Examining the Relationship between Topic Model Similarity and Software Maintenance

Scott Grant James R. Cordy
School of Computing, Queen's University, Kingston, Ontario, Canada
{scott,cordy}@cs.queensu.ca

***Abstract*—Software maintenance is the last phase of software development, and typically one of the most time-consuming. One reason for this is the difficulty in finding related source code fragments. A high-level understanding of the source code is necessary to make decisions about which source code fragments should be modified together, for example, in the context of fixing a bug. Even with a similarity metric available, understanding what it means to measure similarity in the first place is important; if a technique suggests that two source code fragments are related, is there a human-oriented way of explaining that relation? In this paper, we attempt to identify a concrete link between software maintenance and the similarity metrics provided by latent topic models. We show that similarity in topic models is related to the likelihood that source code fragments will be modified together in the future, and that an awareness of similar source code can make software maintenance easier.**

I. Introduction

Software development tends to be dominated by maintenance [12]. One difficult problem in software maintenance involves predicting other source code fragments that should be considered when making a change. One approach to solving this problem involves tracking the maintenance history of code sections and assuming that code that has been changed together in the past may need to be changed together in the future. This approach has been shown to make good suggestions [8], [9], leading to the possibility of using co-maintenance history as an evaluative source of data. While it is possible to observe past co-maintenance by observing the changelists in the history of a project, making meaningful predictions for the future often requires a long history.

In this paper, we show that Latent Dirichlet Allocation, an unsupervised latent topic model, can be effective at predicting required changes. We demonstrate this fact by artificially omitting source code fragments from clusters of historically co-maintained fragments to simulate a forgotten change. By choosing arbitrary points in the project's revision history and generating topic models based on that version of the source code, as seen in Figure 1, we use the observed future changelists to evaluate the model's ability to predict co-maintenance. By basing our experiment on actual maintenance history, we show that in many cases, topic models are able to predict co-maintenance relationships without supervision. In essence, we can evaluate how well they predict what else we might have forgotten to change when making a revision.

II. Background

Co-maintenance is an observable property of software systems under source control, in which source code fragments are modified together within some time frame. The most familiar representation of co-maintenance is the *changelist*, a set of several source code changes grouped together to represent a more complete view of a change [7]. For example, implementing a feature may involve many individual changes across several functions or files. A changelist groups these individual changes together, and each changelist in the history of the project can be considered to define a set of co-maintained code fragments.

Statistical models are one approach for identifying relationships between documents in terms of their "variables". In natural language processing, variables generally refer to the presence or absence of words, or "terms" in a collection of input documents. For example, in a corpus built from a mixture of ten thousand different terms, a statistical model with ten-thousand variables can be constructed to evaluate how the documents relate to each other. In some models, this yields a set of latent "topics", each of which is characterized by details about how the terms co-occur. This idea has been adopted in source code analysis, where source code fragments such as methods, functions, or classes are used in place of the natural language documents to identify related code fragments [10].

Latent Dirichlet Allocation (LDA) is one such topic model that defines a generative prior topic distribution over the documents [1]. This means that each document is considered to have been generated from a mixture of latent unobserved topics, where the probability distributions of each topic are assumed to have a Dirichlet distribution. In this way, each document is represented in the model as a bag-of-words consisting of the individual terms from the original document, and each

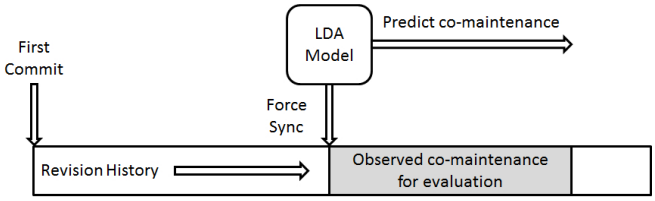

Fig. 1. A visualization of the process used to evaluate a model's predictive ability. We clone the revision history of a project to a local repository. The repository is force synchronized to a particular date in the past, and an LDA model of the source code is generated using the functions from the system at that point in time as documents. Once the model has been created, the successive changelists from that point forward are used as an oracle for correct co-maintenance predictions. For changelists with two or more function modifications, we observe the effect of omitting one function from the list of modified functions and using the remaining functions to predict similarity across all other functions in the system. If the omitted function is in the top-n, then the model has predicted the forgotten function in the changelist.

