2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Towards Mining Answer Edits to Extract Evolution Patterns in Stack Overflow

Themistoklis Diamantopoulos, Maria-Ioanna Sifaki, and Andreas L. Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
Thessaloniki, Greece
thdiaman@issel.ee.auth.gr, mnsifaki@ece.auth.gr, asymeon@eng.auth.gr

***Abstract*—The current state of practice dictates that in order to solve a problem encountered when building software, developers ask for help in online platforms, such as Stack Overflow. In this context of collaboration, answers to question posts often undergo several edits to provide the best solution to the problem stated. In this work, we explore the potential of mining Stack Overflow answer edits to extract common patterns when answering a post. In particular, we design a similarity scheme that takes into account the text and code of answer edits and cluster edits according to their semantics. Upon applying our methodology, we provide frequent edit patterns and indicate how they could be used to answer future research questions. Assessing our approach indicates that it can be effective for identifying commonly applied edits, thus illustrating the transformation path from the initial answer to the optimal solution.**

***Keywords*-code evolution; code snippets; Stack Overflow**

I. Introduction

Nowadays, developers collaborate in online forums and question-answering communities in order to share their ideas and confront common challenges that may arise when writing source code. Stack Overflow, which is currently one of the most popular communities, hosts at the time of writing this paper 17 million questions from more than 10 million users[1]. These questions may refer to different scenarios, including queries about how to use APIs, exceptions thrown when writing code, questions about the usage of libraries, etc.

When these scenarios are interesting and generic enough for many developers, they are usually answered collaboratively by multiple community members, who strive to provide *reusable answers*. To quote Jeff Atwood, one of the creators of Stack Overflow, the goal of the community is not 'answer my question' but 'let's collaboratively build an artifact that will benefit future coders' [1]. According to the guidelines of Stack Overflow[2], an answer is considered satisfactory if it is well formatted, covers the original question, includes links (when relevant) for those willing to do more research on the topic, and refers to the topic as a whole (including any limitations of the answer). Apart from the above, another crucial guideline is that members are encouraged to post partial answers and of course to edit answers (either third-party or their own) in order to find the optimal solution to the posted problem.

[1]https://stackexchange.com/sites
[2]https://stackoverflow.com/help/how-to-answer

This collaborative paradigm is what makes Stack Overflow so broadly appealing. Answers are viewed as artifacts that are originally created to cover some question criteria and evolve into generic solutions. And these artifacts may have more or less the same strengths or weaknesses of any software artifact; they may be of high or low quality, the surrounding text/comments can be well explanatory or even not exist, the code may include error handling (and/or test for corner cases), etc. These are all problems resolved via answer evolution. Multiple members may view an answer that has potential in solving a specific problem, and edit it to improve it. This process is currently taking place in an expert-based manner as it depends mainly on what we may call the knowledge of the community. One may argue that this knowledge could be harnessed to provide better understanding of how Stack Overflow works and enable semi-automated answer evolution.

In this work, we explore the potential of mining Stack Overflow answers and focus particularly on their evolution over consecutive edits. We design a similarity scheme for answer edits that takes into account the characteristics of the text and the embedded code snippets. Upon using our scheme, we apply clustering in order to extract frequent edit patterns that illustrate how initial answers evolve into optimal solutions and represent best practices for evolving software artifacts.

II. Data Extraction and Preprocessing

Our methodology is applied on the SOTorrent dataset [2], which is a relational database built to analyze the evolution of Stack Overflow posts. SOTorrent provides access to all posts (table *Posts*), along with their history both as a whole (table *PostHistory*) and at the level of individual text or code blocks (table *PostBlockVersion*) [3]. We have focused on Java posts (determined by the "java" tag) as a proof of concept for our methodology, which however is mostly language-agnostic.

We first created a view for all answer edits by joining *Posts*, *PostHistory* and *PostBlockVersion* and selecting only answers (i.e. field *PostTypeId* equal to 2). After that, we extracted the text and code snippets of all answer edits by performing a semi-join between *PostBlockVersion* and our newly created view. We added to the result (using UNION) the ids, texts and code snippets of the original answer posts. The final result set has edits with fields *IdBefore*, *IdAfter*, *Comment*, *TextBefore*, *TextAfter*, *CodeBefore*, *CodeAfter*, where *IdBefore* and *IdAfter* are the ids of the post before and after the edit respectively,

