An Empirical Study on the Effect of Testing on Code Quality using Topic Models:
A Case Study on Software Development Systems

Tse-Hsun Chen, Stephen W. Thomas, Hadi Hemmati, Meiyappan Nagappan, and Ahmed E. Hassan

***Abstract*—Previous research in defect prediction has proposed approaches to determine which files require additional testing resources. However, practitioners typically create tests at a higher-level of abstraction, which may span across many files. In this paper, we study software testing, especially test resource prioritization, from a different perspective. We use topic models to generate topics that provide a high-level view of a system, allowing developers to look at the test case coverage from a different angle. We propose measures of how well-tested and defect-prone a topic is, allowing us to discover which topics are well-tested and which are defect-prone. We conduct case studies on the histories of Mylyn, Eclipse, and NetBeans. We find that (i) 34 to 78% of topics are shared between source code and test files, indicating that we can use topic models to study testing; (ii) well-tested topics are usually less defect-prone; defect-prone topics are usually under-tested; (iii) we can predict which topics are defect-prone but not well-tested with an average precision and recall of 75% and 77%, respectively; (iv) our approach complements traditional prediction-based approaches by saving testing and code inspection effort; and (v) our approach is not particularly sensitive to the parameters that we use.**

***Index Terms*—Empirical study, code quality, topic models, testing.**

ACRONYMS AND ABBREVIATIONS

LDA	Latent Dirichlet Allocation
CDDT	Cumulative Defect Density of a Topic
MIC	Maximal Information Coefficient
MINE	Maximal Information-based Nonparametric Exploration
MAS	Maximum Asymmetry Score
MEV	Maximum Edge Value
LSI	Latent Semantic Indexing
LOC	Lines of Code
CCDT	Cumulative Churn Density of a Topic
PCA	Principal Component Analysis
PC	Principal Component
QA	Quality Assurance
LM	Linear Regression Model
RTM	Relational Topic Model
MWE	Maximal Weighted Entropy

T. Chen is with the Department of Computer Science and Software Engineering at Concordia University, QC, Canada. (The paper was written when T. Chen was at Queen's University).

S. W. Thomas and A. E. Hassan are with the Software Analysis and Intelligence Lab (SAIL) at Queen's University, ON, Canada.

H. Hemmati is with the University of Calgary, AB, Canada.

M. Nagappan is with the University of Waterloo, ON, Canada.

NOTATIONS

Z	a set of topics
z_i	topic i
f_j	document j
α	Dirichlet priors for Dirichlet distributions
β	Dirichlet priors for Dirichlet distributions
θ	topic membership
R^2	coefficient of determination
II	number of LDA iterations
K	number of topics in LDA
W	topic weight
δ	topic membership threshold

I. Introduction

SOFTWARE defects are costly and time-consuming to fix, spawning much research in defect prediction [1]. However, in practice, software testing is still widely used for detecting defects prior to the release of a software system. Specifically, practitioners create and execute test cases in an effort to uncover as many defects as possible and maximize software quality.

Existing research in software engineering uses code coverage (or test coverage), i.e., the proportion of the source code files that are tested by the test cases, to determine which files or methods should receive additional testing resources. In addition, defect prediction approaches have been augmented with traditional testing criteria, such as statement coverage, path coverage, and function converge [2]–[5]. However, research [6] indicates that testers create test cases from a higher-level view of the system (e.g., features, business logics, or components), instead of individual files.

Thus, in this paper, we propose an approach to study software testing from a different perspective. We use an advanced statistical technique, called topic models, to generate the topics (i.e., co-occurring words) in source code files. A prior study by Baldi *et al.* [7] has shown that the topics generated by topic models have a strong agreement with that of other aspect mining approaches. Their results indicate that these topics can provide developers a high-level view of a system. We posit that if practitioners can identify which topics, as opposed to which files, require more testing, then practitioners can better utilize their limited testing resources. To this end, we propose an approach to automatically discover which topics are present in which source code files and

in which test cases. In particular, we generate topics using latent Dirichlet allocation (LDA) [8], following the success of related research [9]–[13]. We then define new metrics, most notably *topic testedness*, which indicates how well-tested a topic is. We also use *the Cumulative Defect Density of a Topic* (CDDT) [13] to indicate the defect-proneness of a topic.

Our proposed approach enjoys many advantages. First, unlike traditional coverage metrics, topics can span across many files, which may better represent how practitioners view their code as they create test cases. Second, by concentrating on the topics with low testedness and high CDDT, practitioners can allocate their testing resources on the most vulnerable topics of the source code. As topics are linked back to the source code files, practitioners may use our approach to allocate more testing resources on the files related to under-tested and defect-prone topics. Finally, our approach is able to identify defect-prone files that are much smaller in size compared to those identified by traditional approaches.

To evaluate our approach, we perform an in-depth case study on three large, real-world software systems, focusing on the following research questions.

RQ1: To what degree do source code and test files share topics?
To answer this question, we measure the proportion of a topic found in test code versus source code. We find that in all three studied systems, between 34% and 78% of the topics are shared between source code and test files.

RQ2: Are well-tested topics less defect-prone?
Previous research has shown that high code coverage may help improve software quality [14]. We wish to study if this observation also holds when we study code coverage at the topic level. We find that there is a non-coexistence relationship between topic testedness and CDDT: when a topic is well-tested, it is usually less defect-prone; when a topic is defect-prone, it is usually under-tested.

RQ3: Can we identify defect-prone topics that need more testing?
To answer this question, we use a naïve Bayes classifier to perform a cross-release prediction, and identify defect-prone topics that are under-tested. We find that by training on previous releases of a system, we can obtain, on average, a precision of 0.75 and a recall of 0.77 when classifying topics according to their testedness and CDDT in later releases. By linking the predicted topics back to source code files, our approach allows practitioners to allocate additional testing resources more effectively to these parts of the code.

The result of our parameter sensitivity analysis also shows that our approach is not particularly sensitive to the parameters that we use.

The rest of this paper is organized as follows. Section II, describes the background of our approach for discovering topics in source code and test files. We also briefly describe a new correlation measure called Maximal Information Coefficient (MIC) [15], which we use to study the relationship between topic testedness and CDDT. Section III outlines the design of our case studies. Section IV presents the answers to our research questions. Section V studies the parameter and threshold sensitivity of our proposed approach. Section VI discusses the potential threats to the validity of our results. Section VII describes related work. Finally, Section VIII concludes the paper.

We make the datasets and results of our case study publicly available [16] and encourage others to replicate and verify our studies.

	Top words
z_1	*os, cpu, memory, kernel*
z_2	*network, speed, bandwidth*
z_3	*button, click, mouse, right*

(a) Topics (Z).

	z_1	z_2	z_3
f_1	0.3	0.7	0.0
f_2	0.0	0.9	0.1
f_3	0.5	0.0	0.5

(b) Topic memberships (θ).

Fig. 1. Example topic model in which three topics are discovered from three source code files (not shown). (a) The three discovered topics (z_1, z_2, z_3) are defined by their top (i.e., highest probability) words. (b) The three source code files (f_1, f_2, f_3) can now be represented by a topic membership vector.

II. Background

We generate topics using topic models [8], following previous research [7], [11], [17]. In particular, we use the *linguistic data* from each source code and test file, i.e., the identifier names and comments, which helps to determine the functionality of a file [18]. We then treat the linguistic data as a corpus of documents, which we use as a basis for topic models. In this section, we introduce the background knowledge of the approaches that we use to capture the topics and study the relationship between topic testedness and defects.

A. Topic Modeling

(This subsection is taken from our previous work with some modifications [13]). In topic models, a *topic* is a collection of frequently co-occurring words in a corpus. Given a corpus of n documents $f_1, ..., f_n$, topic models automatically discover a set Z of topics, $Z = \{z_1, ..., z_K\}$, as well as the mapping θ between topics and documents (see Figure 1). The number of topics, K, is an input that controls the granularity of the topics. We use the notation θ_{ij} to describe the topic membership value of topic z_i in document f_j.

Intuitively, the top words of a topic are semantically related and represent some real-world concepts. For example, in Figure 1a, the three topics represent the concepts of "operating systems", "computer networks", and "user input". The topic membership of a document then describes which concepts

are present in that document: document f_1 is 30% about the topic "operating systems" and 70% about the topic "computer networks".

