A Topic Modeling Approach for Code Clone Detection

Sandeep Reddivari
School of Computing
University of North Florida
Jacksonville, FL, USA 32224

Mohammed Salman Khan
School of Computing
University of North Florida
Jacksonville, FL, USA 32224

ABSTRACT

In this paper we investigate the potential benefits of Latent Dirichlet Allocation (LDA) as a technique for code clone detection. Our objective is to propose a language-independent, effective, and scalable approach for identifying similar code fragments in relatively large software systems. The main assumption is that the latent topic structure of software artifacts gives an indication of the presence of code clones. In particular, we hypothesize that artifacts with similar topic distributions contain duplicated code fragments. To test this novel hypothesis, we conduct an experimental investigation using multiple datasets from different application domains. Preliminary results show that, if calibrated properly, topic modeling can deliver satisfactory performance in capturing different types of code clones. It also achieves levels of accuracy adequate for practical applications, showing comparable performance to already existing tools that adopt different clone detection strategies.

Keywords

Refactoring, Topic Modeling, Code Clones

1. INTRODUCTION

Code clones are similar code fragments that appear in a software system [10]. It is estimated that a typical mid-size industrial system contains up to 20% of duplicated code [4, 15, 24]. Clones are often produced by the *copy-&-paste* practice of programmers [16]. Rather than rewriting working code fragments from scratch, programmers prefer to copy, and perhaps slightly modify, working code that has already been tested before [9]. The main assumption is that, simply making a copy of a working code is faster and is less likely to introduce new bugs, especially when a deadline is approaching [29]. However, from a refactoring perspective, code clones are considered a major code smell [3, 10]. They significantly increase the maintenance cost and the error proneness of the code [14, 20]. For instance, inconsistent changes to code duplicates can lead to unexpected behavior [24]. Therefore, clones must be kept in sync during maintenance [21]. In particular, when a bug is fixed in an instance of a cloned code, all other duplicates must be altered as well. In addition, clones decrease the modularity of the system and its level of encapsulation, as well as unnecessarily increase the size of the code which can complicate future maintenance tasks and reduce understandability [9].

DOI reference number: 10.18293/SEKE2018-179

Therefore, code clones have to be refactored whenever detected [18, 30]. Based on the notion of similarity established between code fragments, various types of code clones can be identified. Clones can range from exact matches where the same code fragment is copied, to functional clones where two code fragments perform the same operation but they are syntactically and semantically different [30, 4, 19]. A plethora of clone detection tools have been proposed in the literature [8, 30]. Such tools often support a large variety of programming languages, and adopt different clone detection strategies, at different levels of complexity, designed to target various types of clones. Despite these advances, the usage of clone detection tools is still not pervasive in industry [14]. In general, most of these tools are still far from achieving optimal accuracy. This requires developers working with such tools to manually classify and verify the detected candidate clones [8, 30], a process that is often described as time-consuming and error-prone [25]. In addition, there is still a lack of adequate support for large-scale systems, where clones are likely to spread over several code modules [13, 21].

In an attempt to address these issues, in this paper we propose a novel approach, based on topic modeling, to facilitate a more accurate, language-independent, and scalable clone detection process. In particular, we experiment with Latent Dirichlet Allocation (LDA), the most commonly used technique for topic modeling in Natural Languages Processing (NLP) [7]. LDA is a probabilistic statistical approach for estimating a topic distribution over a text corpus [7]. Our main conjuncture is that, the topic distribution over a code base gives an indication of the presence of code clones. Our contributions in this paper are the following. First, we propose an effective, language independent, and scalable approach for detecting code clones based on topic modeling. Second, we provide an experimental benchmark for calibrating LDA parameters and evaluating its performance in detecting various types of code clones.

The rest of the paper is organized as follows. Section 2 briefly introduces the background and related work. Section 3 introduces our research methodology. Section 4 presents our experimental analysis and results. Section 5 presents threats to validity and finally Section 6 concludes the paper and discusses the future work.

2. BACKGROUND AND RELATED WORK

2.1 Latent Dirichlet Allocation

LDA was first introduced by David Blei et al. [7] as a statistical model for automatically discovering topics in large corpus of text documents. The main assumption is that documents in a collection are generated using a mixture of latent topics, where a topic is a dominant theme that describes a coherent concept of the corpus's subject matter. In recent years, LDA has been utilized heavily to aid several essential software engineering activities. For instance, Andrzejewski et al. [1] proposed an approach based on LDA to support statistical debugging tasks in software systems. Results showed that LDA-based approach outperformed existing methods for bug cause identification. In addition, Thomas et al. [31] used LDA to study source code evolution. Analysis showed that evolution caused by change activity was often reflected in the topic mixture of the system.

