Classifying Unstructured Data into Natural Language Text and Technical Information

Thorsten Merten
Bonn-Rhein-Sieg University of Applied Sciences, Germany
Dpt. of Computer Science
thorsten.merten@h-brs.de

Bastian Mager
Bonn-Rhein-Sieg University of Applied Sciences, Germany
Dpt. of Computer Science
bastian.mager.2010w@informatik.h-brs.de

Simone Bürsner
Bonn-Rhein-Sieg University of Applied Sciences, Germany
Dpt. of Computer Science
simone.buersner@h-brs.de

Barbara Paech
University of Heidelberg, Germany
Software Engineering Group
paech@informatik.uni-heidelberg.de

ABSTRACT

Software repository data, for example in issue tracking systems, include natural language text and technical information, which includes anything from log files via code snippets to stack traces.

However, data mining is often only interested in one of the two types e.g. in natural language text when looking at text mining. Regardless of which type is being investigated, any techniques used have to deal with noise caused by fragments of the other type i.e. methods interested in natural language have to deal with technical fragments and vice versa.

This paper proposes an approach to classify unstructured data, e.g. development documents, into *natural language text* and *technical information* using a mixture of text heuristics and agglomerative hierarchical clustering.

The approach was evaluated using 225 manually annotated text passages from developer emails and issue tracker data. Using white space tokenization as a basis, the overall precision of the approach is 0.84 and the recall is 0.85.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms

Keywords

Mining software repositories, unstructured data, preprocessing, heuristics, hierarchical clustering

MSR'14, May 31 – June 1, 2014, Hyderabad, India
ACM 978-1-4503-2863-0/14/05
http://dx.doi.org/10.1145/2597073.2597112

1. INTRODUCTION

Software repositories try to store data in a structured and machine-readable manner. However, developers tend not to use these facilities and store their data in a way most convenient for fulfilling their goals. For example, a bug report in an issue tracking system (ITS) contains *natural language text* to describe the bug, stack traces to give *technical information* about the problem and additionally code snippets may be pasted in a comment to discuss how the bug should be fixed. A similar mixture of document types can be found in other repositories, e.g. in development mailing lists. Though many ITSs allow data structuring, e.g. by using tags to identify code or categories for issues, these methods are not always used [9].

Why is this mixture of document types a problem in Mining Software Repositories? We know that preprocessing is essential for data and especially text mining techniques [8, pp. 84] [11, pp. 349] [7, p. 57] and that machine learning should be applied to homogenous content. Furthermore, most feature selection techniques take the structure of *natural language* into account. In contrast, code snippets, patches, log files and stack traces look very different from *natural language text*, as they include repetitive terms or lines. This difference reduces data quality and makes data inhomogenous and noisy and machine learning techniques harder to apply.

The approach presented in this paper classifies unstructured data into *natural language text* and other *technical information*. *Technical information* includes code, patches, stack traces or log file excerpts. The approach can be used either as a preprocessing step, if an analyzed software repository contains noisy *natural language text*[1], or it can be applied on its own e.g. to identify code snippets. It was evaluated on a total of 225 documents extracted from ITSs and mailing lists from nine different software projects.

In the next section, the details of the approach are presented. Section 3 describes the evaluation procedure and evaluation results; Section 4 discusses related work and section 5 concludes the paper and shows directions for current and future work.

[1] where noise is technical information

2. THE APPROACH

The proposed approach[2] segments the documents either by line or white space[3] tokenization. Then it uses multiple heuristics to initially identify the *technical information* and finally an agglomerative hierarchical clustering is used to find accumulations of technical and natural language segments. It is independent of both the document's programming language(s) and the natural language(s) and does not require an explicit training phase[4].

2.1 Heuristics for Text Detection

The current implementation uses the following heuristics to identify *technical information*: 1*)* A *Keyword Detection*, which contains keywords from different programming languages and was compiled using the ten most popular languages from the TIOBE index[5], 2*)* a *Fuzzy Line Equality Detection*, which tries to find similar-looking lines, often occurring in log files or stack traces, 3*)* a *Fuzzy Patch Detection*, which searches line by line for common components of a patch (according to the `diff`-format), similar to the detection method in [5], 4*)* several *Regular Expression Detections*, which are used to search for different occurrences of special characters (e.g. parentheses or asterisks), indentations or camelcase. Each heuristic is applied to the text and marks a set of detected sections as *technical information*, comparable to using a highlighter. Since several heuristics are in use, text can potentially be marked multiple times. This, however, has no influence, i.e. no weight is added to multiply marked text.

