. This is expected since the diagram of Figure 4a involves many small snippets that result in sequences that are hard to compare. In these cases, supplying the mechanism with more (or larger) snippets, or even adding some text to the body is preferred.

To assert the validity of these claims and further discuss the effectiveness of our approach, we performed two more tests (results depicted in Figures 4b and 4c). As in Figure 4a, the diagrams depict the percentage of relevant results in the first 20 results of a query, however for snippets with sequence length larger than or equal to 3 and 5 (approximately 200000 and 150000 question posts respectively), for Figures 4b and 4c respectively. The effectiveness of using code snippets to find similar questions is quite clear from these two diagrams. As expected, using all available sources of information, i.e. question title, tags, body, and snippets, is optimal in both cases. Our snippet matching approach seems to be quite effective in retrieving relevant questions, even when the question body is not used. Furthermore, when the provided snippets are larger and thus result in longer sequences, as in Figure 4c, using them is more effective than using the whole question body.

Another decisive piece of information that seems to affect the relevance of the results in all three evaluation scenarios shown in Figure 4 is the use of tags. We notice that using tags and snippets is almost as effective as writing the whole question body. Thus, the developer could write a title and some tags for his/her question and then post a relevant piece of his/her code without requiring to fully formulate a question.

V. Conclusion

In this paper, we have explored the problem of recommending similar questions in Stack Overflow. We have developed a similarity scheme which exploits not only the title, tags, and body of a question, but also its source code snippets. The results of our evaluation indicate that snippets are a valuable source of information for detecting links or for determining whether a question is already posted. Additionally, our similarity scheme allows performing effective queries on Stack Overflow without requiring fully formulated questions.

Although snippet matching is a difficult task, we believe that our methodology is a step towards the right direction. Future research includes further adapting the methodology to the special characteristics of different snippets, e.g. by clustering them according to their size or their complexity. Additionally, different structural representations for source code snippets may be tested in order to improve on the similarity scheme.

Acknowledgment

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission.

References

[1] Annie T. T. Ying. Mining Challenge 2015: Comparing and combining different information sources on the Stack Overflow data set. In *The 12th Working Conference on Mining Software Repositories*, page to appear, 2015.

[2] Elasticsearch: Open source distributed real time search & analytics. http://www.elasticsearch.org/. [retrieved February, 2015].

[3] R. Holmes and G.C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE 2005, pages 117–125, St. Louis, MO, USA, May 2005.

[4] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[5] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: Mining for sample code. *SIGPLAN Not.*, 41(10):413–430, October 2006.

[6] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the 2013 10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 85–88, May 2013.

[7] Siddharth Subramanian, Laura Inozemtseva, and Reid Holmes. Live API documentation. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 643–652, New York, NY, USA, 2014. ACM.

[8] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111, New York, NY, USA, 2014. ACM.

[9] Sheng-Kuei Hsu and Shi-Jen Lin. MACs: Mining API code snippets for code reuse. *Expert Syst. Appl.*, 38(6):7291–7301, June 2011.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*, pages 390–396. The MIT Press, 3rd edition, 2009.