LDA takes the documents collection D, the number of topics K, and α and β as inputs. Each document in the corpus is represented as a bag of words $d =< w_1, w_2, \ldots, w_n >$. Since these words are observed data, Bayesian probability can be used to invert the generative model and automatically learn ϕ values for each topic t_i, and θ values for each document d_i. In particular, using algorithms such as Gibbs sampling [28], an LDA model can be extracted. This model contains for each t the matrix $\phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, representing the distribution of t over the set of words $< w_1, w_2, \ldots, w_n >$, and for each document d the matrix $\theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$, representing the distribution of d over the set of topics $< t_1, t_2, \ldots, t_n >$. Several methods have been proposed in the literature to approximate near-optimal combinations of LDA parameters (α, β, K) in source code. Asuncion et al. [2] and Oliveto et al. [26] proposed the usage of LDA to automatically capture traceability links in software systems. Maskeri et al. [23] proposed a human assisted approach based on LDA to extract business domain topics from source code.

2.2 Code Clones

Four types of clones can be identified based on the notion of similarity considered between code fragments [4, 19, 30]. These types include:

Type I: Exact clones in which the same fragment of code is copied without modification in its semantic or syntactic structure (except for spacing and comments).

Type II: Clones that are syntactically identical fragments except for slight variations, such as different identifiers names, literals, types or spacing.

Type III: Clones that have been slightly changed by added, removed or re-ordered statements, in addition to Type I and Type II modifications.

Type IV: Functional clones which refer to code fragments that perform similar operations but their syntactic and semantic structures are different.

Detecting different types of clones requires different levels of sophistication. While Type I and Type II can be relatively easy to detect using lexical-based techniques, other types (especially Type IV) require a higher level of complexity to match operationally identical code fragments.

3. RESEARCH METHODOLOGY

3.1 Datasets

To conduct our experimental analysis, we used four software systems from different application domains. Table 1 describes the characteristics of these systems including: the size of the system in terms of lines of source code (SLOC), lines of comments (CLOC), implementation language (LANG.) version (VER.) and number of classes (CLS).

Table 1: Experimental Datasets

Dataset	VER.	CLS.	LANG.	SLOC	CLOC
iTrust	15.0	299	Java	20.7K	9.6K
Apache Ivy	2.3.0	451	Java	49.9K	16.7K
QuantLib	1.3.0	874	C++	178.8K	22.3K

3.2 Implementation and Tool Support

In this paper we use JGibbLDA, a Java implementation of LDA . This particular implementation uses Gibbs Sampling for parameter estimation and inference [12]. To integrate JGibbLDA in our analysis, a *C#* prototype is implemented upon the current Java implementation. We refer to this prototype as *CloneTM*, a code **Clone** detection tool based on **T**opic **M**odeling. Our interface provides options to tune the underlying LDA model (α, β, K), as well as visualization support for LDA results. For instance, stacked charts and bar charts are available for visually comparing topic distributions of multiple artifacts.

3.3 Dominant Topic Analysis

We start our analysis by investigating the effect of different values of K over the topic distribution generated for each artifact in each of our experimental systems. It is important to point out that the complexity of the study grows exponentially with the inclusion of other LDA parameters such as α and β. Therefore, at this stage of our analysis, we fix the the values of these parameters. This strategy is often used in related research to control for such variables' effect [11, 22, 31]. In particular, values of $\alpha = 50/K$ and $\beta = 0.1$ are used. These heuristics have been shown to achieve satisfactory results in the literature [12, 32].