More formally, each topic is defined by a probability distribution over all the unique words in the corpus. Given two Dirichlet priors (parameters for Dirichlet distributions), α and β, a topic model generates a topic distribution θ_j for f_j based on α, and generates a word distribution ϕ_i for z_i based on β. The topic membership values define links between topics and source code files. In Figure 1b, the source code file f_1 belongs to topic z_1 and z_2, because its membership values of these two topics are larger than 0. This is because f_1 contains words from topics z_1 and z_2, such as *os, cpu, and network*.

Using the topics and the topic membership vectors, we define topic metrics to help us characterize the topics and answer our research questions. We define metrics as needed in Section IV.

B. Maximal Information Coefficient

Maximal information coefficient (MIC) is an approach that was recently developed to discover different kinds of relationships, such as linear and functional, between pairs of variables [15]. We use MIC later in the paper to measure the relationship between how well a topic is tested and its defect-proneness. Recently, MIC has been shown to be useful in finding relationships in software engineering data [19]. Since the relationship between topic testedness and defects may not be linear, we use MIC to find any possible non-linear relationships in the data that cannot be discovered by the commonly-used Pearson or Spearman correlation coefficients.

MIC provides *generality* (finding many different kinds of relationships), and *equitability* (gives a similar score to different relationships with the same amount of noise). In this way, MIC can be viewed as the coefficient of determination (R^2) in linear regression analysis, except MIC is not limited to a simple linear relationship. Therefore, the interpretation of MIC will be similar to that of R^2 (e.g., a value of 1 represents a perfect relationship). We use the following maximal information-based nonparametric exploration (MINE) measures that are computed from MIC using the tool that Reshef *et al.* [20] provide:

- *MIC - p^2*: a measure of the linearity of the relationship, where p is the Pearson correlation coefficient. Since the MIC value is close to p^2 when the relationship is linear, if MIC - p^2 is close to zero, then the relationship is linear; if MIC - p^2 is close to MIC, then the relationship is non-linear.
- *Maximum Asymmetry Score (MAS)*: a measure of the departure from monotonicity, i.e., a function that preserves a given order. MAS is always less than or equal to MIC. If MAS is very close to MIC, then the relationship is not monotonic; otherwise, the relationship is monotonic.
- *Maximum Edge Value (MEV)*: a measure of the non-functionality, i.e., whether the relationship can pass a vertical line test [21]. If the relationship is functional between two variables, then when one variable changes, the other variable will also change according to some mathematical function. MEV $\leq$ MIC always holds. When MEV is close to zero, the relationship is non-functional; otherwise, the relationship is functional.

TABLE I
STATISTICS OF THE STUDIED SYSTEMS, AFTER PREPROCESSING.

	Lines of source code (K)	No. of source files	No. uniq words in source	No. of test files	No. uniq words in test	Lines of test code (K)	Post-release defects
Mylyn 1.0	107	667	2,902	165	1,363	20	586
Mylyn 2.0	115	761	3,037	173	1,347	21	856
Mylyn 3.0	141	937	3,412	217	1,574	29	409
Eclipse 2.0	797	6,722	12,346	1,011	3,005	237	1,692
Eclipse 2.1	987	7,845	13,691	1,290	4,324	430	1,182
Eclipse 3.0	1,305	10,545	15,726	1,835	5,794	600	2,679
NetBeans 4.0	840	3,874	11,364	502	3,660	95	287
NetBeans 5.0	1,758	7,880	16,219	1,246	5,818	234	194
NetBeans 5.5.1	2,913	14,522	21,397	2,238	8,195	438	739

III. CASE STUDY DESIGN

The purpose of our case study is to answer our three research questions:

1) To what degree do source code and test files share topics?
2) Are well-tested topics less defect-prone?
3) Can we identify defect-prone topics that need more testing?

In this section, we detail the design of our case study. We introduce the systems that we use for our case study, and describe how we identify test files. Then, we describe our data collection and preprocessing steps as shown in Figure 2, and finally outline the specifics of our topic modeling approach.

A. Studied Systems

We use Mylyn, Eclipse, and NetBeans as the three studied systems in our case study (Table I). Mylyn is a popular task management plugin for Eclipse. Eclipse and NetBeans are both popular IDEs, where the former has an extensive plugin architecture, and the latter allows applications to be built from a set of modules. We conduct our case study on three versions of each studied system. We choose these three systems because they all use unit tests, and they differ in size and number of defects. Unit test files are usually large enough to contain enough linguistic data (i.e., identifier names and comments) from which we can extract topics. If the test files were too small (e.g., simple one-line scripts), then our methodology may not have enough information to capture meaningful topics [22].

B. Identifying Test Files

Unfortunately, it is difficult to automatically identify which source code files are in fact test files, as test files are not, in general, explicitly marked. To do so, we use a heuristic similar to the approach proposed by Zaidman *et al.* [23]. First, we check out the entire source code repository. Then, we extract all source code files that have a file path that contains the test keywords *junit* or *test* (the studied systems use the JUnit testing framework). For example, we would

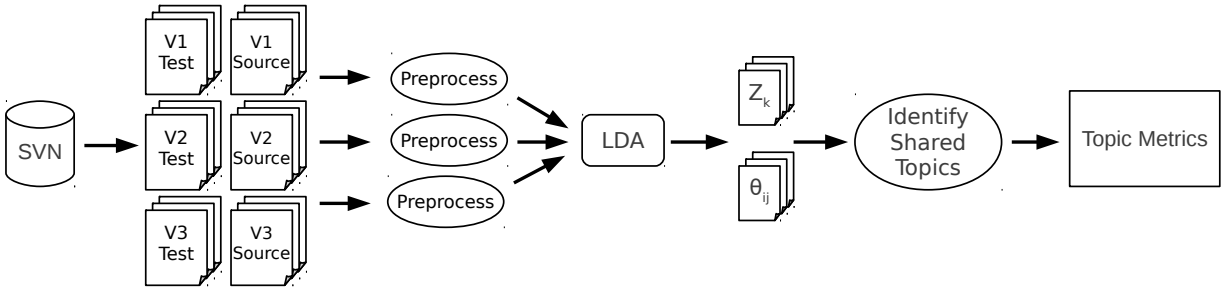


Fig. 2. Our process of applying topic models and calculating topic metrics. We preprocess test file and source code files, then run LDA on *all versions of the data together*. We first obtain shared topics (topics that exist in both source code and test code) using the topics and topic membership values returned by LDA. We then calculate the topic metrics on *shared topics* only.

extract the source code files under the folder *src/test/*, but not *src/UI/*. It is possible that a file may not be related to testing even though its path contains one of these test keywords. For example, in Eclipse 2.0, the source code file *.../core/tests/harness/PerformanceTimer.java* is a utility class that tests use, but is not itself a test case.

To increase the precision of our process for test file identification, we make sure that the name of the source code files also contains at least one of the test keywords. Therefore, a file is identified as a test file if and only if both its name and path contain at least one of the two test keywords, i.e. *"junit"* or *"test"*. Table I shows the number of test files that are identified using this heuristic.

C. Data Preprocessing

We collected the source code and test files from each version of each studied system. Then, we preprocessed the files using the preprocessing steps proposed by Kuhn *et al.* [18]. Namely, we first extracted comments and identifier names from each source code or test file. Next, we split the identifier names according to common naming conventions, such as camel case and underscores. Finally, we stemmed the words and remove common English-language stop words and Java keywords. We further pruned the words that appeared in more than 80% or less than 1% of the source code and test files to eliminate overly common words (e.g., language reserved words) and overly rare words (e.g., typos) [24]. The preprocessing steps may affect the resulting topics, which we discuss in detail in Section VI.

D. Latent Dirichlet Allocation

We used a popular topic model called latent Dirichlet allocation (LDA) [8]. We note that other topic models can be used. We choose LDA because LDA is a generative statistical model, which helps to alleviate model overfitting, compared to other topic models such as Probabilistic LSI [25]. In addition, LDA has been shown to be effective for a variety of software engineering purposes, including analyzing source code evolution [26], calculating source code metrics [27], and recovering traceability links between source code and requirements documents [28]. Recently, Hindle *et al.* [29] show that practitioners with a good general knowledge about a software system can relate the topics generated by LDA to actual code change history, indicating the LDA can produce meaningful and usable topics. Finally, LDA is fast and can easily scale to millions of documents.