978-1-4799-3752-3/14 © 2014 IEEE

CSMR-WCRE 2014, Antwerp, Belgium
Early Research Achievement

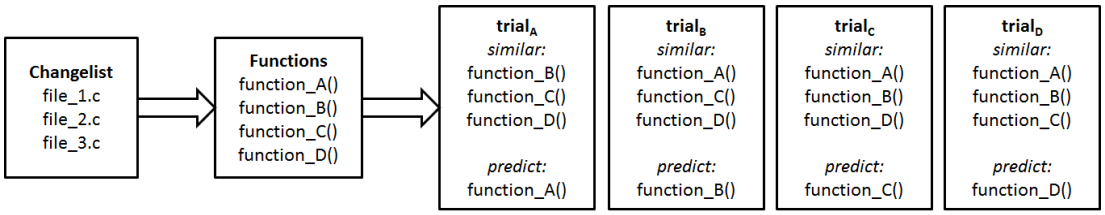

Fig. 2. The set of n functions modified in each changelist is identified. Removing each of the functions one-by-one produces n groups of function subsets and an omitted correct prediction. We refer to these as *prediction trials*, or simply *trials*.

```
commit 96d00b59d5c396b3af37de7a707980793ee14bbf
Author: Stefan Fritsch <sf@apache.org>
Date:   Mon Feb 27 21:45:18 2012 +0000

    Prevent listener thread from ever
    updating a worker's scoreboard slot

diff --git a/server/mpm/event/event.c
           b/server/mpm/event/event.c
index d3aba1e..d6c76f5 100644
--- a/server/mpm/event/event.c
+++ b/server/mpm/event/event.c
@@ -767,10 +767,6 @@ static int
    start_lingering_close(event_conn_state_t *cs)
```

Fig. 3. A sample changelist from Git's revision history for the *httpd* project. On the last given line (the change hunk starting with @@ and split to column-width), Git has attempted to identify the context of the change using regular expression handling. If the change occurs inside of a function, as in this example, the function name will be listed. These are extracted to identify which functions have been changed inside of each changelist.

document has an associated topic distribution identifying how the model categorizes this document in relation to the latent topics, and by association, in relation to the other documents in the data set.

III. Methodology

A. Mining Co-maintenance History

Each changelist holds information about modifications to the project, and therefore has the potential to contain a set of modified functions. For each changelist in the revision history, we generate partial subsets of the modified code fragments. For example, in a changelist that describes modifications made to four software functions, we can obtain four partial subsets of three functions, each associated with the removed fourth function, which is treated as a "correct prediction". This can be seen in Figure 2, where a changelist containing four functions produces four groups of three functions and an implicit correct prediction. In this way, we can use $\mathbf{trial}_A$ as an oracle to show whether or not a model can predict the co-maintenance relationship with *function_A()*. We refer to these groups of function subsets and an omitted correct prediction as *prediction trials*, or simply *trials*.

We cache a full log of the entire set of changelists using the *git log -p* command. This list of changelists and diffs is stored locally in a file and used in later stages of the process. The most relevant data gained using the *log* command is the list of commit ids, dates, and modified functions (if they exist). Figure 3 shows a portion of one changelist from the *httpd* revision history. Each commit id is a 40-character string associated with one particular changelist. The changelist also contains one or more change hunks, listing any additions or deletions to individual files under revision control. This range information begins with a double-at sign, and includes information about where the change occurs, and if possible, the semantic context in which the change was made. In this study, we look for information on this line indicating modifications inside of functions that are then used as input to the topic model.

B. Model Generation

To evaluate the predictive power of a topic model, we force sync the local repository to some date in the observed history of the project and generate a model of the software system from that point. To do this, we use the *git rev-list* command with the *–before* and *–max-count=1* parameters. This returns a single commit id before an arbitrary date. For example, to obtain a model accurate for January 1st, 2010, we use *–before 2010-01-01*. The commit id returned from this command is used with *git reset* to force a hard sync, followed by a *git clean*, ensuring the repository is as close to that point in history as possible.

In this study, the documents used as input to the model are the set of all functions in the system. We obtain this set of functions using the NiCad clone detection tool [13], which includes a function extractor written in TXL [2]. All functions of a non-trivial size (five lines or more) are extracted, stripped of comments, and stored in an XML file. This XML file is parsed and used to generate the set of interesting terms for each document (i.e., function). In our case, these are programming-language keywords and programmer-defined names. Since many programmer-defined names have internal structure, we separate such names into their component pieces if they have been built in one of the standard ways (for example, breaking at underscores) and count both the entire name and each of its sub-pieces as terms. For example, the term *get_attribute* would be represented by three terms: *get*, *attribute*, and *get_attribute*. Models are generated using GibbsLDA++ [11], an open-source implementation of LDA.

