A Lightweight Approach to Uncover Technical Information in Unstructured Data

Nicolas Bettenburg, Bram Adams, Ahmed E. Hassan
Software Analysis and Intelligence Lab
Queen's University
Kingston, Ontario, Canada
Email: {nicbet,bram,ahmed}@cs.queensu.ca

Michel Smidt
Dept. of Computer Science
University of Bremen
Bremen, Germany
Email: michel@informatik.uni-bremen.de

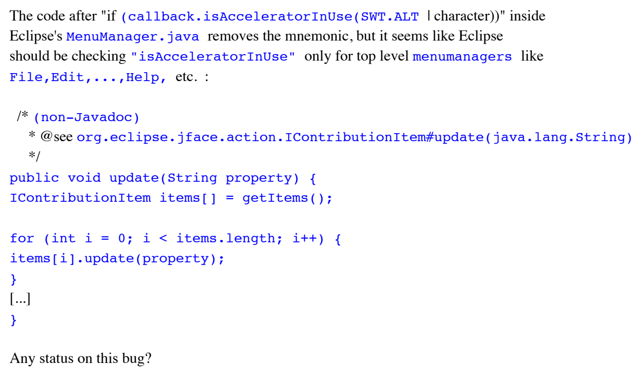
***Abstract*—Developer communication through email, chat, or issue report comments consists mostly of largely unstructured data, i.e., natural language text, mixed with technical information such as project-specific jargon, abbreviations, source code patches, stack traces and identifiers. These technical artifacts represent a valuable source of knowledge on the technical part of the system, with a wide range of applications from establishing traceability links to creating project-specific vocabularies. However, the free-style delimiters between natural language and technical content make the mining of technical artifacts challenging. As a first step towards a general-purpose technique to extracting all kinds of technical information from unstructured data, we present a lightweight approach to untangle technical artifacts and natural language text. Our approach is based on existing spell checking tools, which are well-understood, fast, readily available across platforms and impartial to different kinds of technical artifacts. Through a handcrafted benchmark, we demonstrate that our approach is able to successfully uncover a wide range of technical information in unstructured data.**

***Keywords*-text mining, language analysis, unstructured data, technical information.**

I. Introduction

Every software system has a unique history of design decisions, software changes, as well as development and maintenance effort. This history is captured throughout the development process in the variety of repositories used to store data during the collaborative development process. As this data contains the knowledge and rationale behind the evolution of a software system, it is valuable for many different fields, in particular program comprehension, and hence should be made available to practitioners and researchers alike.

However, much of the information surrounding the development process comes in the form of *unstructured data* [1], which is conceptually different from the sources of structured data that researchers have used in previous research. Structured data (e.g., source code) is well-defined and can be readily parsed and understood by computer machinery. Unstructured data (e.g., developer communication, issue reports, documentation, email or meeting notes [2]), consists of a mixture of natural language text and *technical information*, such as code fragments, abbreviations, references to objects in the source code, file names, logging information or project-specific terms. As such, mining unstructured data is challenging: it is meant for the exchange of information between humans, rather than automated processing using computer machinery. Figure 1 presents an example of technical information commonly found in unstructured data.



The code after "if `(callback.isAcceleratorInUse(SWT.ALT` | character))" inside Eclipse's `MenuManager.java` removes the mnemonic, but it seems like Eclipse should be checking `"isAcceleratorInUse"` only for top level `menumanagers` like `File,Edit,...,Help,` etc. :

```
 /* (non-Javadoc)
   * @see org.eclipse.jface.action.IContributionItem#update(java.lang.String)
   */
public void update(String property) {
IContributionItem items[] = getItems();

for (int i = 0; i < items.length; i++) {
items[i].update(property);
}
[...]
}
```

Any status on this bug?

Figure 1. Examples of technical information uncovered by a prototype implementation of the approach proposed in this paper. (Eclipse Platform Bug #208626).

Recent approaches for discovering technical information in unstructured data [3]–[5] have focussed on recognizing and extracting only particular types of technical information, such as class names [3], stack traces, or patches [5]. In order to resolve the inherent ambiguities between natural language text and technical information, these approaches are highly specialized and tailored towards their specific use cases, and limited in their scope. Furthermore, many kinds of technical information (e.g, project-specific jargon or abbreviations) cannot be extracted by any of the existing techniques.