We give both the preprocessed source code and test files to LDA (an approach proposed by Linstead *et al.* [12]), so the source code and test files will have the same set of topics for measuring topic testedness. In our studied systems, each test file usually corresponds to one Java file (i.e., unit tests). Due to the nature of the Java files, each file usually corresponds to one class. Thus, we are studying the relationship between the topics in source code and test files at the class level in the paper. We did not conduct our study at the method level, because methods in the same source code/test file are usually related to the same business logic, which would result in having very similar topics. In addition, if we run LDA on the method level, the topics that we get may not be as high quality as the topics that we get by running LDA on the class/file level. The number of words in each method is usually very short, which increases the difficulty for LDA to capture the underlying topics in the methods [8], [30]. Therefore, we choose to conduct our study at the file level.

For this study, we use K=500 topics for all studied systems. Chen *et al.* [13] and Lukins *et al.* [31] found that 500 topics is a good number for Eclipse and Mylyn, and to be consistent, we also choose this number for NetBeans. Moreover, a prior study [30] finds that choosing a larger K does not really affect topic quality. The additional topics will be rarely used during the LDA sampling process, so these additional topics will be considered noise and can be filtered out. However, choosing a smaller K can be more problematic, since the topics cannot be separated properly. Thus, we choose to use a relatively large K for our studies.

We use MALLET [32] as our LDA implementation. MALLET uses Gibbs sampling to approximate the joint distribution of topics and words. We ran MALLET with 10,000 sampling iterations (II), of which 1,000 were used as burn-in iterations to optimize for α and β. In addition, we let MALLET build the topics using both unigrams (single words) and bigrams (pairs of adjacent words) at the same time, since bigrams have been shown to help improve the performance of topic models [33]. Section V studies the parameter and threshold sensitivity of our proposed approach.

IV. Case Study

In this section, we present our results for the three research questions. We divide each question into four parts: motivation, approach, results, and discussion. We first introduce our motivation and describe the approach that we use to solve the research question. Then, we provide our findings and conclude with a more detailed discussion.

A. To what degree do source code and test files share topics? (RQ1)

1) Motivation: In this question, we investigate how topics are shared between source code and test files. To do so, we check whether topics are evenly distributed between these two types of files. If topics are present in only one of the file types (source code or test), then we cannot examine the effect of testing on code quality using topics (i.e., there is no overlap between the topics in them). By determining which topics are shared, we also determine which topics are not shared: those only found in source files (*source-only* topics), and those only found in test files (*test-only* topics). Excluding unshared topics allows us to remove a source of outlier topics (e.g., topics about `assert` are well-tested but less defect-prone, because they are keywords in test files; however these topics are not present in source code files) in our further analysis in RQ2 and RQ3.

2) Approach: To measure the per-topic lines of testing code, we define the test ***weight*** of the topic z_i as:

$$\text{W}_{test}(z_i) = 1/LOC_{test} \sum_{j=1}^{n_{test}} \theta_{ij} * \text{LOC}(test_j), \quad (1)$$

where n_{test} is the total number of test files, *LOC($test_j$)* is the lines of code of *test file* $test_j$, LOC_{test} is the total LOC of all test files, and θ_{ij} is the topic membership of topic z_i in file $test_j$.

Similarly, we define source W_{source}, the weight metric for source code files, as

$$\text{W}_{source}(z_i) = 1/LOC_{source} \sum_{j=1}^{n_{source}} \theta_{ij} * \text{LOC}(source_j), \quad (2)$$

where n_{source} is the total number of source code files, *LOC($source_j$)* is the lines of code of *source code file* $source_j$, LOC_{source} is the total LOC of all source code files, and θ_{ij} is the topic membership of topic z_i in file $source_j$.

We normalize the weight metrics above by LOC because LOC is different between test code and source code files. This normalization helps eliminate possible influences by the size difference between the two file types. We note that although LDA operates based on tokens (i.e., unigrams or bigrams in the test files and source code entities), the correlation between LOC and the number of tokens is very high for all subject systems (above 0.93). Thus, the weight metric is not sensitive to using either LOC or number of tokens.

To find shared topics between source code and test files, we define the ***W ratio*** metric of topic z_i as

$$\text{W}_{ratio}(z_i) = \frac{W_{test}(z_i)}{W_{test}(z_i) + W_{source}(z_i)}. \quad (3)$$

TABLE II
Summary of topics that belong to source code, test, and shared topics. Percentage of shared topics is calculated as the number of shared topics over the total number of topics (500 topics).

	No. of source-only topics	No. of test-only topics	No. of shared topics	% shared topics
Mylyn 1.0	250	73	177	35%
Mylyn 2.0	257	71	172	34%
Mylyn 3.0	245	83	172	34%
Eclipse 2.0	208	33	269	54%
Eclipse 2.1	177	42	281	56%
Eclipse 3.0	141	40	319	64%
NetBeans 4.0	132	27	341	68%
NetBeans 5.0	119	27	354	71%
NetBeans 5.5.1	90	21	389	78%

The W ratio metric is used for removing noise in our dataset. We only use W ratio to identify topics that are prevalent in either source code or test code.

There are two possible approaches to determine source-only topics (e.g., topics about mutators and accessors, or printing) and test-only topics (e.g., topics about `assert`): (i) by manual inspection of the topics; or (ii) by removing topics according to predefined thresholds. To avoid bias that could stem from manual inspection, we use the W ratio values of 0.05 and 0.95 as thresholds to remove topics that are more prevalent in source code or test files (Sections V and VI provide a discussion of how different threshold values may impact our study). Topics that have a W ratio less than 0.05 are more prevalent in source code files; topics with a weighted ratio larger than 0.95 are more prevalent in test files. Although the W ratio value is 0.5 when a topic is perfectly shared between source code and test files, this corner case will not affect our result. These perfectly shared topics will not be removed in our further analysis according to our thresholds. Therefore, the topics that have a W ratio between 0.05 and 0.95 are the *shared topics* that we seek.

3) Results: Figure 3 shows the weighted topic ratio of each studied version of Eclipse. We only show the result for Eclipse here, but the results are very similar for Mylyn and NetBeans. The figure illustrates that topics belong to three different categories: source-only topics (left), test-only topics (right), and shared topics (middle). In all the studied systems of our case study, we find that 34–78% of the topics are being shared between source code files and test files, using our chosen threshold value (Table II). Since a majority of topics are shared, we can study code coverage using topic models.

4) Discussion: To understand what kinds of topics are identified as *unshared topics* (source-only or test-only topics) and what these topics represent, we manually look at the top words of some representative test and source code topics (Table III). The complete information of the generated topics and their metrics values are online [16].

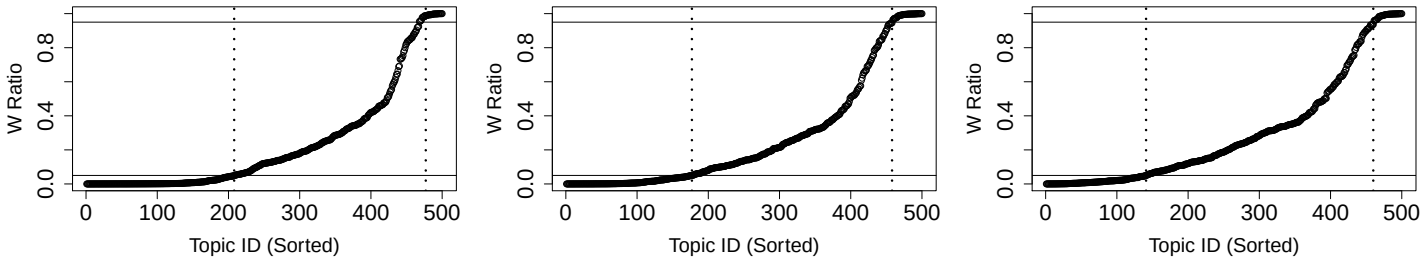
a) **Mylyn:** We find that the source-only topics in Mylyn are about sorting, adding, removing, and monitoring tasks or operations (topics 20, 32, 55). Another source-only topic, Topic 41, is about synchronizing tasks with the user interface.