Each heuristics can be customized according to the analyzed repository. For example, *regular expressions* can be modified for an ITS that uses tags to render code snippets[6]. Similarly the *Fuzzy Line Equality* heuristic can be set to compare the next n lines to the current line. This setting is often useful if a log file or a stack trace has a similar format in general but includes different looking lines in between. Our experiments showed that $n = 3$ is a reasonable default for this parameter and this default has also been used in evaluation.

2.2 Clustering and Classification

The heuristics alone generally fail in two ways: Firstly, they mark some *natural language text* as technical by mistake. Secondly they miss some *technical information*. For example, in Figure 1 the heuristics will only detect the segments printed in bold font.

somenatural language text...
void detector (**int** detection) **{**
    print(” For the heuristics ‘detector’, ‘detection’ and most of this text look like natural **language”)**
}
more code and technical information *...*

Figure 1: Classification example before clustering

[2]source code and evaluation documents can be downloaded at http://www2.inf.fh-bonn-rhein-sieg.de/∼tmerte2m
[3]every character sequence separated by one or more space or tab characters
[4]heuristics and clustering thresholds can be adjusted
[5]http://www.tiobe.com , 09/2013
[6]very common tags are `<pre>` and `</pre>` to identify code

Therefore an agglomerative (or bottom-up) clustering algorithm [10, p. 502] is applied to the tokens of the text to correct the heuristics' results. The clustering tries to 'fill the gaps'. The clustering algorithm is presented in Figure 2. Initially, it considers every token x_i as its own cluster c_i, where C is the set of all clusters. In each iteration, two clusters of C are merged based on their similarity as $sim(\chi)$ shown in Equation 1, therefore it is possible to merge similar clusters even if they are not close to each other. The clustering is stopped if the minimum similarity s_{min} cannot be reached.

$$sim(c_i, c_j) = \begin{cases} 1 & \text{if } intersect(c_i, c_j) = 1 \\ 1 - \frac{dist(c_i, c_j)}{\sum_{c \in C} |c|} & \text{if } class(c_i) = class(c_j) \\ 0 & \text{else} \end{cases} \quad (1)$$

The *class* function in Equation 1 returns either *natural language text* or *technical information* if the ratio of *technical information* characters, as determined by the heuristics is greater or equal 0.7. The *dist* function returns the amount of tokens between c_i and c_j. Overall, the clustering presumes that smaller, differently classified clusters between clusters of the same class have been wrongly classified by the heuristics. The effect of the clustering is that those smaller clusters are gulped down, when merging.

The final result of the algorithm is an annotated text where each token is classified as *technical information* or *natural language text*.

Given: a set $\chi = x_1, \ldots, x_n$ of objects
a function sim: $P(\chi) \times P(\chi) \to \mathbb{R}$

1: **for** $i := 1$ to n **do**
2: $c_i := \{x_i\}$
3: **end for**
4: $C := \{c_i, \ldots, c_n\}$
5: $j := n + 1$
6: **while** $|C| > 1$ **do**
7: $(c_{n1}, c_{n2}) := \arg\max_{(c_u, c_v) \in C \times C} \text{sim}(c_u, c_v)$
8: **if** sim $(c_{n1}, c_{n2}) < s_{min}$ **then**
9: stop
10: **end if**
11: $c_j := c_{n1} \cup c_{n2}$
12: $C := C \backslash \{c_{n1}, c_{n2}\} \cup \{c_j\}$
13: $j := j + 1$
14: **end while**

Figure 2: Bottom-up clustering (adapted from [10])

3. EVALUATION

3.1 Datasets

To evaluate the approach, 225 documents were sampled randomly from issue trackers and mailing list of nine Open Source projects (25 Documents/Project). Most data were obtained from Apache projects since they are easily accessible and most of them use multiple programming languages. Each document was manually annotated using white space tokenization and results were compared to the automated results.

We sampled from multiple projects to show that reasonable default settings of the approach can handle different

Table 1: Evaluated documents

Project	Language	Source	# of documents with nl only	c	ls	p	Link
Apache ActiveMQ	Java, C, C++, Ruby, Perl, Python (+2)	ITS	1	13	14	1	http://issues.apache.org/jira/browse/AMQ
Linux Kernel	C	ITS	0	12	9	11	http://bugzilla.kernel.org
Mozilla Core + JSS	C++, Java	ITS	1	13	2	13	http://bugzilla.mozilla.org
Apache OpenOffice	C++	ITS	0	23	2	0	http://issues.apache.org/ooo
Apache Jmeter	Java	ITS	2	13	7	3	http://jmeter.apache.org/issues.html
Apache OFBiz	C, C++, C#, Java, PHP, Python, Ruby	ITS	2	19	6	0	http://issues.apache.org/jira/browse/OFBIZ
Apache Avro	Java, XML, Python, Ruby, PHP (+4)	Mail	2	19	6	2	http://mail-archives.apache.org/mod_mbox/avro-dev
Apache Camel	Java, Groovy, JavaScript, XML	Mail	1	19	12	1	http://mail-archives.apache.org/mod_mbox/camel-dev
Apache Thrift	ActionScript, D, Delphi, Erlang (+14)	Mail	1	19	6	1	http://mail-archives.apache.org/mod_mbox/thrift-dev

