Tracing Your Maintenance Work – A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages

Andreas Mauczka, Markus Huber Christian Schanes, Wolfgang Schramm, Mario Bernhart, and Thomas Grechenig

Research Group for Industrial Software, Vienna University of Technology
Vienna 1040, Austria
{andreas.mauczka,markus.huber,christian.schanes,wolfgang.schramm, mario.bernhart,thomas.grechenig}@inso.tuwien.ac.at
http://www.inso.tuwien.ac.at/

Abstract. A commit message is a description of a change in a Version Control System (VCS). Besides the actual description of the change, it can also serve as an indicator for the purpose of the change, e.g. a change to refactor code might be accompanied by a commit message in the form of "Refactored class XY to improve readability". We would label the change in our example a perfective change, according to maintenance literature. This simplified example shows how it is possible to classify a change by its commit message. However, commit messages are unstructured, textual data and efforts to automatically label changes into categories like perfective have only been applied to a small set of projects within the same company or the same community. In this work, we present a cross-project evaluated and valid mapping of changes to the code base and their purpose that is usable without any customization on any open-source project. We provide further the Eclipse Plug-In Subcat which allows for a comfortable analysis of projects from within Eclipse. By using Subcat, we are able to automatically assess if a commit to the code was e.g. a bug fix or a refactoring. This information is very useful for e.g. developer profiling or locating bad smells in modules.

1 Introduction

Software is constantly evolving. Leading and monitoring software development projects is a difficult task and performance indicators become mandatory for deciding on a course of action, e.g. is now the time to refactor some of my code or do I need to intensify my quality assurance work, because my development team spends a majority of their time troubleshooting and bug fixing. Managers or project leads need to be well informed to enhance their decision making process and to have an accurate view of the current state of the project. By gathering the information that is actually available in the form of meta data in the Version Control System (VCS), conclusions about the software development and maintenance (e.g. which modules are error prone, which modules have not been refactored recently) can be drawn. Since our approach does not rely on the code itself, it can be applied to any programming language and early in the software life cycles, when code metrics might not be conclusive yet.

In the following paper we use meta data which can be mined from a VCS that uses commit messages to accompany any change (a commit) to the code base. From the textual information in these commit messages, we mine information about the software maintenance and evolution process in open source projects. We base our work on the assumption that commit messages hold information that should give evidence of the purpose of the source code change (see Section 5).

We present in this paper two contributions to the analysis of meta data in a VCS. First, we implemented the tool "Subcat" to classify commit messages based on a set of keywords (we refer to the master set of all keywords as a dictionary). Subcat is a generic analysis tool that can be configured to classify any commit message into arbitrary categories based on a dictionary. It further is possible to customize Subcat to fit any personal project vocabulary by adding categories or keywords to the dictionary.

Subcat provides different kinds of reports (categorization per file or per module) to visualize the results of this categorization. Subcat also generates statistics on the authors of the commit messages or statistics on the words used in the commit messages. We provide Subcat as an Eclipse plugin for integrated analysis by developers or project managers during early software lifecycle stages in the Eclipse IDE and as a standalone command line tool for the mining of large scale software projects (see Section 2).

Second, we used the reports generated by Subcat to create an optimized and cross-project valid dictionary that allowed us to automatically classify commits into Swanson's maintenance categories [10] for the open source domain. To achieve this, we defined an algorithm to incrementally train and improve this dictionary with certain keywords. After training the dictionary on a number of projects, we evaluated this dictionary against a larger set of open source projects (see Section 3 for the algorithm and Section 4 for the results of the evaluation).

Subcat can be used in two different contexts. Subcat can be used by practitioners in the open-source domain to analyze modules based on maintenance characteristics. E.g. when a module has a lot of corrective changes, but no perfective changes, some refactoring of the code might be in order (see Figure 1. Furthermore, Subcat provides Maintenance Profiles of the development team. This means that one can see at first glance, whether a developer is mainly fixing bugs or keeping the code clean.