As a first step towards a lightweight, general-purpose approach to uncovering technical information in unstructured data, this work presents an approach that makes use of state-of-the-art tools for checking and correcting the spelling and grammar of electronically written texts. Technical information is conceptually different from natural language text: it often consists of words that are not part of standard language dictionaries, violate grammatical conventions, and do not respect morphological language rules. These characteristics render modern spellcheckers ideal candidates for lightweight classifiers of natural language.

Through a case study on unstructured data from mailing list and issue report repositories of two open-source projects, we demonstrate the capability of our approach to uncover technical information inside unstructured data, while at the same time being resistant to reporting actual spelling or grammar mistakes.

The rest of this paper is organized as follows. Section 2 presents an overview of related work and background. Section 3 presents our approach from both a conceptual and an actual implementation perspective. In Section 4, we present the evaluation of our approach through a hand crafted benchmark on developer email and issue report discussions. We conclude our work and present future research opportunities in Section 5.

II. Background and Related Work

Past research has been concerned with the extraction of technical information from software repositories, to assist program comprehension [6], [7], understand historical changes [8]–[10] and predict future changes [11], and to measure and analyze different dimensions of historic software development to help practitioners make informed decisions in the future, and predict software errors [12]–[14].

Fischer et al. and Sliwerski et al. were among the first to use technical information (issue report identifiers) embedded in the natural language text descriptions of changes in commit messages, to link software changes to defects [13], [15]. Mockus et al. show that the text describing a change recorded through commit messages is essential for understanding the rationale behind changes and emphasizes the importance of natural language documentation for practitioners and researchers alike.

Recent research concerned with information in unstructured data has mostly focussed on establishing traceability links [3], [16], [17] between source code and documentation surrounding the development process, summarizing communication [2], [18], and bug triage [19].

The most closely related work to this paper is the work on techniques to uncover source code entities in e-mails [20], and classifying text into source code and natural language text on a line-level granularity [3]. Bettenburg et al. presented the use of island parsing and specialized heuristics based on regular expressions to extract structural information from bug reports [5].

Our work is different from past research in the area, in that we aim to uncover technical information in unstructured data by using spell checkers as a lightweight classification-proxy to determine which parts of the text are natural language text and which parts are not. Our approach aims at being general enough to be readily available for any kind of input beyond commit messages, bug reports or e-mail. Furthermore, our approach does not focus on a particular type of technical information, such as bug report identifiers or source code entities, but rather to return information in unstructured data that is not considered natural language text. Such a general set of technical information has the advantage that it can be easily pruned later on, by applying further heuristics to retain only a particular kind of technical information of interest.

III. Approach

In the following, we present our approach from both a conceptual perspective and the concrete perspective of our working prototype.

Conceptual Approach

In this paper, we propose to use existing spellchecking tools to untangle natural language text and technical information from unstructured data. Many of today's state of the art techniques for spellchecking use morphological language analysis, which describes the identification and description of the smallest linguistic units that carry a semantic meaning, called *morphemes*. As such, morphemes are different from the concept of a single word: one or more morphemes composed form a word. For example, the English word "unbearable", is composed of three morphemes, "un", "bear", and "able". This kind of analysis is able to effectively cope with compound words, inflection and other peculiarities of natural language, while at the same time being sensitive to text (technical information) that does not adhere to the morphological rules.

For the purpose of our study, we define technical information as those parts of unstructured data that is not natural language text. This definition includes, but is not restricted to: source code, file names, technical terms, project-specific jargon, source code entities (such as classes or identifier names), or abbreviations.

Concrete Approach

In order to uncover technical information, we first transform the input text in a stream of tokens by splitting the input text whenever we encounter one or more whitespace characters, or punctation followed by a whitespace (sentence delimiters). This is a common approach for morphological language analysis of Western text, where words are delimited by whitespace. If we were to apply our method to Chinese or Japanese input text, we would need to modify tokenization accordingly.