TABLE III
TOPIC LABEL AND TOP WORDS OF SELECTED TEST/SOURCE-ONLY TOPICS IN OUR STUDIED SYSTEMS. THE NUMBER ON THE LEFT-HAND SIDE OF EACH COLUMN REPRESENTS THE TOPIC NUMBER.

Test-Only Topics		**Source-Only Topics**	
Label	Top Words	Label	Top Words
Mylyn 1.0			
171 eclips debug	configur, launch, test, eclips, junit, launch_configur	20 sort	sort, order, sort_order, action, view, categori
198 program arg	arg, program, configur, program_arg, plugin, add	55 progress monitor	work, progress, progress_monitor total_work, total, tick
Mylyn 2.0			
331 assert enable	enabl, assert_enabl, test_assert, accompani, enabl_test, distribut	41 set synchronize	synchron, task_task, update, manag, data_manag, set
496 structur bridge	web, bridg, recourc, structur_bridg, structur, web_resourc	190 select index	version, repositori, combo, valid, server, repositori_version
Mylyn 3.0			
60 sandbox share data	share, share_data, bob, data, sandbox, folder	32 add task	add, add_task, command, servic, id, task_viewer
444 assert wizard	wizard, histori, assert, size, histori_context, page	282 mylyn task core	core, repositori, repositori_repositori, connector, throw, repositori_attach

Test-Only Topics		**Source-Only Topics**	
Label	Top Words	Label	Top Words
Eclipse 2.0			
101 return qualify	code, test, qualifi, creat, creat_test, code_creat	8 search scope	scope, search, pattern, search_scope, limit, privat
291 assert target exit	file, project, exist, assert, workspac, exit_assert	48 submission handler	servic, submiss, shell, command, bind, activ
Eclipse 2.1			
59 nl	unit, nl, nl_nl, source, type, assert	277 content offset	content, offset, local, attribut, local_content, content_offset
445 assert equal	assert, equal, test, assert_equal, public_test, length	243 gdk color	gdk, gtk, window, gdk_color, style, set
Eclipse 3.0			
424 test suit	test, suit,test_suit, add, test_test, add_test	12 print	print, packet, id, command, stream, println
468 item assert equal	select,test,equal, assert, assert_equal, number	124 handl gtk	gtk, handl, widget, handl_gtk, gtk_widget, signal

Test-Only Topics			**Source-Only Topics**		
	Label	Top Words		Label	Top Words
NetBeans 4.0					
0	test suit	test, suit, test_suit, junit, nb, nb_test	18	content pane	pane, set, layout, add, pane_add, border
295	property test	test, exclud, testbag, config, includ, set	470	mutator method	method, pattern, properti, setter, set, getter
NetBeans 5.0					
38	cvs test	cvsroot, test, set, cv, cvss, crso	33	text area	area, text, text_area, jtext_area, area_set, set
390	test editor	test, action, editor, node, test_editor, netbean_test	182	paint component	paint, compon, paint_compon, draw, item, substitut
NetBeans 5.5.1					
10	jframe test	test, frame, assert, jframe, set, visibl	106	slide bar	slide, tab, bar, bound, slide_bar, compon
79	perform test	test, perform, perform_test, pass, method, pass_test	125	mdb	password, mdb, factori, connect_factori, connect, set



(a) Eclipse 2.0. (b) Eclipse 2.1. (c) Eclipse 3.0.

Fig. 3. The W ratio of test topics in source code files in Eclipse. The dashed lines indicate the cut-off thresholds of 0.05 and 0.95. There are three different categories of topics shown in the figures: source-only topics (left), test-only topics (right), and shared topics (middle).

Topics 190 and 282 are about accessing source code repositories such as Bugzilla. Topics 171 and 331, both test-only topics, are about test keywords in Mylyn. We look at test files related to topic 198, and find that they are about launching JUnit plugin tests; for example, launching Mylyn Plug-in Development Environment UI configuration tests. Other test-only topics are about testing hypertext structure bridging (topic 496), testing shared task directory (topic 60), and testing task import wizard (topic 444).

b) **Eclipse:** Source-only topics in Eclipse include topics about searching (topic 8), GUI (topics 124 and 243), converting packets to human readable form (topic 12), key binding services (topic 48), and servlets to interface client (topic 277). In contrast, the test-only topics in Eclipse all contain some test keywords, such as "test", "create test", or "test suit".

c) **NetBeans:** NetBeans has source-only topics about GUIs (topics 18, 33, 182, and 106), mutator and accessor functions (topic 470), and message-driven bean initialization (topic 125). Test-only topics in NetBeans are about different kinds of software components testing, such as editors (topic 390) and CVS (topic 38).

34%–78% of the topics are shared between source code and test files, implying that we can study topic testedness by comparing the prevalence of a topic in these two types of files. In addition, we find that the unshared topics are mostly about keyword topics (e.g., `assert` *in test files) that only exist in either source code or test files.*

B. Are well-tested topics less defect-prone? (RQ2)

1) Motivation: In this question, we want to understand the relationships between how tested a topic is and its cumulative defect density. A previous study has shown that high code coverage may increase software quality [14]; however, we want to study if this observation also holds when we study code coverage at the level of *topics*. The difference between our topic-based approach and other traditional code coverage approaches is that traditional code coverage approaches can only tell us whether a statement, a branch, or a method call, etc., has been covered by the tests or not. On the other hand, topics can span across many files and provide a high-level view of the system. For example, to see if "network event handling" is well-tested, code coverage would not help much, since this feature is implemented in several statements across several methods. Our approach shows how well-tested a topic is, so our approach is not bounded to either statements, methods, or classes. We hypothesize that topics that are well-tested will be less defect-prone, and topics that are under-tested will be more defect-prone. If the hypothesis holds, then practitioners can focus on testing source code files related to defect-prone topics to improve code quality.

2) Approach: Researchers have proposed different metrics to capture the code coverage of test files. In this paper, we want to study code coverage at the abstraction level of topics, and examine the effect of *topic testedness* (i.e., how well-tested is a topic) on the cumulative post-release defect density for each topic. The post-release defects are the defects that are reported within six months after the version is released [34]. We define the ***testedness*** of topic z_i as

$$\text{Testedness}(z_i) = \frac{\text{W}_{test}(z_i)}{\text{W}_{source}(z_i)}, \tag{4}$$

where $\text{W}_{test}(z_i)$ and $\text{W}_{source}(z_i)$ are the computed weight metrics (Equation 1 and 2) for test and source code files, respectively. We normalize the weight metrics according to the total lines of code in test files and source code files to take the size difference between both file types into the account. The intuition behind this metric is that if a topic is more prevalent in the test files, then the topic is well-tested.

Chen *et al.* [13] proposed a metric to capture the *Cumulative Defect Density of a Topic* (CDDT). CDDT is calculated by first computing the weighted defect density for all topics in *each file*. Then, CDDT cumulates the weighted defect density of the topic across all files. We use this metric to quantify the cumulative *post-release defect* density of each topic.

In summary, the ***cumulative effect defect density of a topic (CDDT$_{post}$)*** is calculated as

$$\text{CDDT}_{post}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \left(\frac{\text{POST}(f_j)}{\text{LOC}(f_j)} \right), \tag{5}$$

where *POST(f_j)* is the number of post-release defects of file f_j, which are those defects found up to six months after the release date of a given version, LOC(f_j) is the lines of code in source code file f_j, and θ_{ij} is the topic membership of topic z_i in file f_j. The six-month period for determining post-release defects is used by other studies to study software quality (e.g.,

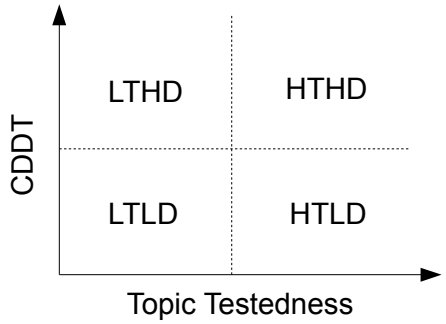


Fig. 4. Position of each class on the scatter plot. The bottom left corner has low testedness and low CDDT$_{post}$ (LTLD) topics. The upper left corner has low testedness and high CDDT$_{post}$ topics. The rest of topics have high testedness and low CDDT$_{post}$ (HTLD). There is only a few topics that have high testedness and high CDDT$_{post}$ (HTHD).