C. Extracting Prediction Trials from Maintenance History

After the model has been generated and the revision history is available, we identify future changes in the project, and examine the similarity between functions modified in those changes. Each changelist modifies zero or more functions, identified using the change hunks as shown in Figure 3. Every change hunk describes the code block in which a change was made, and each changelist describes the files where modifications are made. These change hunks are compared to the

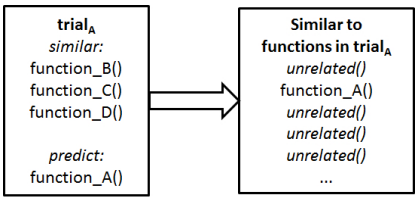

Fig. 4. The topic model is used to identify topic similarity between the functions in the changelist subset and the rest of the functions in the entire software system. This produces an ordered list of functions across the original system. We are interested in how often the left-out function is predicted by the topic model.

list of functions identified using the NiCad function extractor, and from this, we identify which functions represented in the model are modified in each changelist.

Function definitions can be modified over the life of a software project, including modifications to the parameters, return type, or name of a function. Functions can also be moved between files, for example in a refactoring. Since the function definition is provided in the change hunk of a Git changelist as indication of which function has been modified, we employ heuristics to attempt to track functions across such minor modifications in order to retain their identity. For example, if a function used in the generation of the latent model disappears after some time interval, but a new function in the same file exists, similar to the old one but with a different return type, we assume these functions are really versions of the same function.

By acquiring the set of functions represented in the model that are modified in each changelist, the trials shown in Figure 2 are generated and used to evaluate a model's ability to predict co-maintenance changes, as shown in Figure 4. The cosine similarity, a standard similarity metric (described in practice in [6]), is calculated between the remaining functions in a prediction trial and each of the other functions in the software system. For example, in a system with 100 functions, a changelist modifying four functions will result in four prediction trials of three functions each, and each prediction trial with three functions will be compared to 97 other functions. The desired result is that the left-out function will be near the top of a ranked similarity list of those 97 functions.

D. Evaluating Prediction Trials

For each prediction trial, the topic model is used to identify which individual code fragments are most similar to the other functions remaining in the changelist. These similar code fragments are interpreted as co-maintenance suggestions by the model. For example, if the model suggests that *function_A()* is most similar to the functions *function_B()*, *function_C()*, and *function_D()*, we suggest that the model can be interpreted to recommend *function_A()* as a potential co-maintained function. If the omitted function from the changelist is among the most similar functions suggested by the topic model, we argue that the topic model is able to predict co-maintenance relationships.

TABLE I
SYSTEMS USED IN THIS STUDY

System	Functions	Changelists	Topics
git	2989	20547	125
httpd	3509	23438	150
memcached	267	936	50
php-src	8169	70262	200

In Figure 4, one of the trials is used to identify a candidate list of similar functions. The **trial**$_A$ trial from Figure 2 has omitted *function_A()* from the original changelist. We estimate the similarity between the functions in the software system using the cosine distance between the vector representations of the associated topic distributions generated using LDA. The complete list of similar functions is determined by identifying the functions in the software system that are most similar to all the retained functions in **trial**$_A$.

The cumulative set of similarity scores between an arbitrary system function and the functions modified in a changelist is used to predict co-maintenance. For each function modelled in the system that is not already modified in the changelist, the sum of similarity scores between it and each modified function provides a changelist similarity. If a function is very similar to most or all of the functions in a changelist, it will have a high changelist similarity score, and may be a candidate for co-maintenance. If a function is similar to few or none of the functions in a changelist, it will have a low changelist similarity score, and is marked as unlikely to require modification at the same time.

IV. EXPERIMENT

The goal of a maintenance recommender system is to enable a programmer to spend more time working on code instead of navigating it [8]. With that goal, we consider the position of the left-out function to be correctly predicted if it is in the top-n positions of the returned results, for several values of n. For example, in an IDE sidebar, it would not be practical to return several hundred results; users could not be expected to navigate through a long list of false positives. However, including the top-10 or top-25 is more reasonable for a user to consider. For completeness, we include a range of values in the experimental results. We also consider the different sizes of the systems. A model generated from system with thousands of functions, such as *php-src*, will realize more value if the predicted function is frequently returned in the top-25 results.