document types and programming languages. Table 1 provides information on the sources of the evaluated documents, including their programming language and source type (ITS or email). Additionally it shows the number of manually annotated documents containing *natural language text* (nl) only or a mixture of *natural language text* and code (c), log files or stack traces (l/s) and patches (p). The average document size was 1698 characters.

3.2 Analysis of the Obtained Results

True positives, true negatives, false positives and false negatives were measured by comparing the output of the algorithm to that of the manually annotated documents. Manual annotation used white space tokenization. We also conducted line tokenization, considering lines with less than 50% *technical information* as *natural language text*[7].

The evaluation results are presented using the standard measures of precision (p), recall (r) and F-measure (F_1). Table 2 shows the results for each project using line segmentation and Table 3 shows the results for white space tokenization. To show that, even without further training, simple heuristics are applicable for multiple projects, no heuristic parameters were changed. Multiple runs with different minimum similarities were made, varying s_{min} from 0.5 to 1.0 in 0.05 steps.

The best result using line tokenization over all projects was retrieved with a *minimum cluster similarity* $s_{min} = 0.9$. Here the approach reached $p = 0.77$ and $r = 0.88$ giving $F_1 = 0.82$. The heuristics alone, without applying the clustering, gave $p = 0.69$ and $r = 0.88$, therefore the clustering improved the precision by about 8%.

For white space tokenization, we received $p = 0.84$ and $r = 0.85$ giving $F_1 = 0.84$. The values before clustering were $p = 0.67$ and $r = 0.84$. Hence the influence of clustering is even more noticeable, in this case improving the precision by just over 17% and the recall by 1%.

In all experiments, the clustering algorithm reached a maximum performance with s_{min} from 0.8 to 0.95, which therefore present reasonable default values for s_{min}. Figure 3 confirms this result showing different values for s_{min} for line and white space (dotted) tokenizations. Furthermore, it can be seen that the clustering step improves precision and recall for every value of s_{min}.

[7]In the evaluation documents, such mixed lines generally contained only about 4% technical information

Table 2: Evaluation Line Tokenization

Project	s_{min}	p	r	F_1
Apache ActiveMQ	0.8	0.91	0.91	0.91
Apache Avro	0.85	0.65	0.83	0.73
Apache Camel	0.95	0.74	0.90	0.81
Apache JMeter	0.85	0.88	0.896	0.89
Apache OFBiz	0.95	0.82	0.88	0.84
Apache OpenOffice	1.0	0.80	0.91	0.86
Apache Thrift	0.9	0.58	0.83	0.68
Linux Kernel	0.85	0.87	0.94	0.91
Mozilla Core + JSS	0.75	0.78	0.86	0.82
Overall	0.9	0.77	0.88	0.82
Overall (no clustering)	n/a	0.69	0.88	0.77

Table 3: Evaluation Whitespace Tokenization

Project	s_{min}	p	r	F_1
Apache ActiveMQ	0.85	0.93	0.89	0.91
Apache Avro	0.95	0.74	0.87	0.797
Apache Camel	0.9	0.76	0.86	0.81
Apache JMeter	0.95	0.912	0.83	0.87
Apache OFBiz	0.95	0.80	0.84	0.822
OpenOffice	0.95	0.87	0.89	0.88
Apache Thrift	0.95	0.71	0.86	0.78
Linux Kernel	0.9	0.96	0.92	0.94
Mozilla Core + JSS	0.95	0.80	0.82	0.81
Overall	0.9	0.84	0.85	0.84
Overall (no clustering)	n/a	0.67	0.84	0.74

4. RELATED WORK

Various techniques have been presented to deal with the separation of *natural language text* and *technical information*.