One interesting observation of our topic analysis is that for each artifact $d_i \in D$, regardless of K, there is always a few number of topics that stand out from the rest of the topics in the document-topic matrix. Such topics have relatively larger θ_i values. To demonstrate this effect, we run LDA using K=40 over our *iTrust* experimental dataset. For each artifact d in the document-topic matrix of *iTrust*, we sort the 40 topics in a descending order according to their θ_i value, so that topics at rank ($r = 1$) have the highest θ value. We then average these values for all artifacts (N) in *iTrust* over each rank r, producing A_r (Eq. 1).

$$A_r = (\frac{1}{N}) \sum_{j=0}^{N} d_j(\theta_r) \quad \forall r = 1, 2, \ldots, 40 \tag{1}$$

The results are shown in Fig.1 which shows that in the topic distribution of each artifact, only a small portion (≈ 5) of the topics has a θ value larger than a certain threshold value (λ). These topics with relatively large θ values are known as dominant topics [22, 27]. In an attempt to specify λ, we conduct further empirical analysis over our open source experimental systems using different values of K. Results show that topic probability distribution for each artifact seems to always follow a regular distribution. In general,

http://jgibblda.sourceforge.net/

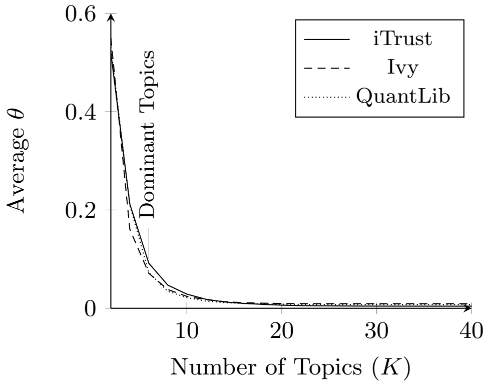

Figure 1: Average θ values in the document-topic matrix arranged in a descending order (K=40).

three categories of topics, based on the empirically observed λ, can be identified as follows:

- $\lambda_1(\theta < 0.01)$: Most of the topics in the document-topic distribution of each artifact fall under this category.
- $\lambda_2(0.1 > \theta => .01)$: Dominant topics, an average of 4 to 8 topics for each document.
- $\lambda_3(\theta => 0.1)$: Absolute dominant topics, usually one or two topics are classified under this category.

We use these observations about dominant topic distributions to derive our main hypothesis in this paper. In particular, we assume that the presence of code clones can be reflected in the dominant topic distribution of software artifacts. Our main assumption is that documents sharing similar code fragments might also share a similar dominant topic distribution. Next we test these assumptions.

4. DETECTING CODE CLONES

To test the hypothesis, we devise an experimental benchmark to analyze the performance of LDA in detecting code clones. In particular, we manually inject code clones of Types I, II, and III in each of our experimental datasets. We exclude Type IV refactoring in this study. Manually injecting and verifying code smells for refactoring studies is a common practice in related research, especially in *proof-of-concept* studies [18, 6, 17, 8]. Also, since we are working with class granularity level, we limit our analysis in this paper to cross-class or cross-file clones.

Table 2 shows characteristics of our injected clones, including the number of clones injected of each type in each system (NO. C) and the number classes affected (CLS). Since *QuantLib* is a relatively larger system, we were able to inject more clones in it. Injecting Type I and II is a straightforward process. In particular, to produce Type I clones, a method call is simply replaced by the method itself, changes in spacing and comments are made. When injecting Type II clones, parameters names are changed. Injecting Type III clones was challenging as code statements have to be reordered, added, and removed. To achieve this, we apply random sequences of certain operation-preserving transformations into copied code fragments. These transformations include:

- Conditional Statements: Break and merge certain `if` and `while` statements into `if else` statements and vice versa. For example, the code segment:

```
If(validteUsrNm(uName) && validPwd(uPwd))
  return true;
```

  can be broken down into:

```
If(validteUsrNm(uName))
  if(validPwd(uPwd))
    return true;
```

- Loops: Certain `for` statements were converted into WHILE loops and vice versa. For example, the following loop statement:

```
for(line=br.readLine(); line!=null; line=br.readLine())
```

  is transformed to the following `while` statement:

```
line = br.readLine();
  while (line != null)
    line = br.readLine()
```

- Re-order: certain statements were reordered in such a way that does not change the structure of the code. For example, in the following code segment, variable `fBloodPressure` declaration can be moved above the method call `setPatientRecords(patientID)` without affecting the functionality of the code.

```
setPatientRecords(patientID);
float fBloodPressure = 0.0;
fBloodPressure = pm.getPressure(patientID)
```