After thorough testing of 15 open-source spellchecking tools, we select the following three popular tools for further study. `Hunspell` is an open-source spellchecking and morphological language analysis framework, which has found extensive use in the `OpenOffice` and `Mozilla` application suites. `Jazzy` is based on the *double metaphone* phonetic language analysis algorithm [21], which transforms words into phoneme codes and compares these to a user-defined dictionary. `JOrtho` performs spell checking by comparing a given input word to large word dictionaries compiled from the `Wiktionary`[1] project.

Next, we run the spellchecker on each token and flag it, depending on wether the spellchecker reported a spelling error or not. Since our goal is to find technical text, rather than spelling mistakes, we iterated over each flagged token in a second pass, executing three different, simple heuristics. If at least one heuristic holds on a flagged token, we mark the token as belonging to the domain of technical information. The heuristics we use are described in the following.

H1: **Camel Case**
We consider the following four cases of camel case to be indicators of technical text: (1) the standard case `CamelCase` is often used for type names and references to source code entities; (2) the interior case `camelCase` is often used for identifier names; (3) capital letters at the end `CamelCASE`, and (4) all capital letters `CAMELCASE`, are often used in abbreviations. We implemented this heuristic with a simple pattern matching using the following regular expressions:

```
(\b([A-Z_][a-z_0-9]*)+[A-Z_0-9][a-z_0-9]+
                        ([A-Z_0-9][a-z_0-9]*)*\b)
(\b([a-z_][a-z_0-9]*)+([A-Z_0-9][a-z_0-9]*)+\b)
(\b([A-Z_][A-Z_0-9]+)\b)
(\b(([A-Z_][a-z_0-9]*)+([A-Z_0-9]+))\b)
```

H2: **Programming Language Keywords**
We compiled a comprehensive list of reserved keywords for the JAVA, C, C++, C#, Pascal, Delphi, Perl, PHP, Bash, HTML and JavaScript languages. If a token is flagged as a spelling mistake, but matches one of the keywords in this list, it is highly likely to be part of a source code.

H3: **Special Characters**
Natural language words usually do not contain special characters within their word boundaries. When a token is flagged as a spelling mistake, we count the number of non-alphanumeric characters in the token and consider it as technical text, if we find more than two special characters.

IV. Evaluation

We evaluate the ability of each of the three selected spellchecking frameworks to untangle natural language text and technical information from unstructured data through a hand-crafted benchmark. We performed a random sampling of 20 issue reports from the `ECLIPSE` project and 20 email discussions from the `PostgreSQL` developer mailing list, containing source code, stack traces, patches and other technical entities. The size of this random sample describes our results across the overall population at a confidence interval of 15%. We annotated technical information in each document by hand, using a graphical tool written for this purpose.

[1]http://wiktionary.org

Tool	**Precision**	**Recall**
JOrtho	88.01%	64.31%
Jazzy	84.16%	68.30%
Hunspell	86.40%	68.34%

Table I
Results of Benchmark

The tool allows the user to select a portion of the text inside a text viewer and annotate it as technical text. When the user then activates a particular spellchecking framework, the portion of the text will be annotated with different colours depending on whether the spellchecker flagged a portion of the text which was previously not annotated (false positives, FP), which was annotated but not flagged by the spellchecker (false negatives, FN), and which was flagged, as well as previously annotated (true positives, TP).

We then measure the average *precision* and *recall* of each spellchecker S_i across all documents in the benchmark. These measures are defined as

$$\begin{aligned} Precision(S_i) &= \frac{|TP_{S_i}|}{|TP_{S_i} + FP_{S_i}|} \\ Recall(S_i) &= \frac{|TP_{S_i}|}{|TP_{S_i} + FN_{S_i}|} \end{aligned}$$

The results of our manual benchmark are presented in Table I. Overall, we found all three spellcheckers to perform well, with a precision between 84.16% and 88.01%, and a recall between 64.31% and 68.34%. The most common error across all tested spellcheckers with respect to precision were spelling mistakes that were not distinguishable from technical text, such as *"found.We"*, or *"another(no one else would)"* resembling package names or method calls. The rather moderate recall can be mainly attributed to the resemblance of many technical items to natural language text, for example source code identifiers are often words present in English dictionaries, e.g., *"Task"*.

