Summarizing Complex Development Artifacts by Mining Heterogeneous Data

Luca Ponzanelli, Andrea Mocci, Michele Lanza
REVEAL @ Faculty of Informatics, University of Lugano, Switzerland

***Abstract*—Summarization is hailed as a promising approach to reduce the amount of information that must be taken in by the person who wants to understand development artifacts, such as pieces of code, bug reports, emails, etc. However, existing approaches treat artifacts as pure textual entities, disregarding the heterogeneous and partially structured nature of most artifacts, which contain intertwined pieces of distinct type, such as source code, diffs, stack traces, human language, etc.**

We present a novel approach to augment existing summarization techniques (such as LexRank) to deal with the heterogeneous and multidimensional nature of complex artifacts. Our preliminary results on heterogeneous artifacts suggest our approach outperforms the current text-based approaches.

I. Introduction

The knowledge of a developer is often not sufficient to overcome a programming problem at hand, such as understanding the usage of a specific API. Overcoming this limitation requires the developers to seek for additional sources of information, generally resorting to asking teammates or mining web artifacts such as forums, blogs, questions and answers (Q&A) websites, and API documentation [1]. The amount of available information is huge; to get an idea it suffices to think of a web search: A developer types a query and retrieves at least ten different documents from the first page. Then, the developer needs to assess each document, or at least the title of the document, or to spot keywords from the small textual summaries provided with the result (*i.e.,* like in Google). If the developer has the feeling that a document could provide the needed information, she opens the related link and starts skimming the contents, she gets the relevant parts, and then she moves on to the next document, if she is not satisfied yet.

We can distinguish two separate phases in this process:

1) The first phase concerns the query creation, and the identification of a set of documents to be assessed by the developer. Different solutions to help developers with technological support have been proposed, such as recommender systems [2] to automatically identify relevant development artifacts, or automated web searches of Stack Overflow discussions for a specific code context [3].
2) The second phase relates to the identification of relevant information and the assessment of artifacts. Here the developer has to deal with documents whose size and complexity is not negligible. One possible solution to overcome this problem is to provide automated summaries of the artifacts.

Reducing the overload of information on the developer by means of summarization is not new in software engineering. Rastkar *et al.* [4] leverage pre-existing summarization techniques used to create summary of email threads, and generate extractive summaries of bug reports. Lotufo *et al.* [5] exploit PageRank [6] to create a general summarization approach for bug reports. Similarly, Mani *et al.* [7] proposed an unsupervised approach to bug report summarization where they employed different techniques (*e.g.,* Grasshopper, DivRank, Centroid) to generate summaries. Other types of artifacts, such as source code, have been used for ad-hoc summarization approaches. Information retrieval techniques, such as vector space model (VSM) or Latent Semantic Indexing (LSI), have been harnessed to generate summaries of code samples [8][9]. Other approaches devised ad-hoc techniques to generate human readable summaries for code samples [10][11], or to automatically generate release notes for a project [12].

All the aforementioned examples represent part of the current state of the art in summarizing software artifacts. All of these share a common limitation: they treat every artifact as a purely textual artifact, or they limit their summarization technique to a single type of artifact. According to Bacchelli *et al.* [13], *"Most of the general purpose summarization approaches are tested on well-formed, or sanitized, natural language documents. When summarizing development emails, however, we have to deal with natural language text which is often not well formed and is interleaved with languages with different syntaxes, such as code fragments, stack traces, patches, etc. [...] Currently no summarization technique takes this aspect into account [...]"*. The challenges they pointed out about the summarization of development emails are still and especially true for other types of artifacts (*i.e.,* bug reports), and for most of the online resources leveraged in software development (*e.g.,* tutorials, forums, Q&A websites).

Software artifacts cannot be considered as containers of homogeneous types of information. Current approaches rely solely on the textual and reductive interpretation of the data. The information provided by software artifacts is rather heterogeneous and includes code, text, and many other types of information whose value cannot be fully captured by a pure textual interpretation. A good example is Stack Overflow, where discussion contents include natural language, code, XML configurations, images and many other things in the same artifact.

We propose a novel technique to summarize Stack Overflow discussions by dealing with heterogeneous information units.