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00043

Version 1

This is how you can read a text file in Java:

```
String content;
try (FileReader reader = new FileReader(file)) {
    char[] chars = new char[(int) file.length()];
    reader.read(chars);
    content = new String(chars);
} catch (IOException e) {
    e.printStackTrace();
}
```

Version 2 - Added comment about the standard library

This is how you can read a text file in Java using only the standard library:

```
String content;
try (FileReader reader = new FileReader(file)) {
    char[] chars = new char[(int) file.length()];
    reader.read(chars);
    content = new String(chars);
} catch (IOException e) {
    e.printStackTrace();
}
```

Version 3 - Put file outside the block

Suppose you are given a file named foo.txt

This is how you can read a text file in Java using only the standard library:

```
String content;
File file = new File("foo.txt");
try {
    FileReader reader = new FileReader(file);
    char[] chars = new char[(int) file.length()];
    reader.read(chars);
    content = new String(chars);
} catch (IOException e) {
    e.printStackTrace();
}
```

Version 4 - Fix to close the file reader

Suppose you are given a file named foo.txt

This is how you can read a text file in Java using only the standard library:

```
String content;
File file = new File("foo.txt");
try {
    FileReader reader = new FileReader(file);
    char[] chars = new char[(int) file.length()];
    reader.read(chars);
    content = new String(chars);
    reader.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

Fig. 1. Example Stack Overflow answer post and subsequent versions (edits are highlighted) for a question about reading a text file

Comment is the comment of the edit, *TextBefore* and *TextAfter* is the text before and after the edit, and *CodeBefore* and *CodeAfter* is the code before and after the edit.

Using the queries, we have generated a set of answer posts and their corresponding versions, which are the results of subsequent edits. An example answer post that shall be used throughout our analysis is shown in Figure 1. Version 1 is originally posted as an answer to a question about reading a text file. Version 2 has a text addition, which clarifies that the advantage of this solution is that it does not require any external library other than the standard library of Java. Versions 3 and 4 improve the code by declaring the (possibly forgotten) file variable and by closing the reader buffer, respectively.

In this example the edits are well documented (comments are shown above the posts) and succeed one another. This is not the norm in the dataset, as there are many cases where no comments are provided by the editor or multiple changes are made in the same edit. However, this example is indicative of what one may expect to extract from the edits. Ideally, we would want to keep track of scenarios such as the ones shown in this Figure, i.e. forgotten dependency definitions, missing declarations, buffers that do not close, and similar.

Our goal is to build a similarity scheme (see next Section) that shall compare edits to one another. Thus, the first step is to create a collection of edits. We represent each edit as a set that contains its comment, the text of the answer before the edit, the difference (additions and deletions) between the text before and after the edit, the code of the answer before the edit, and the difference (additions and deletions) between the code before and after the edit. The differences were computed per line using the *difflib* module of the Python standard library.

III. A Similarity Scheme for Evolving Answers

Upon having created the dataset of edits, we applied a similarity scheme for comparing them, based on text (for comment and answer text) and code (for answer snippets) data.

A. Text Matching

The similarity between two fragments of text, either comments or answer texts, is computed using the *term frequency-inverse document frequency (tf-idf)*. At first, we split text into tokens[3] and then apply stemming and remove any stopwords[4]. After that we employ tf-idf to construct a vector space model, where each term/token is a dimension of the model and each text/comment is a document of the model. The term frequency in each document is computed as the square root of the number of times the term appears in the document, while the inverse document frequency is computed as the logarithm of the total number of documents divided by the number of documents that contain the term. We created two models, one for comments and one for texts[5]. The score between two comments/texts is the cosine similarity between them:

$$tsim(c_1, c_2) = \frac{c_1 \cdot c_2}{|c_1| \cdot |c_2|} = \frac{\sum_1^N w_{t_i,c_1} \cdot w_{t_i,c_2}}{\sum_1^N w_{t_i,c_1}^2 \cdot \sum_1^N w_{t_i,c_2}^2} \quad (1)$$

where c_1, c_2 are the two comments/texts, and w_{t_i,d_j} is the tf-idf score of token t_i in the c_j.

[3]As we focus on Java, we also split CamelCase tokens using the regex `.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)` to ensure that terms such as List and ArrayList are properly linked.

