Using Topic Models to Support Software Maintenance

Scott Grant
School of Computing
Queen's University
Kingston, Ontario, Canada
Email: scott@cs.queensu.ca

James R. Cordy
School of Computing
Queen's University
Kingston, Ontario, Canada
Email: cordy@cs.queensu.ca

David B. Skillicorn
School of Computing
Queen's University
Kingston, Ontario, Canada
Email: skill@cs.queensu.ca

Abstract—**Our recent research has shown that the latent information found by commonly used topic models generally relates to the development history of a software system. While it is not always possible to associate these latent topics with human-oriented concepts, it is demonstrable that they identify historical maintenance relationships in source code. Specifically, when a developer makes a change to a software project, it is common for a significant part of that change to relate to a single latent topic. A significant conclusion can be drawn from this: latent topic models identify co-maintenance relationships with no supervision, and therefore topic models can be used to support the maintenance phase of software development.**

I. Introduction

Making sense of data is hard. In the social sciences, such as psychology or economics, describing a large quantity of data typically requires a large set of variables. One approach to making sense of this information is the latent factors, unobserved variables that explains patterns of relationship among the observed data. For example, happiness is treated as an economic latent variable. Happiness cannot be measured directly, but it can be inferred from other observable variables such as lifespan and education.

In natural language, the observations taken typically relate to word frequency. These word counts are provided as input to a statistical structure called a topic model, in which a "topic" describes some relationship between parts of the data. This leads to a question for software developers. Can the same kind of observations be made of a software system? And if so, what are the patterns identified by the latent factors? Factor analysis has already been applied to software systems, but it is not clear if the results can be explained in human-oriented terms.

Based on extensions of three pieces of our related work, we show that the latent factors found in commonly used topic models relate to the development history of a software system [6]. While it is not always possible to describe software factors with human-oriented concepts, it is demonstrable that these factors identify historical maintenance relationships in source code. These historical maintenance relationships can be obtained using the revision control history of a project, such as CVS, Subversion, or Git source control repositories, and are referred to as the *co-maintenance history*. We observe that, when a developer makes a change to a software project, it is common for a significant part of that change to relate to a single factor. A significant conclusion can be drawn from this: latent topic models identify co-maintenance relationships with no supervision, and therefore topic models can be used to support the maintenance phase of software development.

Understanding that a relationship exists between topic models and co-maintenance history provides strong motivation for using topic models as a tool to help guide programmers during software maintenance. As discussed by Zimmermann *et al.* [22], the goals include suggesting and predicting likely changes, and preventing errors due to incomplete changes. In addition, understanding that a relationship exists allows for parameter estimation; we apply knowledge about source code locality to choose the number of topics to maximize an objective function.

In this paper, we explore the general task of using topic models to support software practitioners. We support our conclusions by examining the project history and topic models generated by a wide set of open source systems ranging from a few dozen methods to the entire Linux system. We also introduce visualizations to explore the relationship between topic models and software maintenance in different ways [6]. The relationships among individual code fragments are empirically evaluated in a case study using web services to show that discovery tasks for web service maintenance are supported by topic models [7]. We also use knowledge about source code structure to estimate an appropriate number of topics for modelling a software system [4]. We show how this can be used to generate an appropriate model for the intended task of software maintenance.

II. Previous Work

The search for latent factors in program documentation and source code has been going on for over a decade. Although it is, in a sense, straightforward to find some latent factors, it is substantially more difficult to interpret them as software or programmer concerns, and validation is troublesome due to a lack of obvious ground truth.

The first application of Latent Semantic Indexing (or LSI, an extension to singular value decomposition) to program comprehension was in 1999. Maletic, Valluri, and Marcus [14], [15] explored how LSI could be used in software by performing a handful of clustering and classification experiments for source code and documentation. Their work evaluated LSI's ability to cluster groups of related code together. Early results suggested that LSI could be used to support some aspects of the program-understanding process. However, the latent factors, as singular vectors extracted during the matrix decomposition, are not easily labelled or explained.