Subcat can also be used by researchers. By using Subcat's corrective classification mechanism, we are able to track bugs within the repository additionally to a normal bug tracker. This allows for research on the difference of bug granularity in repositories and bug trackers like bugzilla (a bug fix in the repository may not correspond to a bug report in the tracker). Additionally, we can use Subcat to analyze how developer profiles change over time in a project (E.g. a developer starts in a project to fix bugs that annoy him and ends up implementing a whole new feature see Figure 2 for an example). Subcat provides this

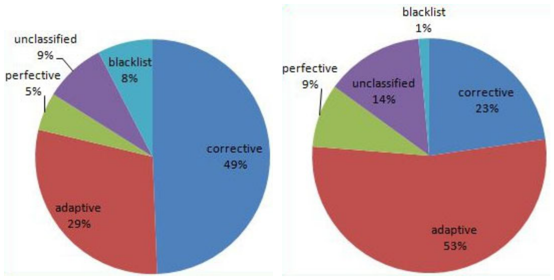
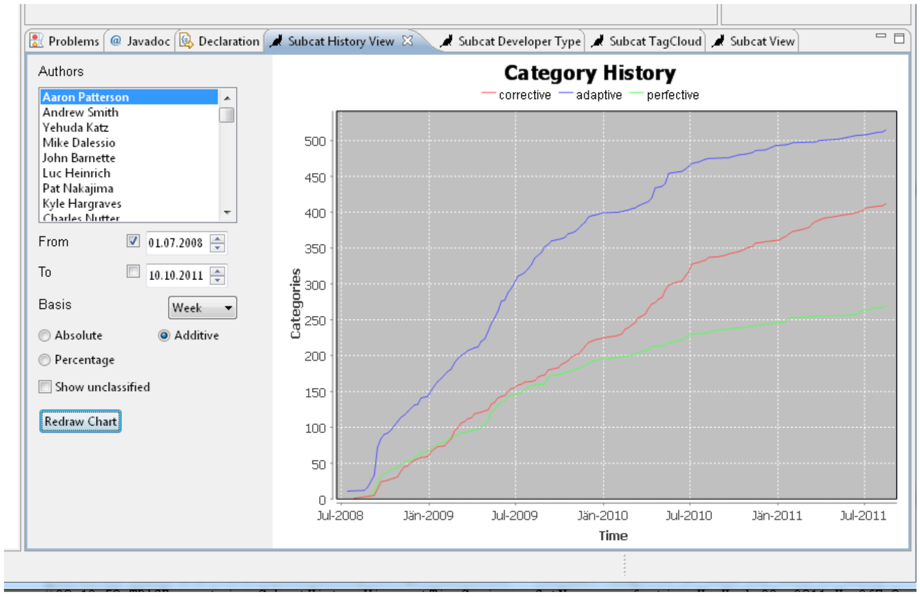


Fig. 1. Visualization of the classified activities in two different software modules

information, which can be combined with mailing list analysis. This can provide a whole new insight into how a developer changes over time in an open-source project.



Authors
Aaron Patterson
Andrew Smith
Yehuda Katz
Mike Dalessio
John Barnette
Luc Heinrich
Pat Nakajima
Kyle Hargraves
Charles Nutter

From [✓] 01.07.2008
To [ ] 10.10.2011
Basis Week
Absolute [✓ Additive]
Percentage
[ ] Show unclassified

Fig. 2. Developer Profile in Subcat

2 Automated Classification Approach

A dictionary, as used in the context of this paper, is a set of categories. A category is a group of keywords that share a common meaning and therefore are indicators for this category, e.g. the word ”fix” is a keyword for the maintenance category ”corrective”. In the context of this work, we apply Swanson’s maintenance categories to group our keywords.