[4]We used the English stopword list of NLTK [4].

[5]For both models, we dropped terms appearing in more than 50% of the documents (as too generic) and terms appearing in less than 10 documents (as too specific). These thresholds were selected after evaluating different settings.

B. Snippet Matching

Matching snippets requires representations that describe both terms and structure. At first, we treat the code as bag-of-words and apply the methodology of the previous subsection to determine edits in a token-based manner. Though useful, this representation is not enough, as it does not capture the order of the code. On the other hand, employing a representation such as the abstract syntax tree (AST) is not optimal in the case of incomplete snippets such as the ones in Figure 1. For instance, it is not possible to employ heuristics between classes and methods, as in [5], as there may not be any information about classes and methods (e.g. inheritance, method declarations, etc.). Other interesting approaches include extracting the snippet types [6], [7] or even the API calls [8], [9]. However, the former are also missing structural information, while the latter are limited to the API discovery problem, and thus do not generalize to snippet similarity regardless of API calls.

Thus, we use as representation a sequence [10] generated by three instruction types: assignments (`AM`), functions calls (`FC`), and class instantiations (`CI`). More complex sequences are also possible, e.g. as in [11], however they are not a good fit for our problem since they lead to very specific patterns. On the other hand, using only these instruction types leads to more abstract representations, and thus we expect to extract more generic patterns. We pass two times over each snippet. During the first pass, we extract code declarations (i.e. classes, fields, methods, and variables), while during the second pass we create a sequence of commands for the snippet. For example, the command 'File file = new File("foo.txt")' provides an item `CI_File`. In the case of function calls, we keep the return type of the call (or `void` if no type can be determined) as types can be very helpful for identifying similar APIs [6], [7]. As an example, the sequence for the snippet of Version 4 of Figure 1 is `[CI_File, CI_FileReader, FC_char, FC_void, CI_String, FC_void, FC_void]`.

We define the similarity between two code sequences using the *Longest Common Subsequence (LCS)*. Given two sequences CS_1 and CS_2, their LCS is the longest subsequence of (not necessarily consecutive) elements common to both sequences. E.g., let CS_1 be the sequence extracted above for Version 4 of the snippet of Figure 1 and CS_2 be the sequence extracted for Version 1 of the snippet (`[CI_FileReader, FC_char, FC_void, CI_String, FC_void]`), the LCS of CS_1 and CS_2 is `[CI_FileReader, FC_char, FC_void, CI_String, FC_void]`. The LCS is computed using dynamic programming, resulting in complexity equal to the product of the lengths of the sequences [12]. Finally, the score between the two snippets is computed as follows:

$$csim(CS_1, CS_2) = 2 \cdot \frac{|LCS(CS_1, CS_2)|}{|CS_1| + |CS_2|} \quad (2)$$

As the length of the LCS of two sequences is always smaller than the length of the smallest sequence, the result of the above equation lies in the range $[0, 1]$. As an example, the score for the snippets of Version 4 and Version 1 of Figure 1 (CS_1 and CS_2, respectively) is $2 \cdot 5/(7+5) = 10/12 = 0.833$.

Finally, given two answer edits X and Y, their similarity is computed as the average of six similarity scores: the scores of their text differences ($tsim(TextAdditions_X, TextAdditions_Y)$, $tsim(TextDeletions_X, TextDeletions_Y)$), the scores of their code computed using the bag-of-words representation ($tsim(CodeAdditions_X, CodeAdditions_Y)$, $tsim(CodeDeletions_X, CodeDeletions_Y)$), and the scores of their snippets computed using sequences ($csim(SequenceAdditions_X, SequenceAdditions_Y)$, $csim(SequenceDeletions_X, SequenceDeletions_Y)$). Note that we do not add comment data in this similarity score, as it will be used as external optimization in the following Section.

IV. Extracting Answer Evolution Patterns

The next step is to use our similarity scheme to create a distance matrix that contains the distance (computed as $1-$similarity) between each pair of answer edits in the dataset. At first, the complexity may seem quite large; given N edits, we need to compute $N(N-1)/2$ values for the upper triangular part of the matrix. However, the problem we are trying to solve is edit pattern recognition, and thus we do not need the full matrix, but rather certain parts of it, i.e. values of posts that contain code (to assess our code mining methodology) and may lead to useful patterns. As a result, we first filtered the edits and kept only those with at least one changed instruction (i.e. $max(SequenceAdditions, SequenceDeletions) \geq 1$). Moreover, for optimization reasons (see next subsection), we dropped any edits that did not have any meaningful comments (e.g. edits with less than 10 characters or edits with default comments produced by Stack Overflow, such as 'Added # characters in body'). This preprocessing produced 20667 edits. The resulting distance matrix would then have more than 200 million values, however we noticed that most of them were too different from each other to form patterns, therefore we kept only the 100 most similar edits for each edit resulting in a more manageable sparse matrix with 822156 non-zero values.