Latent Dirichlet Allocation (LDA) [1] is a generative statistical model in which a set of latent topics are assumed to determine documents and token distribution. LDA was first applied to source code in 2007 by Linstead *et al.*, who visualized topic emergence over several versions of a project [11], [12]. Shortly afterwards, in 2008, Maskeri *et al.* showed LDA's application to business topic extraction from source code [17]. The latent topics were described in vague detail, but for the most part, remained opaque. LDA is not able to provide human-oriented descriptions for the topics, and so manual inspection is necessary to label topics.

Voinea *et al.* [21] introduce a set of visualization techniques for software maintenance using software configuration management systems. They assess software project evolution using these management systems using various levels of granularity. Thomas *et al.* [19], [20] look at software evolution using topic models, and use the topics to describe software system evolution. These studies motivate topic model use for software maintenance and evolution.

Deciding how many dimensions to retain when performing a singular value decomposition or how many topics to identify when generating an LDA model has been described as art instead of science [13]. For natural language, many authors suggest values near 300 dimensions [14], [15], and a recent study showed "islands of stability" around 300 to 500 dimensions for document sets in the millions, with performance degrading outside that range [2]. Kuhn *et al.* [10] suggest using a value of $(m \times n)^{0.2}$ where m and n are the number of terms and documents in the matrix to decompose, and suggest that a smaller number of dimensions is warranted in analyzing software corpora because the document count is smaller than in typical natural-language corpora.

Our earlier research using independent component analysis [5], [8] showed a clear application of the technique for source code analysis. However, much like the topics identified with LDA, it is difficult to explain what information the signals are extracting from the original source data. By using co-maintenance history as defined previously as an oracle for predicting whether or not a technique is able to identify maintenance relationships between source code fragments, we have been able to explain and compare our results more confidently. In our own work, we were able to address these early concerns by developing visualizations.

III. Visualizing Topics By Changelist

A changelist is several code changes grouped together to represent a more complete view of a change [9]. For example, implementing a feature may involve many individual changes across several methods or files. A changelist groups these individual changes together. Since changelists frequently

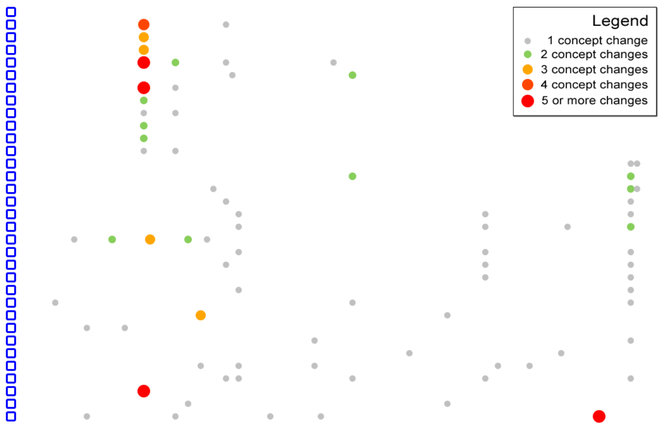

Fig. 1. A full view of the visualization for 35 consecutive changelists in httpd's history. Each blue square on the left defines a horizontal row corresponding to a changelist. Each column corresponds to a concept, and each coloured circle represents some number of modified code fragments from that concept. The size and colour of each circle shows how many code fragments from that concept were modified in this particular changelist.

group related changes together in this way, it follows that we would expect source code fragments modified together in a changelist to also be related in a topic model.

To explicitly show how topics and changelists are related, we developed a visualization to explore how topics are spread across individual changelists in the co-maintenance history of a software project. Viewing the topics at this level shows that code fragments that are modified together in a single changelist often share a conceptual relationship as discovered by statistical techniques such as Latent Semantic Indexing and Latent Dirichlet Allocation. In this work [6], we explored the relationship between topic models and co-maintenance history, and showed that there is a relationship by using a visualization. We also identified a number of common patterns observed, and analyzed their meaning.