In addition to our fine-grained performance analysis, we conducted an experiment to compare the use of technical information uncovered by our approach for a more specialized task, presented by Bacchelli et al., i.e., extracting source code from email [3]. As the latter approach operates on a line-level granularity, we augmented our spellchecking-based technique to consider a line of text as source-code, if more than seventy percent of text in that line was flagged as technical information by our approach. We applied both approaches to the same data set of 40 documents (20 issue reports and 20 developer emails) used in our previous evaluation. The baseline for this experiment was established through the same benchmark annotation tool used in our previous evaluation. In terms of precision, our approach was able to classify 89.27% of lines correctly as source code, compared to 66.13% percent of lines correctly classified by the state-of-the-art technique. In terms of recall, our approach was able to recognize 86.46% of all source code lines correctly, compared to 69.37% of source code lines recognized by the state-of-the-art technique. All performance differences are statistically significant at $p < 0.001$. Overall, we were able to improve on the best existing technique by 23.14% (precision) and 16.09% (recall) respectively.

V. Conclusions and Future Work

In this paper, we presented a lightweight approach to finding technical information in unstructured data, as a first step to making technical information readily available for researchers and practitioners. The evaluation of our approach demonstrates that readily available spellchecking tools, when paired with additional lightweight heuristics, are able to successfully untangle technical information and natural language text. In future work, we plan to study the use of additional heuristics to increase recall and carry out a more detailed evaluation on different kinds of technical information through an extended benchmark.

References

[1] N. Bettenburg and B. Adams, "Workshop on mining unstructured data (mud) because "mining unstructured data is like fishing in muddy waters"!" *Reverse Engineering, Working Conference on*, vol. 0, pp. 277–278, 2010.

[2] E. Shihab, Z. M. Jiang, and A. E. Hassan, "On the use of internet relay chat (irc) meetings by developers of the gnome gtk+ project," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE Computer Society, 2009, pp. 107–110.

[3] A. Bacchelli, M. D'Ambros, and M. Lanza, "Extracting source code from e-mails," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*. IEEE Computer Society, 2010, pp. 24–33.

[4] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization." in *SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, 2004, pp. 92–97.

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *MSR '08: Proceedings of the Fifth International Workshop on Mining Software Repositories*, May 2008.

[6] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan, "Understanding the rationale for updating a function's comment," in *ICSM '08: Proc. of the 24th IEEE International Conf. on Software Maintenance*, 2008, pp. 167–176.

[7] E. Nurvitadhi, W. W. Leung, and C. Cook, "Do class comments aid java program understanding?" *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, vol. 1, pp. T3C–13–T3C–17, 2003.

[8] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis," *Proc. International Workshop on Mining Software Repositories ...*, 2004.

[9] A. Mockus and L. G. Votta, "Identifying reasons for software change using historic databases," 2000, pp. 120–130.

[10] A. E. Hassan and R. C. Holt, "Studying the evolution of software systems using evolutionary code extractors," in *IWPSE '04: Proc. of the Principles of Software Evolution, 7th International Workshop*. IEEE Computer Society, 2004, pp. 76–81.

[11] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering*, 2005.

[12] A. Zeller, "Learning from software," *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, 2008.

[13] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do Changes Induce Fixes?" in *Proceedings of the Second International Workshop on Mining Software Repositories*, 2005, pp. 24–28.

[14] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," 2006, pp. 452–461.

[15] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," *Reverse Engineering*, Jan 2003.

[16] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 970–983, 2002.

[17] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. IEEE Computer Society, 2003, pp. 125–135.

[18] A. Hindle, M. W. Godfrey, and R. C. Holt, "What's hot and what's not: Windowed developer topic analysis," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 339–348, 2009.

[19] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. ACM, 2006, pp. 361–370.

[20] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, pp. 375–384.

[21] L. Philips, "The double metaphone search algorithm," *C/C++ Users J.*, vol. 18, pp. 38–43, June 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=349124.349132