Table I lists the set of C open source systems used in this study. January 1st, 2010 was used as the model generation date for each model for consistency, although any date in the history of the project would have been acceptable. In the interest of reproducing the results, we provide full source code and instructions on how to generate the data[1]. The number of functions given in Table I is therefore the number of functions in the system at that date. The changelists referenced in column 3 of Table I is the total number of changelists

[1]The source code for this experiment is available at http://research.cs.queensu.ca/home/scott/csmrwcre2014/

TABLE II
CO-MAINTENANCE PREDICTION RESULTS FOR ALL SUBSETS OF ALL CHANGELISTS WITH 20 OR FEWER MODIFIED FUNCTIONS

System	Duration	Trials	Top-10	Top-25	Top-50	Top-100
git	365 days	1577	25.7% (406)	39.8% (627)	50.1% (790)	60.0% (946)
httpd	365 days	902	36.3% (327)	46.5% (419)	54.1% (488)	61.8% (557)
memcached	365 days	23	43.5% (10)	56.5% (13)	56.5% (13)	60.1% (14)
php-src	365 days	1813	19.4% (351)	29.0% (525)	39.2% (710)	49.8% (1813)

```
commit 36110078166239ba5b730a873b63f70e89de608c
Author: Daniel Earl Poirier <poirier@apache.org>
Date:   Tue Feb 16 20:50:10 2010 +0000

    Log command line on startup, so there's a record
    of command line arguments like -f.  Suggested by
    Shaya Potter.  [Dan Poirier]

    Functions modified:
    static int prefork_run(...) (prefork.c)
    static int event_run(...) (event.c)
    static int worker_run(...) (worker.c)
    static int netware_run(...) (mpm_netware.c)
```

Omitted: static int prefork_run(...) (prefork.c)	
2.9660	**static int prefork_run(...) (prefork.c)**
2.9626	static int make_child(...) (worker.c)
2.9563	static int make_child(...) (event.c)
2.9217	static void server_main_loop(...) (worker.c)
2.9167	static void server_main_loop(...) (event.c)
2.9150	static int make_child(...) (prefork.c)
2.8874	int ap_create_scoreboard(...) (scoreboard.c)
2.8751	int ap_process_child_status(...) (mpm_unix.c)
2.8632	void ap_init_scoreboard(...) (scoreboard.c)
2.8537	static int reclaim_one_pid(...) (mpm_unix.c)

Fig. 5. A sample changelist from Git's revision history for *httpd* for analysis.

over the entire life of the software system under revision. The number of changelists used in prediction is a subset of that value, based on the number of changes made between the date at which we force sync the source code and the time interval used for prediction.

We use the approach given in our earlier work for identifying an appropriate number of topics for analyzing source code [4]. In that work, we show that the appropriate topic count for a system is dependent on the number of methods or functions, and that a good value for the topic parameter can be obtained if the number of functions is known.

Table II provides the co-maintenance prediction results for all prediction trials of all changelists with 20 of fewer modified functions. We accept each changelist regardless of change type as a lower bound on the performance of the model to predict the observed co-maintenance relationships. The percentage of time that a left-out function occurs in the related predictions over all left-out functions in the timeframe is given. For example, if there are 902 prediction trials (not changelists, but subsets of changelists missing one function, as described in Figure 2) in the prediction window for *httpd*, and the top-10 percentage is 36.3% as given by Table II, the model is able to identify the left-out function in 327 of the 902 trials.

These models are generated with no prior information about co-maintenance between functions or other explicit links; these models evaluate similarity using the observed tokens in documents. Using this simple set of data, the co-maintenance relationships can be identified.

V. ANALYSIS

We use an example from the *httpd* project to show how a set of co-modified functions is used to predict an omitted function. Specifically, we show that if a function is left out from a changelist, and the remaining functions are used to identify related code fragments in the rest of the project, the most relevant functions often include the omitted function. The commit description, given at the top of Figure 5, describes this change as a modification to the startup routines. Four functions (*prefork_run*, *event_run*, *worker_run*, and *netware_run*) are modified in this change.In Figure 5, the top-10 similarity results for three of the four functions are given, and the omitted function and its score is highlighted in the list. Using the three remaining functions to find similarity across the rest of the functions allows us to identify the left-out function.

By using several functions to compare related code fragments in a particular context, two clear benefits are observed. Groups of co-maintained functions are often modified together in some context, so similar functions in the model are often also related to the same task. In addition, functions that are modified together in the current changelist can be pruned from the list of recommended functions, narrowing the results to a tighter focus. We review these two benefits in detail.