Each visualization is generated as an interactive HTML page, based on the input from a source code repository and some secondary source, such as a topic model. An example of this can be seen in Figure 1. The rows of the display are the relevant changelists over the project's history, starting with the oldest, and progressing forward towards the newest revisions at the bottom. Each column in the display corresponds to one of the topics described by the model. The circles of varying size and colour on each row are an indication of which code fragments were modified in that changelist. The horizontal location of the circle indicates the topic in which the modifications were made. In LDA, there is no explicit ordering relationship between topic 1 and topic 2, and "adjacent" topics are not necessarily related to one another.

This visualization allowed us to identify a number of patterns that appear regularly. Vertical strips indicate concept development over time, such as single feature development or maintenance. For example, it is common to observe a long series of changelists that strongly correspond to the same topic as a vertical line of coloured circles. In some cases,

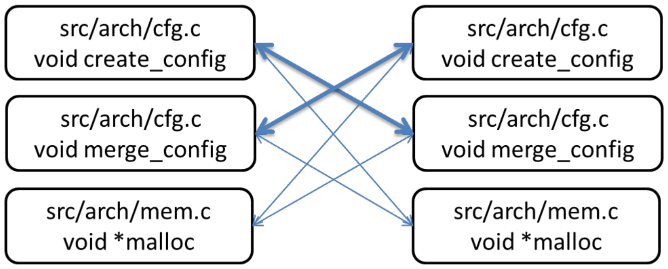

Fig. 2. A visual representation of how *Bluevis* maps functions to positions on either side of the screen. Strong conceptual relationships are marked by a strong line. Self-references are removed, so the display is not dominated by horizontal lines.

this is followed by a horizontal bar in which the feature is enabled across the system. Horizontal strips indicate cross-cutting aspects, or system-wide changes.

By generating a two-dimensional table of information about the topic distribution of code changes and the individual changelists, we are able to visually observe the project history. It also allows us to observe the topic model's relationship with maintenance, and motivates the analysis that follows.

IV. Pairwise Visualization

Understanding when pairs of code fragments are considered similar is necessary when using a topic model in a practical setting. For example, a topic model can be part of the software development process if we understand the type of information it provides. However, topic models are techniques that identify clusterings in a set of input documents, and *topic* is a loaded term; they are described by a topic model as patterns that may be shared across many documents and used to cluster information. Individual topics are patterns of data, and are not guaranteed to have a human-oriented representation.

Topic clusters group related code fragments together. However, the individual pairwise similarity is arguably much more intuitive for end users. For example, this paradigm is used in search engines, where only the results are presented and not the means by which the results are obtained. In Zimmermann *et al.* [22], the paradigm is also used to compare how customers view related items when shopping online and how software developers could view related source code fragments when writing source code.

Even if they are abstract and undefinable in a human-oriented context, the relationship between the entire set of topics discovered by a model can be used to observe similarity between individual code fragments. Since individual topics may not be explained as a particular concept, an alternate approach is to use a similarity metric to evaluate one document's relevance to another. For example, in a software system that has been modelled using LSI, the cosine distance between any of the vectors in the new space can be used to obtain a similarity estimate between the original documents represented by those vectors.

One visualization that we developed to evaluate topic model performance is called *Bluevis*. *Bluevis* explores how the file