We extend LexRank [14], a summarization approach based on PageRank, by devising a custom similarity function for heterogeneous entities like code samples and XML configuration files. The preliminary results we obtained suggest that a holistic approach on the heterogeneity of the information could lead to better summarization results.

II. Extending LexRank

Erkan *et al.* developed *LexRank* [14], an unsupervised summarization algorithm based on PageRank [6], a well known algorithm designed by Brin and Page. Understanding LexRank requires background knowledge of the PageRank algorithm. We briefly describe it from a high-level perspective to give a general idea: To compute the relevance of a page within a network of pages, the PageRank algorithm models the behavior of a "random surfer": a user who randomly surfs the web and *"keeps clicking on links, never hitting "back" but eventually gets bored and starts on another random page"* [6].

PageRank takes as input a directed graph G representing the connection among pages (*i.e.,* hyperlinks) and returns a probability distribution P where each p_i represents the probability that the random surfer visits a page i. The probability associated to a page by this algorithm represents the centrality of this page in the network, and is called PageRank.

If we consider a document as a network of sentences, the PageRank algorithm can be extended to obtain the LexRank algorithm: Erkan *et al.* devised sentences as nodes in the graph G and used textual similarity instead of hyperlinks to define edges. The textual similarity employed in the LexRank algorithm is *tf-idf* [15]. An edge between two sentences in the graph exists if and only if the textual similarity surpasses a threshold (*e.g.,* 0.1, 0.2). Since *tf-idf* is a symmetric function (*i.e.,* $f(a, b) = f(b, a)$), the created edges are not directed, thus forcing the input graph G to be undirected as well. However, this modification does not affect the computation of the original PageRank, allowing to compute the lexical PageRank, or LexRank [14]. We could have used the LexRank algorithm "as is", to generate a summary of a software artifact. However, a software artifact is not a pure textual entity, but it contains heterogeneous types of information units. For example, if we consider a Stack Overflow discussion, and we limit the heterogeneous types of information to text and code, textual similarity, or topic-based similarity, are not appropriate solutions.

Even if we consider simple term extraction or code labeling approaches as the minimal extractive summary processes [8], where the heterogeneity of the information can be reduced to code and comments, a pure vector space model does not perform well. According to De Lucia *et al.* [16], a naïve heuristic that extracts names from class definitions outperforms vector space model approaches (*i.e.,* tf-idf) in labeling source code. They suggest how *"ad-hoc heuristics can be used to better approximate the mental model used by developers when identifying class keywords"*.

We believe that textual similarity as the only dimension to evaluate the "distance" between two heterogeneous types of information is constraining and reductive. For example, textual similarity hinders the whole information provided by a code sample, and it is not practical to devise a concept of similarity between homogeneous code elements. Also, textual similarity cannot be applied to any non-textual elements, preventing us from including information coming from images or video, as well as information derived from third party elements like users or developers.

Textual similarity should be considered as complementary part of the information within a concept of multidimensional information, where every type of information unit contained in an artifact contributes in each dimension in a different way.

<p> I am migrating from xml based spring configuration to "class" based configuration using the corresponding **@Configuration** annotation.

<p> I came across the following problem: I want to create a new bean, which has a reference to another (service) bean. Therefore I autowired this class to set this reference during bean creation. My configuration class looks as follows:

<code>
```
@Configuration
@ComponentScan(basePackages = {"com.akme"})
public class ApplicationContext {

  @Resource
  private StorageManagerBean storageManagerBean;

  @Bean(name = "/storageManager")
  public HessianServiceExporter storageManager() {
    HessianServiceExporter hessianServiceExporter =
                            new HessianServiceExporter();
    hessianServiceExporter.setServiceInterface(StorageManager.class);
    hessianServiceExporter.setService(storageManagerBean);
    return hessianServiceExporter;
  }
}
```

<p> But this doesn't work, because the causes a **BeanNotOfRequiredTypeException** exception during startup.

<code>
```
Bean named 'storageManagerBean' must be of type
[com.akme.StorageManagerBean], but was actually of type
[com.sun.proxy.$Proxy20]
```

<p> The **StorageManagerBean** is annotated with an **@Service** annotation. And the xml based configuration worked as expected:

<code>
```
<bean name="/storageManager"
  class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="storageManagerBean"/>
  <property name="serviceInterface" value="com.akme.StorageManager"/>
</bean>
```

Fig. 1. Example of Stack Overflow question with HTML tagging.