[34]). Note that we also exclude all the unshared topics as identified in RQ1.

We classify topics into four different classes as shown in Figure 4:

- Class LTHD: low testedness and high CDDT$_{post}$
- Class LTLD: low testedness and low CDDT$_{post}$
- Class HTLD: high testedness and low CDDT$_{post}$
- Class HTHD: high testedness and high CDDT$_{post}$

We define the level of testedness and CDDT$_{post}$ (i.e., low or high) relative to the quartiles of both metrics. We classify topics that have a testedness and CDDT$_{post}$ value smaller than the *third quartile (75%)* of all topics to be class LTLD. Topics that have a CDDT$_{post}$ value larger than or equal to the third quartile and have a testedness value smaller than the third quartile are classified as class LTHD. Class HTLD includes the topics that have high testedness (larger than the third quartile) but have lower CDDT$_{post}$ (smaller than the third quartile). Finally, there are a few topics that are classified as having high testedness and high CDDT$_{post}$ by our approach (HTHD).

We use the MIC score and MINE measures (see Section II-B) to describe and verify the relationships that we observe in the scatter plots.

3) Results: Table IV summarizes the number of topics in each class. Class LTHD has the fewest number of topics among all classes, except HTHD, and class LTLD has significantly more topics than other classes. This implies that there exist less defect-prone topics that are under-tested (LTHD), and it would be desirable to focus testing on these topics.

Appendix A shows the scatter plots of the relationship between topic testedness and CDDT$_{post}$ for each version of each system. Due to the approach that we use to automatically classify the topics, topics in HTHD are usually at the boundary between LTHD and HTLD. The topics in other classes are distributed along the X and Y axes.

Table V shows the MINE and MIC scores, indicating that there is a *non-coexistence* relationship between testedness and CDDT$_{post}$ [15]. In our case, this relationship indicates that when a topic is well-tested, it is more likely to be less defect-prone.

TABLE IV
NUMBER OF TOPICS IN EACH CLASS.

	LTHD	LTLD	HTLD	HTHD
Mylyn 1.0	38	94	38	6
Mylyn 2.0	39	90	39	4
Mylyn 3.0	39	90	39	4
Eclipse 2.0	52	142	52	13
Eclipse 2.1	60	150	60	11
Eclipse 3.0	72	167	72	8
NetBeans 4.0	72	183	73	13
NetBeans 5.0	60	205	60	29
NetBeans 5.5.1	72	219	73	25

TABLE V
SCORES OF MINE METRICS COMPUTED BETWEEN TOPIC TESTEDNESS AND THE CUMULATIVE POST-RELEASE DEFECT DENSITY OF A TOPIC (CDDT$_{post}$).

	MIC	MIC-p^2	MAS	MEV	p
Mylyn 1.0	0.54	0.51	0.16	0.54	-0.16
Mylyn 2.0	0.56	0.53	0.16	0.56	-0.16
Mylyn 3.0	0.36	0.31	0.11	0.36	-0.22
Eclipse 2.0	0.20	0.18	0.02	0.19	-0.12
Eclipse 2.1	0.21	0.20	0.02	0.21	-0.11
Eclipse 3.0	0.24	0.21	0.08	0.24	-0.16
NetBeans 4.0	0.24	0.22	0.07	0.24	-0.13
NetBeans 5.0	0.20	0.20	0.02	0.20	-0.02
NetBeans 5.5.1	0.19	0.19	0.03	0.19	-0.08

A non-coexistence relationship happens when one variable is more dominant, and the other variable is less dominant (only one can have a high value). Topics in class HTLD have a relatively low CDDT$_{post}$ compared to the other two classes. In addition, topics in class LTHD always have the highest CDDT$_{post}$ amongst all topics, and their testednesses are also low. Although there are some outliers (HTHD), where a few relatively well-tested topics have a higher CDDT$_{post}$ value, most topics follow the non-coexistence pattern: *when a topic is defect-prone, it is usually under-tested; when a topic is well-tested, its CDDT$_{post}$ is usually low.* Topics in class LTLD are usually related to configuration tasks or trivial operations, which are usually under-tested. However, topics in LTLD also exhibit low CDDT$_{post}$, so we are not interested in testing these topics.

In order to support our argument that the discovered relationships are non-coexistence and not random, we check the linearity, monotonicity, and functionality of the relationship between two variables (Table V). In all studied systems, the p values are all negative, and the differences between MIC and MIC$-p^2$ are very small. This indicates that the relationship between topic testedness and CDDT$_{post}$ is non-linear [15]. For example, a not well-tested topic can be either in LTHD or LTLD, since testedness and CDDT$_{post}$ are not directly proportional to each other. In addition, the relatively small MAS values imply that the relationship is monotonic, which provides additional evidence that the relationship is non-coexistence; i.e., when one variable increases, the other variable also increases/decreases accordingly. For example,

TABLE VI
TOP WORDS, TESTEDNESS, AND CDDT_{post} OF SELECTED TOPICS FROM EACH CLASS OF TOPICS.

	Top words	Testedness	Median of Testedness	CDDT_{post}	Median of CDDT_{post}
class LTHD: Mylyn 1.0					
417	task, mylar, eclips, eclips_mylar, mylar_task	1.05	1.05	0.249	0.002
class LTLD: Eclipse 2.1					
494	sheet, cheat, cheat_sheet, resourc, properti_sheet	0.22	0.343	<0.001	0.008
class HTLD: NetBeans 5.5.1					
206	frame, jintern, jintern_frame, intern, intern_frame, pane	2.40	0.644	<0.001	0.002
class HTHD: Eclipse 3.0					
206	breakpoint, debug, ijava, thread, core, suspend	2.65	0.387	0.12	0.002

when testedness increases, CDDT_{post} decreases. The values for MEV are all very close to MIC, which means that there is a functional relationship. For example, CDDT_{post} can be computed as some function of testedness. Although the relationship is non-linear, negative p values indicate that when the value of testedness increases, CDDT_{post} decreases (negative correlation). Taken in aggregate, the results in Table V support our hypothesis that there is a non-coexistence relationship between topic testedness and CDDT_{post} (well-tested topics are usually less defect-prone), and this relationship is functional and not random.

4) Discussion: We list one concrete example of a topic in each class from the studied systems in Table VI. We note that there are many other examples in each case study system for each class. We have posted the data online, and interested readers can interpret and verify the topics [16]. The topic on task-related actions and Eclipse integration is more defect-prone in Mylyn 1.0. As we can see from Table VI, the testedness of this topic is also very low, and it is classified as class LTHD. This topic corresponds to tasks management in Mylyn.

The stack map frame topic in NetBeans 5.5.1 is a topic that is well-tested and has a very low CDDT_{post}. It is therefore classified as class HTLD. This topic corresponds to one of the basic operations in NetBeans for controlling the stack, on which many higher-level functions rely. The topic on property sheets is used for displaying program properties to developers in Eclipse IDE. This is not a core functionality in Eclipse. Hence it neither requires much testing nor does it exhibit many defects. Debugging related functionality in Eclipse is tested more, but it also has a relatively high CDDT_{post}.

We find that there is a non-coexistence relationship between testedness and CDDT_{post}: when a topic is well-tested, it is usually less defect-prone; when a topic is defect-prone, it is usually not well-tested. We verify that this relationship exists and is not random using the MIC score and MINE measures.

C. Can we identify defect-prone topics that need more testing? (RQ3)

1) Motivation: In RQ2, we see evidence that well-tested topics are usually less defect-prone. Therefore, it will be beneficial to know which topics require more testing in future releases. Topics that are less tested can be classified into two categories: high defect-prone and low defect-prone. If we could automatically identify those topics that are under-tested and more defect-prone, then practitioners could put more testing resources on these topics, or the source code files that are related to these topics (we can link topics back to the source code files). By allocating the testing resources more effectively, we can reduce the time and cost in the testing phase before releases. Even though topics in class LTLD are under-tested, we are not interested in them, since they already have a low CDDT_{post} value. By avoiding further testing on topics in class LTLD, we can avoid the unnecessary allocation of testing resources.