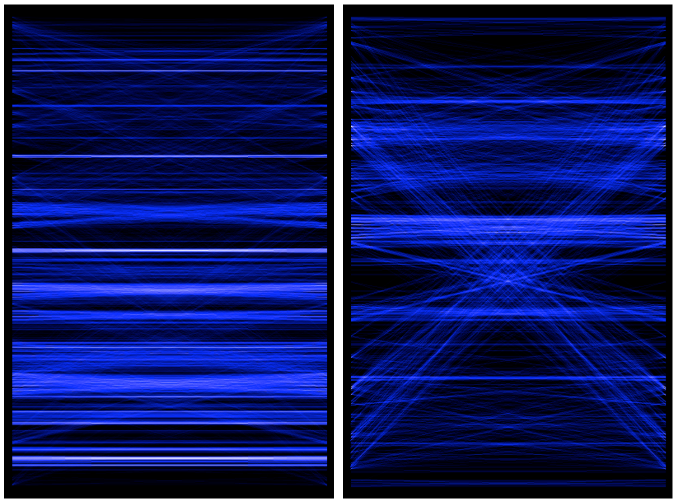

Fig. 3. Two screenshots of *Bluevis*'s interface. The left and right sides of each screenshot correspond to an alphanumerically ordered list of the source code functions, and a connection made between the sides represents a conceptual similarity between two different source code fragments above a user-defined threshold. On the left, the large number of horizontal lines indicates a strong local conceptual structure, and may indicate that the code is structured in a good way. On the right, the related functions are sparsely distributed on a large scale with only a few small sections of strong locality, and probably indicates poor package design.

structure of the system relates to its conceptual distribution. In a system that is well designed conceptually and architecturally, a majority of conceptual pairings (pairs of source code fragments with a conceptual score higher than some threshold value) should reside near one another. Related methods should be found in the same files or folders instead of spread across the system.

In systems where we do not have a full revision history, we can use source code locality as an estimate. If the topic model is able to identify co-maintenance relationships, we believe that this will be reflected in *Bluevis* by a view with relatively little noise.

Each side of the window in Figure 3 relates to an alphanumerically ordered list of the source code functions in the same software system. The first position in the ordered list is the same function in each side of the window. If a line extends from one side of the window to the other, it represents a conceptual relationship (above a threshold) between those two source code functions. Larger blocks of related code form brighter lines, and random pairings appear as dark and almost invisible. In this way, the global conceptual structure of the system can be seen. elf-references are removed, so no line is horizontal.

We expect aspects and other cross-cutting concerns to be distributed across the code. Some topics span the entire codebase, and it would not be surprising to see them. The benefit of a plot like *Bluevis* is that the aspects can be visualized directly using this method. For example, in the left visualization from Figure 3, there are several fans near the top of the graph.

Smaller systems will not generate interesting visualizations because of the weakness of method ordering. *Bluevis* uses an alphanumeric ordering by file and folder, and this works best on a larger system. For flat file structures with no hierarchy, or large directories with many files, there will be significant scattering. On systems of a reasonable size, this visualization offers a good way to investigate the global conceptual layout of the software system.

We have also begun to use this visualization as a way to compare how good various models are at capturing conceptual structure in a particular data set. For example, given a source code package, and using LDA and LSI with various topic and dimension counts, how similar do the visualizations look for each model, and how much do the models cross over as in the right image in Figure 2. If all of the visualizations for all of the models look similar, we may conclude that the models are essentially similar.

A. Case Study: Web Services

This research uses topic models to predict co-maintenance of code fragments. We also use topic models for a related problem, uncovering web service similarity between service specifications written in a domain-specific language. Web services are software components used to communicate over a network. These web services are often described using domain-specific languages, outlining the existing operations, the type of messages that can be sent, and other information about the provider. The structure of service descriptions written in the Web Service Description Language (WSDL), one such domain-specific language, makes reading and understanding them a difficult task. This problem makes it even more difficult to discover relationships between service operations when considering a large repository of web services.

The sparsity of tokens in high-level languages such as WSDL do not provide enough context for a topic model to make interesting conclusions. We solve the sparsity problem by explicitly injecting referenced elements into the WSDL source (see [16] for details and examples). This explicit inclusion of referenced tokens increases the information in the document, and provides context that was previously implicit and unusable. Our theory was that contextualizing WSDL operations to use as input for an LDA model would produce a significant improvement in web service discovery using topic models.