We propose the following procedure to create a dictionary:

Pre-Processing the Meta Information. The meta information for our analysis was derived from the commit messages in the VCS. As these messages are written in natural language, we have to normalize them to be able to extract sensible information (e.g. we want to match "this **fixes** a re-ocurring crash" and "I **fixed** an overflow" to its lemma "fix" - a head word under which the word would be found in a dictionary). We use WordNet[1] to normalize the commit messages

Initializing the Dictionary. We generate an initial seed for a dictionary by referring to prior work (Mockus and Votta in [9] and Hassan in [5]). This initial seed only contains words that hold a high likelihood of indicating a maintenance category

Training the Dictionary. To be able to categorize as many changes as possible with a high accuracy for a single project, we use a defined algorithm to train the dictionary. We employ the algorithm to train the dictionary on additional open source projects to further increase the accuracy of the dictionary (see Section 3)

Evaluating the Dictionary. After the initial training, we use the dictionary on another set of projects to evaluate cross-project validity. We do not further change the dictionary during this step. Only blacklist items (keywords that filter out administrative changes) are introduced (see Section 4)

2.1 Classification Rules

The research area of the identification and classification of maintenance tasks in the software development process has evolved for decades. In [10], Swanson defines a maintenance task as an activity that can be assigned to one of the following three categories:

Corrective Software Maintenance. Activities that are necessary to fix processing failures, performance failures or implementation failures

Adaptive Software Maintenance. Activities that focus on changes in the data environment or changes in the processing environment

Perfective Software Maintenance. Activities that strive to decrease processing inefficiency, enhance the performance or increase the maintainability

For the development of the automated classification in this work, Swanson's original definition of maintenance tasks is used and slightly extended. An additional category, the "Blacklist" is introduced. We use the Blacklist to filter all commits, which underlying modifications were not carried out by humans or which do not actually include any source code modifications. For example commits generated by the "cvs2svn" repository-converter[2] or commits that just "tag" a version. In addition we merged the implementation category, as presented by Hindle et al.[6]

[1] `http://wordnet.princeton.edu/`
[2] `http://cvs2svn.tigris.org/`

with Swanson's adaptive maintenance category. As a result we are able to map every commit to exactly one category. Using Swanson's original maintenance classification provides a categorization into a few, well defined categories and is therefore a suitable starting point to develop an automated classification algorithm.

As mentioned above, our algorithm relies on two sources of information to carry out the classification, namely the commit message and the dictionary. The **commit message** is attached to every commit and encapsulates the information about the intention of the modification. The **dictionary** defines the knowledge base for the classification including the categories. The different categories are defined by a set of keywords that indicate that a commit message may belong to this category. In addition, every word has an associated weight. The weight value constitutes how strong the indication is. The same word can be contained in multiple categories. See Figure 3 for a sample dictionary that is used to classify a commit message.

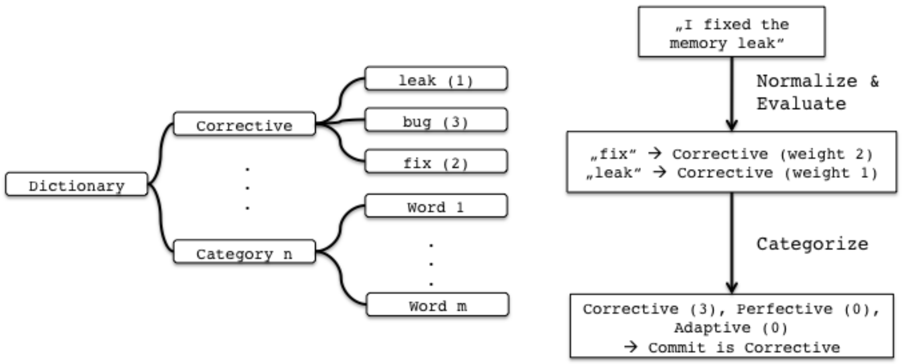

Dictionary
Corrective
leak (1)
bug (3)
fix (2)
Category n
Word 1
Word m
„I fixed the memory leak"
Normalize & Evaluate
„fix" → Corrective (weight 2)
„leak" → Corrective (weight 1)
Categorize
Corrective (3), Perfective (0), Adaptive (0)
→ Commit is Corrective

Fig. 3. Example for Dictionary and Classification

To implement the blacklist feature, "absolute categories" have been introduced. If a commit message contains a word (e.g. "cvs2svn") that is included in the listing of an absolute category, the commit is instantly assigned to this category, ignoring the weighting mechanism and the normal categories.