The Second step is to define the notion of matching between dominant topics. In general, we follow a set-matching procedure, if any two classes have the same topic appearing in their set of topics with $\theta_i > .01$, we consider this case to be a candidate instance of code clones. We use the word *"candidate"* since we suspect that in some cases, matching also might happen without the presence of cloning. In that case we get a false positive. The procedure for our LDA-based clone detection technique can be described as follows.

```
     Detect Clones: INPUT $D$, $\alpha$, $\beta$
1.   $K = 40$;
2.   Doc_Topic_Metrix = Generate_Topic_Model($D$, $\alpha$, $\beta$, $K$)
3.     FOR EACH $d_i \in D$ IN Doc_Topic_Merix
4.       FOR EACH $t_j \in di$
5.         IF $\theta_j < .01$ THEN Remove $t_j$
6.   FOR EACH $d_i \in D$
7.     FOR EACH $d_j \in D$
8.       IF i $\neq$ j
9.         IF(Match (Doc_Topic_Merix(di, dj)) > 0)
10.          RETURN TRUE
11.  $K$ += 40
12.  GOTO 2
```

To assess the effectiveness of LDA in capturing instances of different types of clones. Standard recall and precision metrics of information retrieval are used. Such metrics are often used to assess the performance of clone detection tools [8, 6, 30]. Recall measures coverage and is defined as the percentage of clones that are correctly identified by the tool, and precision measures accuracy and is defined as the percentage of identified clones that are correct.

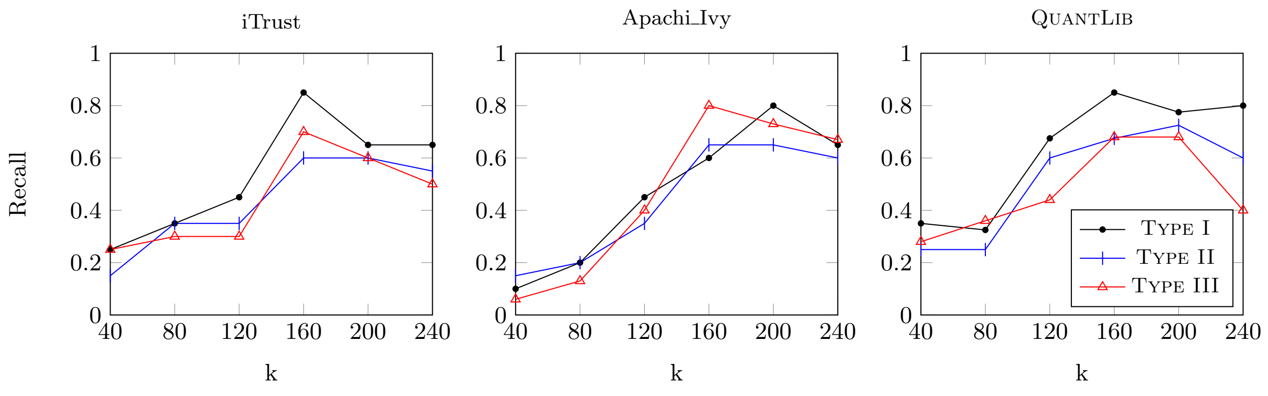

Figure 2: Recall values of different types of code clones at different values of K.

4.1 Model Calibration

To run our analysis, we initially set K to be 40 topics. The document-topic distribution of each artifact in the system is then generated. A pair-wise comparison is conducted to capture matching in the latent topic structures of different classes. Results are then evaluated against the answer set using recall, or the number of injected clones the tool managed to identify. The value of K is then increased by 40 and the process is repeated. This particular step size has been found to yields noticeable changes in the recall values. We follow a hill climbing approach to monitor the changes in the recall. Our objective is to identify, or approximate, near-optimal K settings to detect clones. We tie optimality in this paper to recall. Therefore, in our analysis we emphasize recall over precision. The main rationale is that errors of commission (false positives) are easier to deal with than errors of omission (false negative). In other words, it is easier for a developer to discard instances that were misclassified as clones, rather than deal with clones that were not detected by the tool.