A. Clustering Answer Edits

Although there are several approaches for clustering source code sequences [13]–[16], these approaches focus on the API call extraction problem, and thus they cannot be applied on datasets including both text and snippets. Using our methodology, on the other hand, we obtain a distance matrix, therefore we are able to employ hierarchical clustering techniques.

We applied agglomerative hierarchical clustering with the average (also known as UPGMA [17]) linkage method, which was chosen as it is robust in cases with many outliers. The next challenge that we had to confront was to determine the optimal number of clusters. Ideally, it would be best to have an annotated dataset with answer edits that indeed represent similar functionality; this would be a ground truth against which clustering could be optimized. As, however, these annotations are not available (and handcrafting them would pose threats to validity), we use instead the comments of edits as ground truth, and consider that when two comments are similar, then the corresponding edits are also similar.

TABLE I
SAMPLE CLUSTERS WITH THEIR NUMBER OF POINTS (EDITS) AND THEIR 5 MOST REPRESENTATIVE COMMENTS

ID	#Edits	5 Most Representative Comments
64	93	Changed code and added caveat(s), Changed from ArrayList to List in type declaration, Changed ArrayList to HashSet, changed the inventoryItems to List interface, List instead of ArrayList
201	84	Changed code to use for each loop, Used for-each loop, Use Arrays.fill instead of loop., Updated to use ListIterator, label if panel pressed
156	68	changed to StringBuilder, changed out from String to StringBuilder, Replaced StringBuffer with StringBuilder., use StringBuilder instead of StringBuffer, Changed it to StringBuilder form StringBuffer (see comments)
61	64	Added List's initializer, Added Collection to List, added line to code, Avoid modifiying the list, I forgot to instantiate the ArrayList!
112	43	finished the code, Changed bool to boolean, changed bool to boolean, added code block, Copied code to IDE and noticed the 'serverSocket' variable was named incorrectly. Also, 'bool' is not valid Java. Should be 'boolean'.
94	35	Added a check for nullness in Book constructor., added out of range check, changed exception types, it's a little more intuitive, Added range check to Constructor., Throw the right exception type
153	31	requires a toString, many thanks to hrickards, Forgot the toString(), I corret the "String s = (String) tv.getText();" to "String s = tv.getText().toString();", missed the toString(), should be toString()
107	29	properly implement equals, implementation of hashCode and equals, corrected equals signature, impl. of hashCode & equals, generic & sc
116	29	added scanner support, changed to scanner, declared Scanner outside loop, make less verbose, No need to create a new scanner everytime.
152	26	Edited setText to use String.valueOf., replace .setText(counter) with setText(String.valueOf(counter)), Use String.valueOf() instead of concatenating an empty String with a number, Added some code for multiple timestamps, editing the remainder operation

We applied the algorithm multiple times with different number of clusters (min 50, max 3000, with a step of 10), and each time computed the *sum of squared errors (SSE)* for each cluster and averaged over all clusters. The computation of SSE was made using the distances of the edit comments[6]. Figure 2 depicts the computed SSE values versus the number of clusters. Using the Kneedle algorithm [18], we determined that the elbow of this curve is found for 390 clusters.

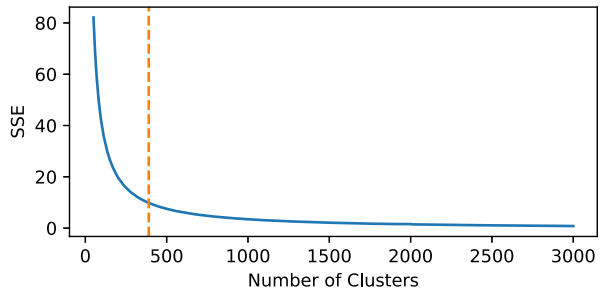

Fig. 2. Sum of squared errors using the distances between comments for different number of clusters (the dashed line depicts the elbow of the curve)

B. Identifying Edit Patterns

Our analysis produced 390 clusters, with roughly three quarters of them suggesting interesting patterns (the others were too generic/large clusters or too specific/small clusters)[7]. To illustrate the effectiveness of the clustering, we provide the 5 most representative comments of a sample subset of the clusters in Table I. The most representative comments of each cluster were defined as the ones that on average were closest to all cluster comments (computed using the distance matrix).