We must consider how *Bluevis* can actually provide relevant information to benefit the maintenance process. In our experience, we have consistently observed that visualizations with visible noise show poorer maintenance relationships, and visualizations with visible structure, such as the one on the left in Figure 3, identify stronger maintenance relationships. The visible structure can be examined in detail to show strong global features of the code, such as large subsystems in software or large blocks of related web services in this instance. In this way, it is possible to visually inspect the topic model generated by using the bare operations and to presume that a model generated by using contextualized operations will do a better job at supporting web service maintenance.

Using *Bluevis*, we observed that basic WSDL operations provided a much more visually chaotic semantic structure than contextualized operations. Many of the similar operations identified using the basic WSDL operations as input are meaningless, and show up due to shared tokens like *get* or *SOAP*. The visualization using contextualized operations showed a significant reduction in sparse random connections. When we investigated these more closely, it appeared that a majority of the information shown by the visualization contained relevant semantic structure. Several large fan-out points appeared, indicating web services that have similar operations offered by other providers. Large horizontal blocks indicated clusters of related operations from the same web service.

A local view of the improvements helped to explicitly show how individual operations were similar to one another, and specifically how the relationships become more useful after contextualization. To show that this type of improvement is common, we expanded our view to look at the most similar operation for single web service operations. In the majority of cases, the contextualized operations are more useful in a human-oriented context. With contextualization, a topic model like LDA is able to use the newly added tokens to derive interesting correlations between operations.

V. Estimating Topic Counts

In many cases, a full set of revision history does not exist for a software system. In this section and, as in the work with *Bluevis*, we assume that the revision history is unavailable and use source code locality in its place. If topic models are to be used in software maintenance, a reasonable effort must be made to obtain the most appropriate model for the task. To obtain the best fitting model for the data, we must ensure that an appropriate topic count is used.

In this section, we infer an appropriate number of latent topics needed to optimize the topic distributions over a set of source code methods [4]. The topic count has a significant effect on the model, and too few or too many topics may produce clusters that do not show relationships in the source data.

We segment a source code package into its individual methods, treat each method as a document, and generate successive LDA models with different values for the number of latent topics. Our goal is to maximize the model's ability to predict a known metric in the source code, similar to the co-maintenance metric used in our other research. In this section, our metric is a simple heuristic based on location in the source code package structure that gives a quick and reasonable estimate about whether or not a pair of methods are conceptually similar to one another. The proximity measure suggests that two code fragments are likely to be conceptually related if they are found in the same file or folder.

For each document, the n nearest neighbours are determined using the cosine distance by interpreting the rows of the topic membership matrix as vectors. The number of documents among the nearest neighbours that are conceptually related

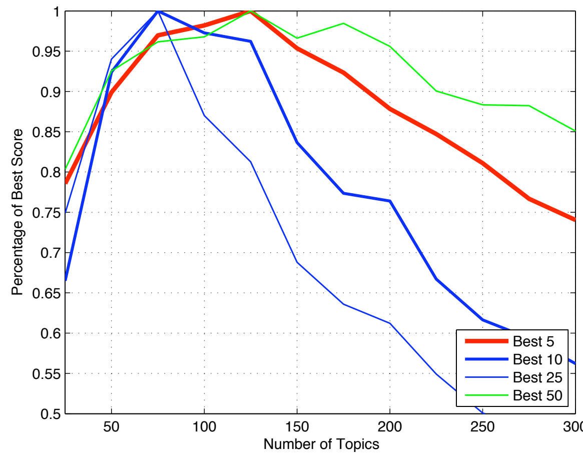

Fig. 4. PostgreSQL proximity scores. By combining the similar source code fragments identified by a topic model with the source code fragments that are nearby in the software package file hierarchy, peaks can be uncovered. In this case, a clear peak emerges around 75 to 125 topics, suggesting that the appropriate number of latent topics lies in this range.