Since we have injected different types of clones, we were able to produce a separate recall curve for each type (cf. Fig 2). As for precision, a single precision chart, which shows the percentage of misclassified cases, is produced for each system (cf. Fig 3). The list of candidate clones generated for each system was scanned for already existing cross-file clones before injecting our clones, such clones were excluded from our precision calculations. We implemented our evaluation benchmark into *CloneTM*. Candidate clones are displayed to the user and the lines of code which include words from the matching topics are highlighted in the class view window of each class. Results show that in all three systems the recall seem to converge to a local maximum at the range of $K = [160, 200]$ topics for all systems. The precision values also show satisfactory performance at this level. This can be explained based on the observation that at this range of K, topics tend to be more distinguishable from each other which makes this particular number of topics seem to be the most nearly optimal for code clone detection. However, at lower values of K, topics tend to have less density, generally spreading all over the class, and at higher K values (i.e., > 200) topics tend to be very specific, not able to cover code fragments with meaningful size. In general, the results show that our LDA-based approach was able to capture most types of clones, showing particularity good performance in detecting Type I and Type III clones. Results also showed that the precision, while can be considered satisfactory, is still far from being optimal. To put the performance of *CloneTM* in perspective, we compare its recall and precision with other clone detection tools such as CCFinder [15] and CloneDR [5]. Table 3 shows the results of the tool comparison. For each type of clone in each system we compare the recall values. Results show that *CloneTM* is able to achieve comparable levels of recall to other tools in all systems. In particular, results show that our LDA-based approach managed to outperform *CCFinder* in Type III refactorings. Which can be explained based on the fact that the sequential analysis of code statements in *CCFinder* makes it fragile to statement reordering and code insertion. In general, many other token-based detection approaches do not detect clones with reordered statements [30]. However, the fact that LDA treats a class as a bag of words makes it immune to such changes. In contrast, results show that *CCFinder* was more successful in detecting Type II clones, this can be explained based on the fact that token-based methods are immune to name changing. On the other hand, LDA can be very sensitive to the information value embedded in the identifiers names and comments, so inconsistency in such information is expected to lower the accuracy. Results also show that, in comparison to *CloneTM* and *CCFinder*, *CloneDR* captured the smallest number of clones in all different types of clones. That might be explained based on the fact that this tool tends to do better in cross-method rather than cross-file clones detection [8].

Table 2: Injecting Code Clones into Our Systems

	iTrust		Ivy		QuantLib	
Type	No. C	CLS	No. C	CLS	No. C	CLS
Type I	20	42	20	85	40	86
Type II	20	44	20	17	40	92
Type III	15	30	15	45	25	50

5. THREATS TO VALIDITY

This study has several limitations that might affect the validity of the results. In terms of external validity, the results of this study might not generalize beyond the underlying experimental settings. For instance, only four systems

Table 3: Comparing Recall Values of *CloneTM*, *CCFinder*, and *CloneDR*

	Recall								
	Type I			Type II			Type III		
System	*CloneTM*	*CCFinder*	*CloneDR*	*CloneTM*	*CCFinder*	*CloneDR*	*CloneTM*	*CCFinder*	*CloneDR*
iTrust	0.85	0.9	0.2	0.6	0.8	0.2	0.7	0.4	0.13
Ivy	0.8	0.8	0.2	0.65	0.85	0.15	0.73	0.67	0.27
QuantLib	0.775	0.7	0.35	0.575	0.7	0.25	0.68	0.53	0.32

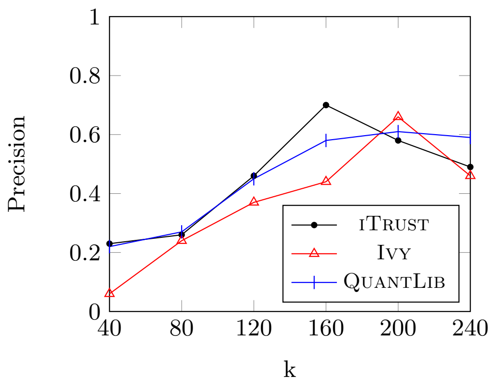

Figure 3: Precision values different values of K.

were used in our analysis. Nevertheless, we believe that using four datasets from different domains, including a proprietary software product, helps to mitigate these threats. In fact, we believe that using these heavily-used, open source tools and systems increases the reliability of our results as it makes it possible to independently replicate our results. Other threats to the external validity might stem from specific design decisions, such as using heuristic values for α and β. However, as mentioned earlier, due to the exponential complexity of the problem, it was not feasible to evaluate the effect of all LDA parameters in this study. In addition, the heuristics we used in our analysis have been proven to achieve satisfactory performance in related research.

Internal validity refers to factors that might affect the causal relations established in the experiment. A major threat to our study's internal validity is the fact that we used manually injected clones to calibrate our model, in addition to manually verifying the candidate clones of different tools. This can lead to an experimental bias due to the subjectivity of this process. However, this particular experiment design decision was necessary to gain more insight into our procedure's performance, in particular, its effectiveness in detecting different types of clones. In addition, as reported earlier, in the current state of research, human approval of the outcome of the code clone detection tool or method is inevitable.