By examining multiple related modifications across functions, as would be observed in a developer's active changelist, a query against the rest of the functions in the software system will more often be in a specific task context. This implies that prediction results will be more accurate as more information about the current task context is known. For example, in Figure 5, any three of the four modified functions clearly leads to discovery of the omitted function, as all four functions are performing a related task. In the top-left portion of the table, the *prefork_run* function has been omitted from the similarity calculations, leaving the remaining three functions listed in the changelist above. When the remaining functions are used to calculate similarity against all other functions in the system, *prefork_run* is at the top of the list. In addition to this omitted function, other related functions can be seen, including the *make_child* functions from other run operations. In an expanded task context, it may make sense to include these functions in the changelist. And if these new functions are included, they will be used in the similarity detection, bringing up the relevance of other *make_child* or creation functions similar to the ones given. By using knowledge about an existing changelist, as performed in the study presented in this paper, better predictions can be obtained about forgotten or missing changes.

VI. Threats to Validity

In this work, we only examined systems written in C. This choice was made to minimize the number of variables in our study. It is possible that this approach does not generalize to other languages. For example, in our previous work on identifying appropriate topic counts using latent topic models [4], [5], it appears that a model is less likely to capture good co-maintenance information in Java than in C. We would like to expand this study in the future to include a wider range of systems in different languages.

An additional choice made to reduce the number of variables was a consistent point in time to generate each model. We chose January 1st, 2010, as it was recent enough to have meaningful changes in each of the systems, but distant enough to allow us to use longer time intervals for prediction. One issue this resulted in was a low number of changes in *memcached*. This can be seen in the low number of prediction trials identified in the 365-day interval. A future study will need to try a range of model dates in addition to a range of time intervals.

VII. Related Work

Ying *et al.* [16] were among the first to investigate how mining the change history of a project could be used to aid developers find related source code fragments. By identifying files that are changed together frequently in the project history, common change patterns are identified and used to suggest future changes. They performed an analysis on Eclipse and Mozilla, two large open-source systems, and show how change pattern not only be identified, but evaluated based on their predictability and interestingness.

Kersten and Murphy [8], [9] explore the task context model, an approach to reducing views of entire systems to context-specific views relevant only to the current task. This narrowing of focus has led to statistically significant increases in productivity, and has now been expanded into a top-level Eclipse project where it is used by millions of users. The task context model is further validated by additional studies [3], [14] showing how targeted recommendations can be a valuable part of the development environment.

The Hipikat project, developed by Čubranić and Murphy [15], is a tool that uses project archives to build a recommender for artifacts relevant to a particular task. Hipikat's goal is to assist new developers who do not have a full set of information about the software project by replacing the traditional mentoring role. Hipikat performed well as a starting point for new developers working on a new task, validating the use of recommendation systems.

VIII. Conclusion

In this paper we have explored the hypothesis that latent topic models, such as LDA, may be able to predict code fragments that should be changed together directly from the source code, without the need for a record of previous co-maintenance. We have tested this hypothesis using the actual revision histories of several open source projects as oracles for sets of co-maintained functions, and observed that LDA can often identify missing functions in a co-maintained set as part of the top 10 to 25 suggested related functions. When a set of several functions is used as the basis, the suggestions appear better than those we have observed for single functions in previous work.

Acknowledgements

This work is supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[2] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61:190–210, August 2006.

[3] Thomas Fritz, Gail C. Murphy, and Emily Hill. Does a programmer's activity indicate knowledge of code? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 341–350, New York, NY, USA, 2007. ACM.

[4] Scott Grant and James R. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '10)*, Timişoara, Romania, September 2010. IEEE Computer Society.

[5] Scott Grant, James R. Cordy, and David B. Skillicorn. Using topic models to support software maintenance. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*, pages 403–408, March 2012.

[6] Scott Grant, James R. Cordy, and David B. Skillicorn. Using heuristics to estimate an appropriate number of latent topics in source code analysis. *Science of Computer Programming*, 2013. To appear.

[7] A.E. Hassan and R.C. Holt. Studying the evolution of software systems using evolutionary code extractors. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*, pages 76 – 81, sept. 2004.

[8] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 159–168, New York, NY, USA, 2005. ACM.

[9] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06)*, SIGSOFT '06/FSE-14, pages 1–11, New York, NY, USA, 2006. ACM.

[10] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Xuan-Hieu Phan and Cam-Tu Nguyen. GibbsLDA++, A C/C++ Implementation of Latent Dirichlet Allocation (LDA) using Gibbs Sampling for Parameter Estimation and Inference. http://gibbslda.sourceforge.net, 2013.

[12] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

[13] Chanchal K. Roy and James R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. *Journal of Software Maintenance and Evolution, Special Issue on WCRE '08*, 22(3):165–189, 2010.

[14] J. Sillito, G.C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, July 2008.

[15] D. Čubranić and G.C. Murphy. Hipikat: recommending pertinent software development artifacts. In *25th International Conference on Software Engineering (ICSE2003)*, pages 408–418, May 2003.

[16] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574 – 586, September 2004.