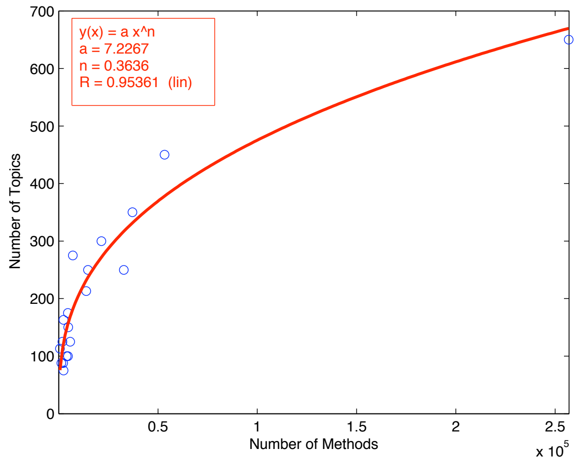

Fig. 5. Fitting a curve to the estimated appropriate topic counts. The x-axis measures the number of source code fragments, which for us is the number of code fragments. The y-axis measures the number of topics that our approach has estimated for the appropriate human-oriented value.

using the code proximity heuristic is taken as a document score, and the average score over all documents is the *nearest neighbour score* for the latent model with *k* topics.

For each topic, the *m* documents with the highest probability of fitting this topic are identified, and these can be considered the documents that best represent the information uncovered by this latent topic. For each of the *m* documents that best match a topic, we determine how many of the other *m* documents are conceptually related using the code proximity heuristic, and take the sum of the scores as the *proximity score* for the latent model with *k* topics.

These two results provide us with a way to estimate how well a topic model is able to uncover the latent topics that identify conceptually related documents. The nearest neighbour score for the latent model describes how well the nearest neighbours of a document in the vector space relate conceptually. The proximity score for the latent model describes how well the topics in the model are able to match up related sets of documents with one another.

To evaluate our approach, we tested 26 different open-source systems written with four different programming languages (C, C#, Java, and Python). The smallest system we examined was gzip, with 117 different functions. The largest was the entire Linux system, with more than 250,000 source code functions. For each system, we generated a range of LDA models, and calculated the proximity and nearest neighbour scores for each model. In most cases (almost all software systems with more than a few hundred code fragments), the proximity score increased to a peak value and then dropped off quickly. We identified these peak values, and fitted a curve to the observations we obtained.

For a source-code system with *m* code fragments, Equation 1 provides an estimate for the appropriate number of topics for source code using LDA based on our observations of several dozen software systems.

$$t(m) = 7.25 * m^{0.365} \tag{1}$$

This equation can be compared to an estimate given by Kuhn *et al.* in [10]. For an $m \times n$ document-term matrix, where m is documents (classes, instead of methods or functions) and n is the term count over all documents, the authors suggest using a value of $(m \times n)^{0.2}$. In the software systems we examined, the average term-to-document ratio suggests a linear relationship between the tokens and source code fragments. From this, we can approximate the equation by replacing n by m to get $(m^2)^{0.2}$, or $m^{0.4}$. The exponent 0.4 suggests that as the document count increases, the appropriate rate of change between Kuhn's topic count and ours is roughly similar.

While proximity is not a perfect heuristic for similarity, recent work in the clone detection community has demonstrated a clear relationship between proximity in the package structure and the likelihood of finding clones [18]. Using this, together with our observation that clones often share similar semantic information and are frequently identified as semantically related in latent models [3], we believe that the proximity score is a reasonable measure. In smaller systems, and those that use methodologies such as aspect-oriented programming, source code locality in files and folders may be reduced.

Poor parameter choices can lead to models that provide little to no benefit for software analysis. However, when combined with an understanding of what information the model is capturing, making good choices for parameter values can demonstrably provide insight into how a software system is organized. Accurate and appropriate models are necessary if topic models are to be used for software maintenance. For example, topic models that identify similarity between code fragments could replace the find function in a programming environment.