2.2 Categorization Tool - Subcat

Subcat is a tool implemented to generically categorize commit messages based on their content. It consists of two parts, the command line tool Sublex and the Eclipse plugin, which we describe in detail in the following sections.

Sublex is the tool that implements all relevant functionality to classify commits from a generic data source. Due to the modular design of the command line tool different Versioning Systems are supported, if adapters for pre-processing a logfile to the generic data source format are available. The adapter for Subversion is supplied together with Sublex and ships also as a part of the Eclipse plugin.

The results of the classification are reports in the CSV-format. Sublex offers the following reports:

Categorization-Report. The categorization report contains all commits and their corresponding classification results in detail. It is the base for the detail reports that follow. It can be used by analysts to generate their own statistics based on the report data. The displayed information per row are: commit including the revision, the category it has been assigned to, the author, the date of the change, the length of its commit message, the overall number of added and deleted lines for the commit, the score of the commit for each category from the dictionary, the affected modules, the affected files and the revised commit message.

Author-Report. The author report shows the analysis of commits (including the assigned maintenance categories) per author. Its purpose is to analyze the profiles of developers in the project. For example if an author is responsible for perfective maintenance or if perfective maintenance is distributed evenly on the team.

Dictionary-Report. This report provides required information to create and improve dictionaries by showing statistical information for every unique word found in any of the parsed commit messages. The report provides the lemma for the word, the average number of appearances of the word in the commit messages it was found in, the total number of appearances in all the commit messages, the total number of classified and unclassified commits the word was found in

Lemma-Report. The lemma-report is the second required report for creating and improving dictionaries. It includes an entry for every unique lemma, together with the number of classified and unclassified commits the lemma was found in

Modules-Report. This report shows categorization statistics about the modules of a project. Module structure to be analyzed can be parametrized. E.g. the project has the structure of /util/login/security. We configure a module depth of 2. There will be a row for /util/* and one row for util/login/* in the report

Control-Report. This report was used to manually validate the analysis result during our research. It contains every original commit message and the category it was assigned to

Eclipse Plugin. The Eclipse plugin has been implemented to integrate Subcat into the Eclipse IDE to give analysts and developers, but also project managers, a familiar environment for maintenance analysis. The integration into Eclipse allows a comfortable comparison between our reports and any other metric suites a user might employ. E.g. a project manager can view results for code metrics next to the results of the categorization of the commits of a module and use both of these views in his decision making process. Furthermore, the usability of Subcat is improved by using the point and click paradigm to generate reports, e.g. for authors and modules, as a user can navigate through the proper views in the IDE (see Figure 2).

The Eclipse plugin uses the generic data source adapter for Subversion and the classification functionality and the logic from the command line tool Sublex. Due to its generic approach the classification functionality can be used without adaptations in a different context. The complete plugin project is split into three individual plugins. The generic data source adapter and Sublex and the main plugin which integrates the categorization functionality into the Eclipse workbench. This corresponds to the MVC design pattern (see Buschmann et al. [2]).

3 Generation of a Cross-Project Valid Dictionary

To build a representative dictionary, a set of projects to provide initial keywords and to train our dictionary are required. We further need another set of projects to ensure cross-project validity of the dictionary.

3.1 Criteria and Selection of Open Source Projects

Eight open source projects were chosen to build, test and cross verify the dictionary. The following criteria were used to select the projects:

Number of Commits. For our analysis we only considered projects with at least 30,000 commits[3]

Number of Developers. To show the categorization of the developer role in a project and also to increase the variance of different commit message style only projects with at least 30 developers[4] are considered.

Subversion Repository. Our approach is currently based on Subversion repositories. Therefore only projects with access to their Subversion repositories are included.

Table 1 shows the key figures of the selected projects.

3.2 Populating the Dictionary

As a starting point to create the dictionary we analyzed the log of the FreeBSD-Project and used exemplary keywords from prior work (see [5] and [9]) for the categorization. In the next step we ranked the keywords by occurrence. The top three ranked keywords of each category are included in the first dictionary:

[3] The number of commits is the number of commits in the log.