6. CONCLUSION AND FUTURE WORK

In this paper we proposed a novel approach based on topic modeling for code clone detection. In particular, we investigated the potential benefits of using LDA to identify cross-class similar code fragments in Object Oriented software systems. We built our main research hypothesis upon observations related to the latent topic structure of the software artifacts, and the effect code clones might have on that structure. In particular, we assume that matching on the dominant topic distribution between individual artifacts indicates cloning. To test our research hypothesis, calibrate, and evaluate our approach, we conducted an experimental analysis using four software systems from different application domains. We also compared the performance of our approach with other popular clone detection tools that adopt different clone detection strategies including, *CCFinder* and *CloneDR*. Results show that LDA can achieve satisfactory levels of recall, showing particularly good performance in detecting Type III clones that other related tools usually tend to miss. It also achieves levels of accuracy that can be adequate for practical applications. In the future, we plan to evaluate our approach by testing *CloneTM* using open source software systems to assess the usefulness and the scope of applicability of our approach. Also, we plan to fully implement our finding in the tool and provide visualization support to allow users to visually compare topic distributions of different classes as well as accept or reject candidate clones. We also investigate the potential effect of other code smells, such as God Class or Feature Envy, on the latent topic structure of software artifacts.

7. REFERENCES

[1] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *European conference on Machine Learning*, pages 6–17, 2007.

[2] H. Asuncion, A. Asuncion, and R. Taylor. Software traceability with topic modeling. In *International Conference on Software Engineering*, pages 95–104, 2010.

[3] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *European Conference on Software Maintenance and Reengineering*, pages 81–90, 2010.

[4] B. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering*, pages 86–95, 1995.

[5] I. Baxter, A. Yahin, L. Moura, M. SantAnna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, pages 368–377, 1998.

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions of Software Engineering*, 33(9):577–591, 2007.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*,

3:993–1022, 2003.

[8] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *International Workshop on Source Code Analysis and Manipulation*, pages 36–43, 2002.

[9] S. Ducasse, M. Rieger, and S. Demeyer. Language independent approach for detecting duplicated code. In *International Conference on Software Maintenance*, pages 109–118, 1999.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison–Wesley, 1999.

[11] S. Grant and J. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *International Working Conference on Source Code Analysis and Manipulation*, pages 65–74, 2010.

[12] T. Griffiths and M. Steyvers. Finding scientific topics. In *The National Academy of Sciences*, pages 5228–5235, 2004.

[13] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *International Working Conference on Source Code Analysis and Manipulation*, pages 203–212, 2007.

[14] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *International Conference on Software Engineering*, pages 485–495, 2009.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions Software Engineering*, 28(7):654–670, 2002.

[16] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[17] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Working Conference on Reverse Engineering*, pages 44–54, 1997.

[18] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering*, pages 253–262, 2006.

[19] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering*, pages 301–309, 2001.

[20] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *International Conference on Software Maintenance*, pages 314–321, 1997.

[21] Z. Li, L. Shan, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[22] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides. Modelling class cohesion as mixtures of latent topics. In *International Conference on Software Maintenance*, pages 233–242, 2009.

[23] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using Latent Dirichlet Allocation. In *India software engineering conference*, pages 113–120, 2008.

[24] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *International Conference on Software Maintenance*, pages 244–253, 1996.

[25] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *International Conference on Software Engineeringl*, pages 421–430, 2008.

[26] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *International Conference on Program Comprehension*, pages 68–71, 2010.

[27] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *International Conference on Software Engineering*, pages 522–531, 2013.

[28] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling. Fast collapsed gibbs sampling for Latent Dirichlet Allocation. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 569–577, 2008.

[29] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *MSR*, pages 72–81, 2010.

[30] C. Roy and J. Cordy. A survey on software clone detection research. *Technical Report 541, School of Computing TR 2007-541, Queens University*, 2007.

[31] S. Thomas, B. Adams, A. Hassan, and D. Blostein. Validating the use of topic models for software evolution. In *IEEE Working Conference on Source Code Analysis and Manipulation*, pages 55–64, 2010.

[32] X. Wei and B. Croft. LDA-based document models for ad-hoc retrieval. In *ACM SIGIR conference on Research and development in information retrieval*, pages 178–185, 2006.