VI. Expected Impact

Latent topics emerge from code fragments, but we don't yet know what they mean. In this research, we analyse software maintenance history, and show that topics represent code fragments that are maintained together. We can use this correlation both to categorize and understand maintenance history, and to predict future co-maintenance in practice.

In program comprehension, topic models can be applied to both high and low level views of a software system. For example, at a high level, a topic model can identify what the syntax suggests about the software architecture. From a maintenance standpoint, this information may suggest significant refactorings because code is poorly structured. At a low level, for individual source code functions, a topic model can identify other similar functions. From a maintenance standpoint, this knowledge is important when fixing bugs and preventing faults.

VII. Conclusion

This paper builds on our existing research by exploring the relationship between latent topic models and co-maintenance history. Specifically, by taking a close look at the results of our recent work, we believe that latent topic models can identify co-maintenance relationships in source code systems. This relationship can be used in the maintenance phase of software development to suggest related changes and prevent errors.

Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada, by the Ontario Graduate Scholarship Program, and by an IBM CAS faculty award.

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] Roger B. Bradford. An empirical study of required dimensionality for large-scale latent semantic indexing applications. In *Proceeding of the 17th ACM Conference on Information and Knowledge Management (CIKM '08)*, pages 153–162, New York, NY, USA, 2008. ACM.

[3] Scott Grant and James R. Cordy. Vector space analysis of software clones. In *17th IEEE International Conference on Program Comprehension (ICPC '09)*, pages 233–237, May 2009.

[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. Automated concept location using independent component analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 138–142, October 2008.

[6] Scott Grant, James R. Cordy, and David B. Skillicorn. Reverse engineering co-maintenance relationships using conceptual analysis of source code. In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*, pages 87–91, October 2011.

[7] Scott Grant, Douglas Martin, James R. Cordy, and David B. Skillicorn. Contextualized semantic analysis of web services. In *Proceedings of the 13th IEEE International Symposium on Web Services Evolution (WSE '11)*, pages 33–42, September 2011.

[8] Scott Grant, David B. Skillicorn, and James R. Cordy. Topic detection using independent component analysis. In *Proceedings of the 2008 Workshop on Link Analysis, Counterterrorism and Security (LACTS '08)*, pages 23–28, April 2008.

[9] 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.

[10] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[11] Erik Linstead, Cristina Lopes, and Pierre Baldi. An application of latent dirichlet allocation to analyzing software evolution. In *Proceedings of the 2008 7th International Conference on Machine Learning and Applications (ICMLA '08)*, pages 813–818, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pages 461–464, New York, NY, USA, 2007. ACM.

[13] Apache Mahout. Latent Dirichlet Allocation. https://cwiki.apache.org/MAHOUT/latent-dirichlet-allocation.html.

[14] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '00)*, page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[15] Jonathan I. Maletic and Naveen Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE '99)*, page 251, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Doug Martin and James R. Cordy. Towards web services tagging by similarity detection. In Mark Chignell, James R. Cordy, Joanna Ng, and Yelena Yesha, editors, *The Smart Internet*, pages 216–233, 2010.

[17] Girish Maskeri, Santonu Sarkar, and Kenneth Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st Conference on India Software Engineering Conference (ISEC '08)*, pages 113–120, New York, NY, USA, 2008. ACM.

[18] Chanchal K. Roy and James R. Cordy. Are scripting languages really different? In *4th International Workshop on Software Clones (IWSC '10)*, May 2010.

[19] Stephen W. Thomas, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. Validating the use of topic models for software evolution. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 55–64, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Stephen W. Thomas, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. Modeling the evolution of topics in source code histories. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 173–182, New York, NY, USA, 2011. ACM.

[21] Lucian Voinea, Johan Lukkien, and Alexandru Telea. Visual assessment of software evolution. *Science of Computer Programming*, 65(3):222 – 248, 2007. Special Issue on: Software Configuration Management (SCM).

[22] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429 – 445, june 2005.