A. Contents Parsing and Information Units

There are several information units to be taken into account when describing a complex software artifact containing heterogeneous elements. In this paper we focus on Stack Overflow. Figure 1 shows an example of question taken from Stack Overflow[1]. We take advantage of the semi-structured nature of the data contained in the posts, thus relying on the tagging

[1] http://goo.gl/4sdHdB

performed by users on Stack Overflow. The contents are tagged by means of HTML tags where `<pre>` and `<code>` identify code-tagged contents. Every part tagged as code needs to be parsed to reconstruct its syntactic structure. However, such fragments tagged as code often contain natural language, and are likely to be incomplete and formally invalid with respect to the Java or XML grammar. We implemented a full-fledged *island grammar* [17] to detect and parse both Java and XML constructs in `<code>` elements. Island parsing supports the extraction of interesting constructs (the *islands*) immersed in non-interesting parts (the *water*). This enables both the separation of natural language from code, and the reconstruction of the syntactic structure of incomplete code, recovering identifiers, declarations, statements, *etc.* [18] and whatever matches the constructs of interest in our grammar.

Textual Units: whatever is not tagged as code is treated as pure text. Since we consider the very first level in the HTML DOM, textual units can actually contain small portion of code. For example, a paragraph `<p>` can have a `<code>` tag among its children. However, these code tags are most likely used to highlight tiny code fragments, like variables, class and method identifiers, or method calls.

Code Sample Units: Whatever matches a Java construct implemented in the island parser is considered a code sample unit. Whatever is not parsed as a valid or incomplete construct of interest is checked as xml. Every code sample unit carries a partial Abstract Syntax Tree (AST) generated by the island parser, representing the code and navigable to perform analyses.

XML Sample Units: Whatever matches a XML construct is considered as xml sample unit, and the related AST is generated. If the contents do not matches any xml-related construct in the parser, then it is pushed back as textual unit.

B. Meta-Information

Each information unit carries one or more meta-information types concerning a specific aspect of its contents. For example, meta-information can be extracted from the AST generated by the island parser: Every unit, including textual units, are parsed to extract code fragments. We devise the following meta-information units:

Types: It represents the set of Java types mentioned in a information unit. We consider fully qualified types (reference types), simple names matching Java convention for classes (*i.e.,* begin with a capital letter), and primitive types (*e.g.,* int, double). This meta-information applies to all the information units (*e.g.,* types mentioned in natural language and extracted with the island parser).

Variable Names: All the AST nodes matching a variable name are extracted and stored in this meta-information node. This applies to code samples and textual information units.

Invocation Names: All the AST nodes matching a method invocation are extracted and the name of the invoked method is stored in the meta-information node. We discard arguments passed to the method. This meta-information applies to both code samples and textual information units.

Natural Language: We also complement the meta-information with pure textual information. For each type of information unit we can generate a *tf-idf* vector that will be used later on in the calculation of the similarity.

C. Similarity Function

The final step in extending the LexRank algorithm is to devise a similarity function that takes two information units and returns a similarity value, which ranges between 0 and 1. Each type of information unit can carry an arbitrary number of meta-information. To explain how we construct the similarity of two information units from their meta-information, and thus how the similarity function works, we go through an example scenario. Consider two information units U_1 and U_2 of different type. Since we can compare only shared meta-information, let $T_{1,2}$ be the set of shared types of meta information between the units, and let $M(U_i, t)$ be the meta-information of type t for the information unit U_i.

We define the similarity vector V as:

$$V_{x,y} = \langle v_0, \ldots, v_{|T_{1,2}|} \rangle$$
$$with\ v_i = M(U_1, t_i) \sim M(U_2, t_i)\ and\ t_i \in T_{1,2}$$

Each element v_i of the vector V represents the similarity value between two homogeneous meta-information units, and ranges in the interval $[0, 1]$. Once we have computed all the elements of V, we calculate the general similarity between two information units U_1 and U_2 as the norm of the vector:

$$f_{sim}(U_1, U_2) = \frac{||V_{1,2}||}{\sqrt{dim(V_{1,2})}}$$

Since we want f_{sim} to provide a value in the range $[0, 1]$, we divide the norm of the vector by the maximum possible value of the norm. We can use this similarity function with the LexRank algorithm to compute the centrality of an information unit inside the network of information units of an artifacts.