In the research field of defect prediction, some researchers predict the number of defects in a file, whereas others predict if a file is defect-prone or not. Similarly, in this paper, we have made the choice to predicting whether a topic is in LTHD instead of predicting the actual values. We do this because the topic testedness measure is (a) continuous in value and not categorical, and hence the statistical models would implicitly be more erroneous (i.e., data is highly skewed and cannot be modelled by normal distribution), and (b) the results in terms of categories like LTHD are more intuitive for the developer to interpret than a topic testedness value.

2) Approach: We build a classifier to predict the class (i.e., class LTHD, LTLD, or HTLD) to which a topic belongs *across versions* (e.g., train the classifier on 2.0 and test on 2.1). We exclude the topics in HTHD in our analysis, because the main focus of this research question is to predict the topics in LTHD. In addition, HTHD has a much smaller number of topics, which may affect the overall quality of the classifier [35]. We use the following topic metrics as the independent variables (i.e., features) in the classifier: topic weight, support, scatter, and the cumulative churn density of a topic (CCDT). These topic metrics are defined below.

The ***weight*** of a topic measures the total lines of code in the topic as

$$\text{Weight}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \text{LOC}(f_j), \tag{6}$$

where *LOC(f_j)* is the lines of code of file f_j. Since size is a well-known predictor for software defects, we include this metric in the classifier.

The ***support*** of a topic measures how many files contain the topic, and is defined as

$$\text{Support}(z_i) = \sum_{j=1}^{n} I(\theta_{ij} \geq \delta), \tag{7}$$

where I is the indicator function that returns 1 if its argument is true, and 0 otherwise. We set the membership threshold δ in Equation 7 to 1% in order to remove insignificant topics in files when computing the support metric for each topic. A

more detailed analysis on this threshold value is presented in Section V.

The ***scatter*** of a topic measures how spread out the topic is across all source code files, based on the information entropy of the topic memberships values. Both Thomas *et al.* [17] and Baldi *et al.* [7] use this metric to measure topic scattering. We define the scatter of a topic z_i as

$$\text{Scatter}(z_i) = -\sum_{j=1}^{n} \log(\theta_{ij}) * \theta_{ij}. \quad (8)$$

Scatter is a measure of the level of coupling of a topic. If a topic is highly scattered, the topic is implemented across many different source code files, which increases maintenance difficulty.

Cumulative Churn Density of a Topic (CCDT) measures the number of changes (i.e., added and modified changes) per lines of code that is made to the part of files related to the topic. We sum up the churn density for each topic across all files to obtain the cumulative churn density of a topic. We define CCDT as

$$D_{\text{CHURN}}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \left(\frac{\text{CHURN}(f_j)}{\text{LOC}(f_j)} \right), \quad (9)$$

where *CHURN(f_j)* is the total number of prior changes to source code file f_j before release. Previous research has shown that if the code in a topic is changed more often, then this topic is more likely to be defect-prone [36].

The above-mentioned metrics measure the structure and history of a topic, which may affect the testing and maintenance practice of a topic, and its CDDT.

Previous studies have shown that a naïve Bayes classifier is an effective algorithm for defect prediction [37], [38], thus we use naïve Bayes in this paper. We train the classifier on older releases, and predict on newer releases. For example, we train our classifier using Eclipse 2.0, and predict the class in which the topics in Eclipse 2.1 belong. We aim to predict to which topics should the testers allocate more testing resources. To avoid the problem of having highly correlated independent variables in the classifier, we use principal component analysis (PCA) to transform these variables into a new set of uncorrelated variables [37]. We select the principal components (PCs) until either 90% of the variances are explained, or when the increase in variance explained by adding a new PC is less than the mean variance explained of all PCs [39].

Since we are interested in finding the defect-prone topics that require more testing, we only report the precision and recall for classifying topics in this class (i.e., class LTHD). Our goal is not predicting defects, but rather, helping practitioners allocate testing resources more effectively.

3) Results: We show the classification results in Table VII. In all the studied system, we obtain, on average (computed across all versions of the studied systems), a precision of 0.75 and a recall of 0.77, which means that most of the under-tested and defect-prone topics are classified correctly. Mylyn has the best classification result among three studied systems, possibly because Mylyn has a larger proportion of test files. In Table I, we can see that although Mylyn is the smallest system among the three, it has relatively more test files. NetBeans, on the other hand, has the fewest proportion of test files, which also yields the lowest F-measure.

TABLE VII
PRECISION, RECALL, AND F-MEASURE FOR CLASSIFYING UNDER-TESTEDNESS AND DEFECT-PRONE TOPICS (CLASS LTHD).

Train on	Test on	Precision	Recall	F-measure
Mylyn 1.0	Mylyn 2.0	0.82	0.92	0.87
Mylyn 2.0	Mylyn 3.0	0.78	0.79	0.78
Eclipse 2.0	Eclipse 2.1	0.79	0.63	0.70
Eclipse 2.1	Eclipse 3.0	0.84	0.79	0.81
NetBeans 4.0	NetBeans 5.0	0.65	0.60	0.62
NetBeans 5.0	NetBeans 5.5.1	0.64	0.86	0.73

Since we obtain high precision and recall values for all studied systems, practitioners may use our approach to reliably identify those defect-prone topics that require more testing. Then, practitioners can allocate more testing resources on these specific topics, and improve the overall code quality.

4) Discussion: The results above show that we can predict which topics are under-tested and defect-prone. By focusing on these topics, the QA resources may be allocated more efficiently. Further, to show that our approach can be used to complement current prediction-based resource allocation approaches (i.e., statistical models that help allocate testing resources by predicting which parts of the system are more defect-prone) for finding defects, we perform an additional experiment that compares our approach with existing approaches.

Namely, we use our prediction model to identify LTHD topics, resulting in a list of possibly-defect-prone topics. We link these topics to their corresponding source code files using a topic membership value of 0.5. If a source code file has a membership value larger than 0.5 in any of the LTHD topics, then this file belongs to LTHD. We choose the relatively high value of 0.5 because it helps us find the source code files that truly belong to the topic (a file can only belong to one topic with a membership value > 0.5), and it helps limit the number of linked files that must be examined by testers.

To compare our approach with the prediction-based resource allocation approaches, we build a linear regression model (LM) as a baseline for comparison. We use code churn and LOC as the independent variables in the model and predict the number of defects [40]. Previous studies have used prediction models to predict the defect-proneness of source code files to help allocate software quality assurance effort [41], [42]. We select the top n files predicted by our approach and the top n files selected by LM, and examine their similarities and differences. (n is determined by the number of files that belong to LTHD, which is fixed for each system.)

We choose LOC and churn as our baseline metrics for LM due to several reasons. Although LOC may not represent all static metrics, LOC is shown to be a good general software metric and has been used for benchmarking prior proposals of new metrics [43], [44]. In addition, LOC is shown to have a high correlation with other complexity metrics [45], [46]. We choose code churn as the baseline for change-related metrics because it has been shown to be a good explainer for

defects [47], [48], and has also been used as baseline models for comparing metrics [40]. Moreover, LOC and code churn are shown to have the best explanatory power for defects, and are typically used as baseline metrics by other researchers when proposing new metrics [43], [46], [49], [50].

Table VIII shows the median LOC, defect density, and percentage of overlap between the source code files predicted by our approach and the LM model. In all studied systems, our approach identifies files with fewer LOC and higher overall defect densities. As an example, a relatively small file in Eclipse 3.0 (*ant/internal/ui/editor/model/AntDefiningTaskNode.java*, which has 71 lines of code) is identified as defect-prone by our topic-baed approach, while LM does not. The defect in this file is related to linkage issues, and using information such as file size, complexity, or churn may not capture the defect [51]. According to the bug report, by providing more test cases, this defect was successfully located and fixed. In our study, our approach outperforms the traditional approach (LM) in terms of saving effort in code inspection and testing.

We note that the actual number of defects in the files identified by LM is higher (because the sheer size of each file is much larger). However, our approach can serve to complement LM, since the overlap between the files identified by our approach and LM is small (0–6.2%), and our approach identifies much smaller files. In addition, developers are already aware that larger files are usually more defect-prone and thus require more testing. This means that our approach can help developers also find the smaller files that require more testing. By using the two approaches in concert, developers and testers may locate additional defects.

We can predict defect-prone and under-tested topics with an average precision and recall of 0.75 and 0.77, respectively. In addition, we find that our approach outperforms traditional prediction-based resource allocation approaches in terms of saving testing and code inspection effort. Our approach is able to identify parts of the systems that are defect-prone and under-tested, which may help practitioners allocate testing resources more effectively.