[4] The number of developers represents the number of distinct author names in the log.

Table 1. Key figures of the analyzed open source projects

App. Name	App. Type	# Devs	# Commits
Boost	Prog. Library	294	63,616
Enlightenment	Window Manager	187	51,884
Evolution	E-Mail-Client	431	37,500
FreeBSD	OS	536	150,595
Firebird	RDBMS	43	51,509
GCC	Compiler-Suite	426	102,672
Python	Interpreter	216	83,100
Wireshark	Packet Analyzer	43	34,067

Corrective: fix, bug, problem
Adaptive: new, change, patch
Perfective: style, move, removal

This first dictionary constituted the "seed" to create a more exhaustive dictionary. This initial dictionary only categorized a low number of commits, leaving a large number of commits uncategorized. Starting with this seed, we set up an algorithm with the goal to increase the ratio of classified commits to 80% while maintaining adequate values for a self-evaluated precision (0.8) and recall (0.8). Values beyond these thresholds yield diminishing results - either less commits will be classified, or precision and recall will suffer. An early attempt at the algorithm had to be abandoned, because of a too conservative approach in adding words to the dictionary (stagnation at about 65% of categorized commits). For the final algorithm we used a more open and flexible approach so that more words would qualify for the dictionary. We further introduced weighting of keywords and rulesets for ambiguous, yet strongly indicative words. The following list describes step wise our final algorithm to create the dictionary:

1. Classify the commit using the "seed" dictionary
2. If the total percentage of classified commits is greater than 80%, EXIT
3. Count the appearances of all words in the commit messages of the non-classified commits and order them by frequency
4. Choose a set of words from the top of the list and add these as a test set to the existing dictionary
5. Count the number of appearances of every word in the test set in each category
6. If the number of appearances of a word in a category is at least 1.5 times of the appearances of the same word in the other categories, add it to the dictionary with a weight of 2 and remove it from the test set
7. If the number of appearances of a word in two classes is at least 1.5 times of the appearances of the same word in the third class, add it to the dictionary to both classes with a weight of 1 and remove it from the test set
8. If neither 6 or 7 are true, remove the word from the test set and do not add it to the dictionary
9. Go to Step 2

This algorithm achieved a classification rate of 80.34 % after 21 iterations on the FreeBSD project. The output is the following dictionary (weights of keywords in brackets, default weight 1).

Corrective: active, against, already, bad, block, bug, build, call, case, catch, cause(2), character, compile, correctly, create, different, dump, error(2), except, exist, explicitly, fail, failure(2), fast, fix(2), format, good, hack, hard, help, init, instead, introduce, issue, lock, log, logic, look, merge, miss(2), null(2), oops(2), operation, operations, pass, previous, previously, probably, problem, properly, random, recent, request, reset, review, run, safe, set, similar, simplify, special, test, think, try, turn, valid, wait, warn(2), warning, wrong(2)

Adaptive: active, add(2), additional(2), against, already, appropriate(2), available(2), bad, behavior, block, build, call, case, catch, change(2), character, compatibility(2), compile, config(2), configuration(2), context(2), correctly, create, currently(2), default(2), different, documentation(2), dump, easier(2), except, exist, explicitly, fail, fast, feature(2), format, future(2), good, hack, hard, header, help, include, information(2), init, inline, install(2), instead, internal(2), introduce, issue, lock, log, logic, look, merge, method(2), necessary(2), new (2), old(2), operation, operations, pass, patch(2), previous, previously, probably, properly, protocol(2) provide(2), random, recent, release(2), replace(2) ,request, require(2), reset, review, run, safe, security(2), set, similar, simple(2), simplify, special, structure(2), switch(2), test, text(2), think, trunk(2), try, turn, useful(2), user(2), valid, version(2), wait

Perfective: cleanup(2), consistent(2), declaration(2), definition(2), header, include, inline, move(2), prototype(2), removal(2), static(2), style(2), unused(2), variable(2), warning, whitespace(2)

Blacklist: cvs2svn, cvs, svn

The analysis further showed that the word "documentation" was assigned to the adaptive category by the algorithm. Since "documentation" is a perfective task per definition, the word "documentation" was moved from **adaptive** back to **perfective**. The implications warrant further research however.