D. Summary Generation

Once the extended Lex Rank is computed, each information node receives a centrality value. We select the top n units, according to the percentage of the summary we want to show. In essence the user can decide how concise the summary will be. However, we keep one single constraint on the original structure of the Stack Overflow discussion: There must always be one information unit from the question and one information unit from one of the answers. The unit can either be extracted from the body or from one of the related comments.

The generated summary is interactive and allows the developer to incrementally disclose information on demand. By means of a slider, the developer chooses the percentage of the original discussion that she wants to see.

III. Preliminary Evaluation

We present a preliminary evaluation of our approach, starting from its setup.

A. Experimental Setup

We involved nine people (6 Master students and 3 PhD students) to annotate information units on Stack Overflow discussions through a web application developed by us, illustrated in Figure 2.

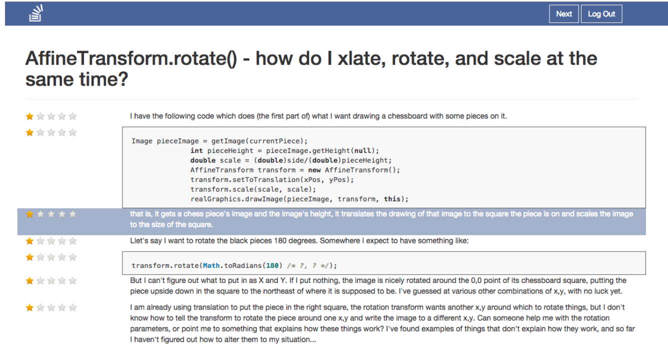

Fig. 2. Example of Stack Overflow discussion proposed to users.

The application presents the discussion with a random order to each user, and shows the contents of a discussion by separating each single information unit from the other. Each subject annotated the units of 9 different discussions. Every user gave a rating to each information unit by providing the number of stars on a Likert scale between one and five. We asked people to give a rating according to the prominence of the unit in the discussion.

B. Evaluation Approach

According to the current state of the art, there is no standardized way of evaluating artifacts summaries [13]. Moreover, nobody tackled the evaluation of summaries containing heterogeneous information units. We devised our own approach to evaluate our summaries. We tried to simulate the generation of a "golden standard" summary. Given a Stack Overflow discussion, we calculate the average rating received for each information unit, and we sort the units in descending way. Then, we fix a percentage value that represents the subset of information unit we want to show in the summary. This subset represents the golden summary for a given percentage.

Once we have constructed the golden summary of a discussion, we generate a summary with our own approach. To evaluate the generated summary we calculate the precision, that is, the percentage of units selected by our approach that matches the units in the golden summary.

C. Preliminary Results

Table I shows the preliminary results we obtained. We created six golden summaries representing 5%, 10%, 15%, 25%, 35%, and 50% of the information units contained in a Stack Overflow discussion. To have a reference value, we applied the same evaluation approach on the original LexRank algorithm, that is, every information unit treated as pure textual information. Best results are reported in bold.

At a first observation, we can distinguish four different scenarios of the two approaches. The first scenario concerns discussions D3 and D5, where results show no difference in terms of performance. The second scenario concerns discussion D1, where the original LexRank algorithm performs better than our approach. On the opposite side, we have the third scenario, where for discussion D4, D7, and D8 our approach outperforms the pure textual based approach. In the fourth scenario we have discussions D2, D6, and D9, where each approach performs better than the others depending on the size of the summary.

TABLE I
PRECISION ON HUMAN ANNOTATED DISCUSSIONS.