Bettenburg et al. present two techniques to categorize *natural language text* and *technical information*. The first approach [4] is the most similar to ours and also includes text heuristics. They first use a spell-checker to identify

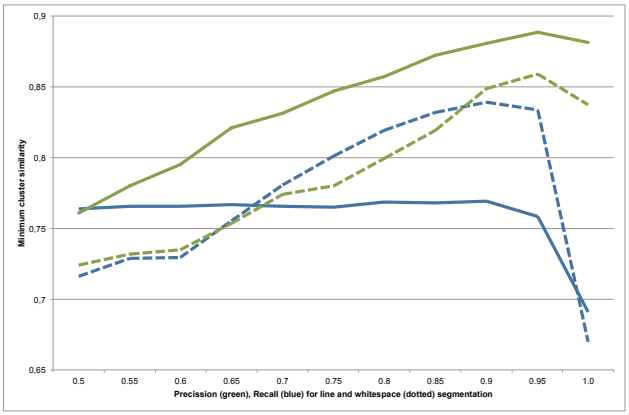

Figure 3: Precission and recall for different cluster similarities.

wrongly written words and treat them as *technical information*. Afterwards they validate the findings of the spell-checker by applying heuristics. Their second approach uses Island Grammars to identify patches, stack traces, source code and enumerations [5]. The outcome of this approach, however, is a binary classification, since it only checks whether code, stack traces, patches or enumerations are present in the document or not. In contrast, our approach classifies each token.

Bacchelli et. al introduced an approach that combines parser-based techniques with a Naive Bayes classifier as a term-based technique [2] to classify documents on line level in natural language, stack traces, patches, code and a class they call junk[8]. Though their approach performs very well it relies on a training dataset for Naive Bayes. Their approach is based on earlier work [3, 1].

Cerulo et al. address the problem using Hidden Markov Models, which train directly from the data. Their approach does not require any manual tuning or the definition of e.g. parsers or regular expressions and tries to learn directly from the data [6].

5. CONCLUSION & FUTURE WORK

The paper presents a heuristic and clustering-based approach to classify document content as *natural language text* or *technical information*. The approach can either be used on its own to extract *technical information* from software repositories or it can be combined with other text mining algorithms as a preprocessing step. Our approach is the first, to use hierarchical clustering to improve text classification.

The approach was evaluated on 225 documents on the basis of line and white space tokenization resulting in $F_1 = 0.82$ for line and $F_1 = 0.84$ for white space tokenization. The improvement from clustering was most noticeable for white space tokenization. We sampled documents from multiple projects with different programming languages for evaluation and did not change any parameters of the heuristics during evaluation, to show that performance is good even if the algorithm is not adjusted for a certain project or situation. That said, adjusting parameters or adding additional heuristics to match a concrete mining problem could improve the results.

[8]includes non-valuable information, such as signatures

In current and future work, we will evaluate a more detailed classification that splits *technical information* into the classes *code*, *stack traces or log files* and *patches* and evaluate the influence of clustering for these classes. Then, the heuristics and the clustering algorithm should consider a confidence weighting for each class. Finally, since the approach can be used as preprocessing for other, especially trained, algorithms such as Naive Bayes or Support Vector Machines to 'clean up' noisy data, we will experiment on the performance of such classifiers with and without the presented approach as preprocessing.

6. ACKNOWLEDGMENTS

This work is partly funded by the Bonn-Rhein-Sieg University of Applied Sciences Graduate Institute.

7. REFERENCES

[1] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci. Extracting structured data from natural language documents with island parsing. In *2011 26th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE 2011)*, pages 476–479. IEEE, Nov. 2011.

[2] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. *2012 34th Intl. Conference on Software Engineering (ICSE)*, pages 375–385, June 2012.

[3] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting Source Code from E-Mails. *2010 IEEE 18th Intl. Conference on Program Comprehension*, pages 24–33, June 2010.

[4] N. Bettenburg, B. Adams, A. E. Hassan, and M. Smidt. A Lightweight Approach to Uncover Technical Artifacts in Unstructured Data. *2011 IEEE 19th Intl. Conference on Program Comprehension*, pages 185–188, June 2011.

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In *Proceedings of the 2008 Intl. Workshop on Mining Software Repositories*, page 27, New York, New York, USA, 2008. ACM Press.

[6] L. Cerulo, M. Ceccarelli, M. Di Penta, and G. Canfora. A Hidden Markov Model to detect coded information islands in free text. *2013 IEEE 13th Intl. Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, Sept. 2013.

[7] R. Feldman and J. Sanger. *The Text Mining Handbook*. Cambridge University Press, 2006.

[8] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2011.

[9] K. Herzig, S. Just, and A. Zeller. It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction. In *Proceedings of the 2013 Intl. Conference on Software Engineering (ISCE)*, pages 392–401. IEEE Press, 2013.

[10] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press. Cambridge, MA, 1999.

[11] I. Witten, E. Frank, and M. A. Hall. *Data Mining*. Morgan Kauffmann, 2000.