This final dictionary was used to classify the FreeBSD-project again and precision and recall were measured based on modification records (MR) as shown in Table 2.

Table 2. Recall and precision of the classification for the FreeBSD-project

Class	MR	%	Recall	Precision
Corrective	54,015	35.86%	0.92	0.85
Adaptive	56,046	37.21%	0.91	0.80
Perfective	8,484	5.63%	0.86	0.80

We then used the dictionary and the algorithm on the "Boost" project (inital classification rate 74.94%), thereafter on "Enlightenment" project (initial classfication rate 72.80%) and altered the dictionary until it achieved 80%

of classified changes. We decided to train the dictionary on two other projects to achieve a greater classification ratio and to work out project-individual language issues (e.g. ambiguously connotated lemmas). After this training phase, the dictionary was used with the "Evolution", "Firebird", "GCC", "Python" and "Wireshark" projects and scored a classification rate of over 80% for each project, without adaption.

Table 3. Recall and precision of the analysis for various open source projects

Project	# MR	Recall	Precision
Enlightenment	51,884	0.90	0.80
Evolution	37,500	0.96	0.92
Firebird	51,509	0.95	0.90
GCC	102,672	0.92	0.83
Python	83,100	0.93	0.85
Wireshark	34,067	0.92	0.85
FreeBSD	150,595	0.90	0.82
Boost	63,616	0.94	0.88

4 Evaluation of the Dictionary

To evaluate our results, we did a survey with five professional Software Developers. The developers are working for different companies since between two to five years (2,2,4,4 and 5). Our survey was structured as follows:

- Five questionnaires, each with the 21 changes in the code (7 of each category).
- Five questionnaires, each with the same changes in the code, but with their corresponding commit messages

4.1 Inter-rater Agreement

To measure inter-rater Agreement of the developers, we used Fleiss' Kappa on six commits that were identical in each questionnaire. Table 4 shows the agreement amongst the developers for these six commits (two commits per category). The resulting Fleiss' Kappa for this matrix is $\boldsymbol{K = 0.48}$. This indicates a **moderate agreement** according to Landis and Koch's Benchmark [8] between the developers themselves.

If a commit is assigned to two categories, its count is split between the categories.

4.2 Conducting the Evaluation

We conducted the survey in two rounds. Table 5 and Table 6 show the agreement between developers and the automated classification tool. If a developer chose two categories, a point was split between these categories.

Table 4. Matrix showing the agreements amongst the developers for the six common commits in evaluation round two

Commit/Category	**Adap.**	**Corr.**	**Perf.**
Corr. 1	1.0	4.0	0.0
Corr. 2	0.5	4.5	0.0
Perf. 1	1.0	2.0	2.0
Perf. 2	0.0	0.0	5.0
Adap. 1	4.5	0.0	0.5
Adap. 2	4.0	0.0	1.0

Table 5. Agreements between developers and classification tool for the evaluation round one

	Automated Classification			
Developers	**Adap.**	**Corr.**	**Perf.**	
Adaptive	**11.0**	4.5	0.5	16.0
Corrective	4.5	**12.0**	0.5	17.0
Perfective	7.5	8.5	**24.0**	40.0
	23.0	25.0	25.0	**47.0**

Table 7 shows the summarized results of the evaluation rounds one and two. The columns show the total number of agreements between the developers and the automated classifications for each category and the Cohen's Kappa-value.