Our Approach

Size	D1	D2	D3	D4	D5	D6	D7	D8	D9
5%	0%	**50%**	0%	**100%**	100%	0%	50%	0%	33%
10%	33%	**25%**	50%	**67%**	50%	**67%**	**50%**	0%	**33%**
15%	60%	43%	33%	**60%**	75%	**40%**	**33%**	20%	**44%**
25%	56%	55%	30%	33%	43%	25%	**50%**	**38%**	**53%**
35%	62%	75%	50%	50%	30%	45%	**57%**	**42%**	52%
50%	58%	57%	57%	56%	43%	65%	**70%**	47%	67%

Original LexRank Algorithm

Size	D1	D2	D3	D4	D5	D6	D7	D8	D9
5%	0%	0%	0%	0%	100%	0%	50%	0%	33%
10%	33%	0%	50%	33%	50%	33%	25%	0%	17%
15%	60%	43%	33%	40%	75%	20%	17%	20%	33%
25%	**67%**	55%	30%	33%	43%	**38%**	40%	25%	47%
35%	**69%**	75%	50%	50%	**40%**	45%	50%	33%	52%
50%	**63%**	**61%**	57%	56%	43%	65%	65%	47%	**70%**

In general, the results show that our approach generally either outperforms the classic LexRank algorithm, and specifically it does so on the shorter summaries. This is what the user in the end finds more desirable, since shorter summaries reduce the information that needs to be taken in by a person. This is also especially true in the case of heterogeneous artifacts, where a person is otherwise forced to comprehend the various distinct pieces of information of different nature.

IV. Conclusion

Summarizing complex software artifacts is a non-trivial task due to their heterogeneous and multidimensional nature. Different fragments of information (*e.g.,* code, text, xml) co-exist in the same artifacts and contribute differently to the overall knowledge contained in them. Current approaches in summarization do not take this fundamental fact into account, and reductively treat artifacts as if they were purely textual.

We propose a novel approach to summarization, adapting LexRank to integrate different aspects of the heterogeneous information contained in them. We obtained promising results in our preliminary evaluation, and we discussed how results suggest that a holistic point of view on the heterogeneous information contained in software artifacts is worth being explored to improve the current state-of-the-art approaches.

Acknowledgments

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

REFERENCES

[1] M. Umarji, S. Sim, and C. Lopes, "Archetypal internet-scale source code searching," in *Proceedings of OSS 2008*, 2008, pp. 257–263.

[2] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, pp. 80–86, 2010.

[3] L. Ponzanelli, "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter," in *In Proceedings of MSR 2014 (11th Working Conference on Mining Software Repositories)*. ACM, 2014, pp. 102–111.

[4] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transaction on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[5] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the "hurried" bug report reading process to summarize bug reports," in *Proceedings of ICSM 2012 (28th IEEE International Conference on Software Maintenance)*. IEEE Computer Society, 2012, pp. 430–439.

[6] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of WWW 1998 (7th International Conference on World Wide Web)*. Elsevier Science Publishers B. V., 1998, pp. 107–117.

[7] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: Approach for unsupervised bug report summarization," in *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 11:1–11:11.

[8] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of WCRE 2010 (17th Working Conference on Reverse Engineering*. ACM, 2010, pp. 35–44.

[9] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the ICSE 2014 (36th International Conference on Software Engineering)*. ACM, 2014, pp. 390–401.

[10] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proceedings of ICPC 2013 (21st International Conference on Program Comprehension)*. IEEE Computer Society, 2013, pp. 23–32.

[11] S. Rastkar, G. Murphy, and A. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *Proceedings of ICSM 2011 (27th IEEE International Conference on Software Maintenance)*. IEEE CS, September 2011, pp. 103–112.

[12] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proceedings of FSE 2014 (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 484–495.

[13] A. Bacchelli, M. Lanza, and E. Mastrodicasa, "On the road to hades–helpful automatic development email summarization," in *In Proceedings of TAinSM 2012 (1st International Workshop on the Next Five Years of Text Analysis in Software Maintenance)*, 2012.

[14] G. Erkan and D. R. Radev, "Lexrank: Graph-based lexical centrality as salience in text summarization," *Journal of Artificial Intelligence Research*, vol. 22, no. 1, pp. 457–479, Dec. 2004.

[15] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[16] A. D. Lucia, M. D. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?" in *Proceedings of ICPC 2012 (20th IEEE International Conference on Program Comprehension)*. IEEE Computer Society, 2012, pp. 193–202.

[17] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of WCRE 2001 (8th Working Conference on Reverse Engineering)*. IEEE CS, 2001, pp. 13–22.

[18] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting structured data from natural language documents with island parsing," in *In Proceedings of ASE 2011 (26th IEEE/ACM International Conference On Automated Software Engineering)*, 2011, pp. 476–479.