V. Parameter Sensitivity Analysis

Our approach, detailed in Section II, involves the choice of several input parameters. Each parameter may influence the results of our case study. In particular, LDA takes four parameters as input: number of topics (K), number of iterations (II), and two Dirichlet priors used for smoothing (α and β). In addition, we define three other parameters: W ratio (used to remove noise and determine shared topics), δ (used in Equation 7), and PCA cut-off (used to determine the number of PCs in the naïve Bayes classifier).

We perform a sensitivity analysis to determine how sensitive our results are to our particular parameter value choices. We define a *baseline* set of parameter values, which are the values used in the aforementioned three research questions. In particular, the baseline values are: K=500, II=10,000, W cut-off=0.05, δ=0.01, and PCA cut-off=90%. Since we use the α and β that are automatically optimized by MALLET, we do not change these two values [32]. We increase and decrease each parameter value independently to study the sensitivity of each parameter. We also provide more variance to the LDA parameters due to its probabilistic property [52]. We decrease K to 300 and 400, and increase K to 600 and 700. We decrease II to 8,000 and 9,000, and increase II to 11,000 and 12,000. For δ, we consider $\delta/2$ and $\delta * 2$, which are 0.005 and 0.02. Finally, we consider two different PCA cut-off thresholds, which are 80% and 95%.

For each parameter, we repeat the experiments in RQ2 and RQ3, and report the MIC score from RQ2, and the precision and recall from RQ3. Table IX shows only the parameters that may influence the MIC score, since the topics will be slightly different when K, II, and W cut-off change (the other two parameters cannot change the MIC score, since they do not change the topics). Table X shows the precision and recall of all the parameters that may influence the classification result. Note that, since we are doing cross-release prediction, the precision and recall of the first version of each studied system are not available.

In this paper, we follow the guidelines of previous work [13], [31] for choosing K=500 for Eclipse and Mylyn. We also used K=500 for NetBeans, since the size of NetBeans is similar to that of Eclipse (Table I). We see from Table IX that in most cases, when K, W cut-off change, and II change, the MIC scores remain relatively stable. The MIC score in Mylyn is not as stable as the other two studied systems, since the number of shared topics is less in Mylyn (i.e., we have less data points). We find that changing K varies the MIC score by an average of 16% from the baseline value; changing II varies the MIC score by an average of 12%; and changing W cut-off varies the MIC score by an average of 8.8%. These three parameters determine the number of data points (topics) and topic accuracy in the dataset, which has a direct effect on the overall relationship between topic testedness and CDDT$_{post}$. In addition, since changing K directly changes the number of data points (i.e., number of topics), varying K has a larger effect on MIC. One thing to note is that the MINE measures of all the systems follow the same pattern as described in RQ2, which supports our hypothesis that the relationship is non-coexistence.

Changing the parameters may influence the MIC score. However, the resulting MIC scores and MINE measures are still relatively stable, and the observed patterns (i.e., non-coexistence) are preserved.

From Table X we can see that when K, W cut-off, and II change, the prediction results do not vary much. Changing K varies the precision and recall by an average of 8% from the baseline value; changing II varies the precision and recall by 2% and 3%; and changing W cut-off varies the precision and recall by 3%. We find that the results are also relatively insensitive to the threshold δ: changing δ varies both the precision and recall by only 1% and 0%. We find that changing

TABLE VIII
MEDIAN LOC, DEFECT DENSITY, AND PERCENTAGE OVERLAP OF THE *source code files* THAT ARE PREDICTED TO BE IN CLASS LTHD AND THE MOST DEFECT-PRONE FILES PREDICTED BY THE LINEAR REGRESSION MODEL. THE NUMBERS IN THE PARENTHESES INDICATE THE PERCENTAGE IMPROVEMENT OF OUR APPROACH OVER LM IN TERMS OF THE DEFECT DENSITY OF THE RETURNED FILES. *% files* IS THE PERCENTAGE OF THE SOURCE CODE FILES IN A SYSTEM THAT IS USED FOR COMPARING THE TWO APPROACHES.

Predicted System	Prediction Approach	% files (n)	Median LOC	Defect Density (Defect/KLOC)	% overlapping files
Mylyn 2.0	Topic-based LM	1.9 %	57.0 718.0	11.64 (+62%) 7.20	0 %
Mylyn 3.0	Topic-based LM	0.7 %	50.0 947.0	6.90 (+128%) 3.03	0 %
Eclipse 2.1	Topic-based LM	4.2 %	23.0 450.0	4.08 (+274%) 1.09	1.8 %
Eclipse 3.0	Topic-based LM	5.9 %	22.0 633.0	5.76 (+210%) 1.86	2.1 %
NetBeans 5.0	Topic-based LM	5.2 %	78.0 447.0	0.39 (+56%) 0.25	2.0 %
NetBeans 5.5.1	Topic-based LM	12.5%	64.0 376.0	0.48 (+4%) 0.46	6.2 %

the PCA cut-off value has no effect on precision and recall, since we only have a few independent variables in the naïve Bayes classifier and the cut-off results in the same number of selected PCs.

Table X also shows that Mylyn 2.0 has the best result when K is 700, and Mylyn 3.0 has the best result when K is 500. For larger systems like Eclipse and NetBeans, the result we get when K is 700 do not differ much from our baseline (when K is 500). We even find that when K is 700, our approach gives a worse result compared to the baseline. Since Mylyn is a smaller system compared to other studied systems, using a smaller K may be more intuitive; however, the precision and recall are not significantly better (or may even be worse) when we change K to a smaller number. Our sensitivity analysis shows that the ratio between K and the size of the system may not have a considerable impact on the performance of our approach. A prior study by Hindle *et al.* [53] has found that text in the source code is much more repetitive than natural language text. In addition, topic models are very good at finding words with similar meanings (synonymy) [18], which may explain why using the same K for Mylyn yields comparable performance to NetBeans and Eclipse. Future research should explore the use of other variants of topic models (e.g., [18], [25], [27], [52], [54]).

The precision and recall are consistent for all systems (change $\leq 8\%$, on average) when any of the parameters are increased and decreased.

VI. POTENTIAL THREATS TO VALIDITY

In this section, we discuss the potential threats to validity of our approaches.

A. Construct Validity

1) Parameter and Threshold Choices: Since it is possible that the source-only topics that are excluded in RQ1 belong to the class LTHD, we study how many such topics there are and how defect-prone they are (Table XI). We find that most of the excluded topics do not belong to LTHD (81%–94%), and these topics have a low overall CDDT$_{post}$ value. Therefore, these topics do not significantly affect our results.

TABLE XI
SUMMARY OF THE EXCLUDED SOURCE-ONLY TOPICS THAT BELONG TO THE CLASS LTHD, AND THE MEDIAN CDDT$_{post}$ OF THESE TOPICS.

System	% Source-only Topics that are in LTHD	Median CDDT$_{post}$
Mylyn 1.0	16 %	$3.9 * 10^{-3}$
Mylyn 2.0	16 %	$6.3 * 10^{-3}$
Mylyn 3.0	7 %	$1.8 * 10^{-3}$
Eclipse 2.0	19 %	$6.9 * 10^{-3}$
Eclipse 2.1	8 %	$4.9 * 10^{-3}$
Eclipse 3.0	11 %	$8.4 * 10^{-3}$
NetBeans 4.0	6 %	$5.1 * 10^{-6}$
NetBeans 5.0	8 %	$3.5 * 10^{-6}$
NetBeans 5.5.1	8 %	$2.9 * 10^{-4}$

We use the third quantile of topic testedness and CDDT$_{post}$ to classify topics into three classes. However, this threshold can be changed, and more advanced clustering algorithms may be used. To study the effects of this classification threshold on the results, we change the threshold from the third quantile to median and to the 90% quantile, and repeat our study in RQ3. Namely, we study the precision and recall of predicting LTHD topics across versions and report the defect density of the *files* that are classified as in LTHD. Table XII shows the prediction result when the threshold is changed to median. Since the class imbalance reduces when we change the threshold to median, the classification accuracy increases (except for Eclipse 3.0). However, since now LTHD includes some topics that were previously classified as other classes (e.g., LTLD), the overall defect density of the files that are identified by our approach decreases (when compared to Table VIII).