4.3 Interpretation of the Evaluation

The following conclusions can be drawn from these results:

- Both the agreements in the adaptive category as well as the agreements in the perfective category stayed constant for both rounds. In contrast, the number of agreements for the corrective category has significantly risen between round one and two. From this fact we conclude that corrective maintenance tasks are most difficult to spot just by looking at the source code and without reading the commit message
- The number of agreements for the perfective category is almost perfect in both rounds. We therefore conclude that our classification tool excels at identifying perfective maintenance tasks (a finding similar to Mockus et al's inspection change finding in [9])
- The Kappa-value has risen from 0.46 to 0.61 from round one to round two. This means that with the additional information of the commit message, the developers have converged their decisions with the decisions of the automated classification. Curiously this affected mainly corrective changes
- 0.46 and 0.61 both indicate a **moderate agreement** according to the El Emam Benchmark - see “SPICE Software Process Assessment Kappa benchmark” as introduced in[3]

Table 6. Agreements between developers and classification tool for the evaluation round two

	Automated Classification			
Developers	**Adap.**	**Corr.**	**Perf.**	
Adaptive	**12.0**	1.5	0.0	13.5
Corrective	3.5	**19.0**	1.0	23.5
Perfective	8.5	4.5	**24.0**	37.0
	24.0	25.0	25.0	**55.0**

Table 7. Comparison of evaluation rounds one and two

	Agreement			
Round	**Adap.**	**Corr.**	**Perf.**	**Kappa**
Round 1	11	12	24	0.46
Round 2	12	19	24	0.61

5 Related Work

In 2000 Mockus and Votta presented a study [9] that followed an approach similar to this work. They propose the importance of a textual description to understand the reasons behind software changes. The evaluation performed in our work strongly suggests the truth of that statement. They further state that other factors might also influence the change classification. We aim to find new classification rules to improve the classification algorithm. The classification algorithm and the dictionary that we developed solely focus on the "textual description field" but our implemented tool Subcat was built already keeping in mind an extension of the classification algorithm also involving other aspects, such as size of the commit, measured in changed lines of code, or interval.

In 2008 German and Hindle released a study about the taxonomy of large commits [6]. They define large commits as commits that include a large number of files. In their study, they manually classified large commits from nine open source projects by their intentions. They started by extending Swanson's categories by the categories "implementation" and "non functional". During their work they observed that these categories did not suffice for the categorization of the intention of large commits and developed a new set of categories which they call the "Categories of Large Commits". In [7] , Kemerer and Slaughter presented a set of methods and techniques to study software evolution. Bevan et al. introduced a system called Kenyon [1]. They imply resource intensive logistical constraints, e.g. the extraction of analysis specific facts, the storage of the results of the extraction. These tasks have to be performed for each change or configuration separately. Kenyon is designed to support these logistical tasks. It provides support for different software configuration management systems and retrieves consistent source code configurations. This issue is solved by implementing different plugins for every software configuration management system.

The plugins themselves are very lightweight because they can all utilize the same libraries, that hold the core functionalities.

A good overview and description of existing change metrics for commits and their different subtypes is provided by German et al. in [4]. They also presented a framework for the classification of change metrics. They present five different groups of change metrics: entity change metrics, MR-scoped change metrics, time-based change metrics, event-triggered change metrics and change metrics that do not measure code. Additionally they divide the change metrics into modification-unaware and modification-aware metrics. The classification presented in this paper can be the basis for the computation of related change metrics. The software that we present, in a first step is only capable of classifying changes using the classification algorithm. In the future the software can be extended in a way that it automatically calculates some of the metrics presented by [4].

6 Conclusion

The presented work provides a tool, Subcat, and a dictionary for cross-project analysis of software evolution based on an automated classification. To achieve these two goals, we completed the following tasks:

Classification Algorithm. We developed a classification algorithm which uses a dictionary as its base of decision-making. The classification algorithm uses a set of commits as its input and returns an assignment between the commits and the categories that are defined in the dictionary. It is based on the analysis of the natural language in the commit messages and follows a lexical approach.

Dictionary. We presented a dictionary for our classification algorithm that is capable of assigning commits to Swanson's maintenance categories. The cross project validity of the dictionary has been proven on five different open source software projects. We instantly reached a percentage of successfully classified commits of over 80% for each of the projects, without having to adopt the dictionary.