Several clusters are relevant to optimizations. For instance, cluster 64 refers to using the abstract List class instead of the implementation-specific ArrayList. Similarly, cluster 201 indicates the preference for for-each loops (over index loops), while cluster 116 dictates that Scanner should be instantiated outside the loop (and not every time an object is written). Other edits are relevant to bug fixes. Cluster 94 comprises null/range checks (also known as off-by-one bugs, occurring when a loop is executed one more/less time than required), while cluster 156 refers to replacing StringBuffer with StringBuilder (an important detail as StringBuffer is synchronized, whereas StringBuffer is not). Finally, there are also edits that add functionality, such as the implementations of hashCode and equals in cluster 107, which are necessary for the corresponding objects to be used as keys in hash-based collections.

[6]Given a cluster, the SSE can be computed by the sum of pairwise squared distances of its points divided by the number of its points.

[7]All scripts and instructions required to reproduce our analysis are available online at https://doi.org/10.5281/zenodo.2597655

V. CONCLUSION

In this work, we proposed a similarity scheme for edits to Stack Overflow answer posts, using their text and code snippets, and applied clustering to extract useful edit patterns. Important challenges left for future research include quantitatively evaluating our results as well as assessing whether they can be useful in different scenarios. For example, given a new answer, we could determine its closest cluster (by matching on the text and code of answer posts) and thus recommend the corresponding edit. By applying this process repetitively, we could even suggest series of edits. In addition, given that our clusters are highly cohesive (i.e. they exhibit low SSE), they may also be used to revise the generic comments (e.g. 'improved code') often produced by answer editors.

Further extending on our current research, we also plan to build a code recommender (possibly as an IDE plugin) that shall be given as input snippets and suggest edits. Other interesting challenges involve studying the co-evolution between questions and answers, or even identifying the edits to answer posts that are triggered by Stack Overflow comments.

ACKNOWLEDGMENT

This work has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-02296).

REFERENCES

[1] J. Atwood, "What does Stack Overflow want to be when it grows up?" https://blog.codinghorror.com/what-does-stack-overflow-want-to-be-when-it-grows-up/, [last accessed January, 2019].

[2] S. Baltes, C. Treude, and S. Diehl, "SOTorrent: Studying the Origin, Evolution, and Usage of Stack Overflow Code Snippets," in *Proceedings of the 16th International Conference on Mining Software Repositories (MSR 2019)*, 2019.

[3] S. Baltes, L. Dumani, C. Treude, and S. Diehl, "SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. New York, NY, USA: ACM, 2018, pp. 319–330.

[4] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*, 1st ed. O'Reilly Media, Inc., 2009.

[5] R. Holmes and G. Murphy, "Using Structural Context to Recommend Source Code Examples," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, St. Louis, MO, USA, May 2005, pp. 117–125.

[6] S. Subramanian and R. Holmes, "Making Sense of Online Code Snippets," in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*, May 2013, pp. 85–88.

[7] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. New York, NY, USA: ACM, 2014, pp. 643–652.

[8] S. Thummalapenta and T. 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)*. New York, NY, USA: ACM, 2007, pp. 204–213.

[9] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," *SIGPLAN Not.*, vol. 41, no. 10, pp. 413–430, Oct. 2006.

[10] T. Diamantopoulos and A. L. Symeonidis, "Employing Source Code Information to Improve Question-answering in Stack Overflow," in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015)*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 454–457.

[11] S.-K. Hsu and S.-J. Lin, "MACs: Mining API code snippets for code reuse," *Expert Syst. Appl.*, vol. 38, no. 6, pp. 7291–7301, Jun. 2011.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009, pp. 390–396.

[13] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining API Usage Examples from Test Code," in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 301–310.

[14] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, "Adding Examples into Java Documents," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. Washington, DC, USA: IEEE, 2009, pp. 540–544.

[15] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Enriching Documents with Examples: A Corpus Mining Approach," *ACM Trans. Inf. Syst.*, vol. 31, no. 1, pp. 1:1–1:27, 2013.

[16] N. Katirtzis, T. Diamantopoulos, and C. Sutton, "Summarizing Software API Usage Examples using Clustering Techniques," in *Proceedings of the 21th International Conference on Fundamental Approaches to Software Engineering (FASE 2018)*. Cham: Springer International Publishing, 04 2018, pp. 189–206.

[17] R. R. Sokal and C. D. Michener, "A Statistical Method for Evaluating Systematic Relationships," *University of Kansas Science Bulletin*, vol. 38, pp. 1409–1438, 1958.

[18] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior," in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, June 2011, pp. 166–171.