Evaluation. We evaluated the dictionary and the automated classification by using a two-step evaluation process. In a first step we evaluated the decisions of the automated classification and the dictionary against our own manual classification. We reached an average recall of **0.93** and an average precision of **0.86**. In the second evaluation step we evaluated the dictionary against the opinion of five professional software developers. We have proven a **moderate agreement** between the decisions of the automated classification and the decisions of the developers. This result is similar to the result achieved by Mockus et al. in [9] and proves that the approach presented is valid for cross-project analysis in the open source project landscape.

Subcat. We developed a command line tool, which implements the classification process. We defined a generic input format for the commits, to ensure the reusability for data, extracted from different VCS. The command line tool delivers the results of the classification as CSV-based reports. We presented an

Eclipse plugin which integrates the automated classification directly into the Eclipse workbench. It provides easy access to the classification functionality and includes a rudimentary visualization of the results of the classification.

The successful evaluation of the lexical-approach on generic open source projects has many implications. Researchers can use Subcat for a new definition of maintenance and software evolution metrics, not only in open-source projects, but also in any project using non-obscure commit messages. Or fellow researchers can use Subcat to comfortably analyze for example developer profiles over time in open source projects, e.g. which developer does the bug fixing, who is implementing the new features. Subcat can not only be used for profiling or software evolution metrics though. Additionally, Subcat has been adapted to work with GIT repositories recently.

Additionally to the main purpose, the categorization of changes into software maintenance categories, Subcat can be easily adapted (by changing the dictionary) for any other studies on commit messages in repositories. The author and dictionary report and to some extent the lemma report are especially useful for this purpose.

6.1 Discussion

There are many extensions to Swanson's classification of maintenance tasks. In our work, we adhere to Swanson's original category set, because it is manageable and well defined. During our research we studied a lot of commits and recognized that often, one commit holds multiple, non-associated changes that would have to be assigned to different categories. This indication warrants further research. In the survey based evaluation we used Cohen's Kappa as a measure. The definition of a nominal scale implies that every item can be classified to exactly one category. As indicated earlier, there are cases where a commit can not be distinctly classified to one category but includes different activities that stretch between two or even all of the three maintenance categories.

For detailed results per project, please contact the authors at the given mail address.

6.2 Future Work

In this paper we use WordNet solely for matching words and lemmas. WordNet also possesses the possibility for cognitive matching which could be included in the matching algorithm. Furthermore, Subcat is capable of measuring further details of the commits (e.g. commit size, length of commit messages). These parameters provide possibilities for further tuning of the categorization mechanism.

References

1. Bevan, J., Whitehead Jr., E.J., Kim, S., Godfrey, M.: Facilitating software evolution research with kenyon. In: Proceedings of the 10th European Software Engineering Conference, pp. 177–186 (2005)

2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns, vol. 1. John Wiley and Sons (1996)
3. Emam, K.E.: Benchmarking kappa for software process assessment reliability studies. Empirical Software Engineering 4, 113–133 (1999)
4. German, D.M., Hindle, A.: Measuring fine-grained change in software: Towards modification-aware change metrics. In: Proceedings of the 11th IEEE International Software Metrics Symposium, p. 28 (2005)
5. Hassan, A.E.: Automated classification of change messages in open source projects. In: Proceedings of the 2008 ACM Symposium on Applied Computing, pp. 837–841 (2008)
6. Hindle, A., German, D.M., Holt, R.: What do large commits tell us? a taxonomical study of large commits. In: Proceedings of the International Working Conference on Mining Software Repositories, pp. 99–108 (2008)
7. Kemerer, C.F., Slaughter, S.: An empirical approach to studying software evolution. IEEE Transactions on Software Engineering 25 (1999)
8. Landis, J.R., Koch, G.G.: The measurement of observer agreement for categorial data. Biometrics 33, 159–174 (1977)
9. Mockus, A., Votta, L.G.: Identifying reasons for software changes using historic databases. In: Proceedings of the International Conference on Software Engineering, pp. 120–130 (2000)
10. Swanson, E.B.: The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering, ICSE 1976, pp. 492–497 (1976)