Table XIII shows the prediction result when the threshold is changed to the 90% quantile. Since the number of LTHD

TABLE IX

RESULTS OF THE PARAMETER SENSITIVITY ANALYSIS OF THE PARAMETERS THAT MAY INFLUENCE *the MIC score*. THE BASELINE PARAMETERS AND THEIR VALUES ARE SHOWN IN THE TABLE. FOR EACH SYSTEM, WE SHOW THE MIC SCORE WHEN THE PARAMETER CHANGES. THE VALUES IN THE PARENTHESES INDICATE THE INCREASE/DECREASE FROM THE BASELINE MIC SCORE.

Baseline Values: K=500, II=10,000, W cut-off=0.05			
Lower		*Higher*	
New Value	MIC	New Value	MIC
Mylyn 1.0 (Baseline MIC = 0.54)			
$K=^{300}_{400}$	0.32 (-0.22) 0.38 (-0.16)	$K=^{600}_{700}$	0.40 (-0.14) 0.76 (+0.22)
$II=^{8K}_{9K}$	0.49 (-0.05) 0.57 (+0.03)	$II=^{11K}_{12K}$	0.53 (-0.01) 0.47 (-0.07)
W cut-off=0.025	0.46 (-0.08)	W cut-off=0.1	0.63 (+0.09)
Mylyn 2.0 (Baseline MIC = 0.56)			
$K=^{300}_{400}$	0.38 (-0.18) 0.46 (-0.10)	$K=^{600}_{700}$	0.40 (-0.16) 0.68 (+0.12)
$II=^{8K}_{9K}$	0.49 (-0.07) 0.62 (+0.06)	$II=^{11K}_{12K}$	0.53 (-0.03) 0.53 (-0.03)
W cut-off=0.025	0.52 (-0.04)	W cut-off=0.1	0.63 (+0.07)
Mylyn 3.0 (Baseline MIC = 0.36)			
$K=^{300}_{400}$	0.30 (-0.06) 0.30 (-0.06)	$K=^{600}_{700}$	0.28 (-0.08) 0.43 (+0.07)
$II=^{8K}_{9K}$	0.31 (-0.05) 0.26 (-0.10)	$II=^{11K}_{12K}$	0.31 (-0.05) 0.34 (-0.02)
W cut-off=0.025	0.33 (-0.03)	W cut-off=0.1	0.30 (-0.06)
Eclipse 2.0 (Baseline MIC = 0.20)			
$K=^{300}_{400}$	0.25 (+0.05) 0.25 (+0.05)	$K=^{600}_{700}$	0.21 (+0.01) 0.28 (+0.08)
$II=^{8K}_{9K}$	0.22 (+0.02) 0.22 (+0.02)	$II=^{11K}_{12K}$	0.19 (-0.01) 0.22 (+0.02)
W cut-off=0.025	0.21 (+0.01)	W cut-off=0.1	0.21 (+0.01)
Eclipse 2.1 (Baseline MIC = 0.21)			
$K=^{300}_{400}$	0.27 (+0.06) 0.18 (-0.03)	$K=^{600}_{700}$	0.20 (-0.01) 0.25 (+0.04)
$II=^{8K}_{9K}$	0.27 (+0.06) 0.22 (+0.01)	$II=^{11K}_{12K}$	0.20 (-0.01) 0.24 (+0.03)
W cut-off=0.025	0.25 (+0.04)	W cut-off=0.1	0.22 (+0.01)
Eclipse 3.0 (Baseline MIC = 0.24)			
$K=^{300}_{400}$	0.28 (+0.04) 0.25 (+0.01)	$K=^{600}_{700}$	0.23 (-0.01) 0.21 (-0.03)
$II=^{8K}_{9K}$	0.25 (+0.01) 0.21 (-0.03)	$II=^{11K}_{12K}$	0.20 (-0.04) 0.24 (—)
W cut-off=0.025	0.25 (+0.01)	W cut-off=0.1	0.22 (-0.02)
NetBeans 4.0 (Baseline MIC = 0.24)			
$K=^{300}_{400}$	0.20 (-0.04) 0.19 (-0.05)	$K=^{600}_{700}$	0.25 (+0.01) 0.26 (+0.02)
$II=^{8K}_{9K}$	0.24 (—) 0.23 (-0.01)	$II=^{11K}_{12K}$	0.29 (+0.05) 0.28 (+0.04)
W cut-off=0.025	0.27 (+0.03)	W cut-off=0.1	0.26 (+0.02)
NetBeans 5.0 (Baseline MIC = 0.20)			
$K=^{300}_{400}$	0.20 (—) 0.21 (+0.01)	$K=^{600}_{700}$	0.19 (-0.01) 0.19 (-0.01)
$II=^{8K}_{9K}$	0.22 (+0.02) 0.23 (+0.03)	$II=^{11K}_{12K}$	0.21 (+0.01) 0.25 (+0.05)
W cut-off=0.025	0.21 (+0.01)	W cut-off=0.1	0.19 (-0.01)
NetBeans 5.5.1 (Baseline MIC = 0.19)			
$K=^{300}_{400}$	0.19 (—) 0.18 (-0.01)	$K=^{600}_{700}$	0.18 (-0.01) 0.23 (+0.04)
$II=^{8K}_{9K}$	0.27 (+0.08) 0.18 (-0.01)	$II=^{11K}_{12K}$	0.20 (+0.01) 0.27 (+0.08)
W cut-off=0.025	0.20 (+0.01)	W cut-off=0.1	0.19 (—)

TABLE X

RESULTS OF THE PARAMETER SENSITIVITY ANALYSIS OF THE PARAMETERS THAT MAY INFLUENCE THE *prediction result*. THE BASELINE PARAMETERS AND THEIR VALUES ARE SHOWN IN THE TABLE. FOR EACH SYSTEM, WE SHOW THE PRECISION AND RECALL WHEN THE PARAMETER CHANGES. THE VALUES IN THE PARENTHESES INDICATE THE INCREASE/DECREASE FROM THE BASELINE PRECISION AND RECALL.

Baseline Values: K=500, II=10,000, W cut-off=0.05, δ=0.01, PCA cut-off=90%					
	Lower			*Higher*	
New Value	Precision	Recall	New Value	Precision	Recall
Mylyn 2.0 (Baseline Precision = 0.82, Base Recall = 0.92)					
K=300	0.81 (-0.01)	0.76 (-0.16)	K=600	0.80 (-0.02)	0.89 (-0.03)
K=400	0.80 (-0.02)	0.91 (-0.01)	K=700	0.90 (+0.08)	0.96 (+0.04)
II=8K	0.85 (+0.03)	0.98 (+0.06)	II=11K	0.78 (-0.04)	0.92 (—)
II=9K	0.84 (+0.02)	0.97 (+0.05)	II=12K	0.86 (+0.04)	0.95 (+0.03)
W cut-off=0.025	0.91 (+0.09)	0.89 (-0.03)	W cut-off=0.1	0.79 (-0.03)	0.93 (+0.01)
δ=0.005	0.82(—)	0.95 (+0.03)	δ=0.02	0.82 (—)	0.92 (—)
PCA cut-off=80%	0.82 (—)	0.92 (—)	PCA cut-off=95%	0.82 (—)	0.92 (—)
Mylyn 3.0 (Baseline Precision = 0.78, Base Recall = 0.79)					
K=300	0.88 (+0.09)	0.70 (-0.09)	K=600	0.63 (-0.15)	0.85 (+0.06)
K=400	0.74 (-0.04)	0.64 (-0.15)	K=700	0.52 (-0.26)	0.89 (+0.07)
II=8K	0.79 (+0.01)	0.78 (-0.01)	II=11K	0.76 (-0.02)	0.74 (-0.05)
II=9K	0.73 (-0.05)	0.78 (-0.01)	II=12K	0.78 (—)	0.76 (-0.03)
W cut-off=0.025	0.77 (-0.01)	0.74 (-0.05)	W cut-off=0.1	0.71 (-0.07)	0.76 (-0.03)
δ=0.005	0.78 (—)	0.78 (-0.01)	δ=0.02	0.78 (—)	0.79 (—)
PCA cut-off=80%	0.78 (—)	0.79 (—)	PCA cut-off=95%	0.78 (—)	0.79 (—)
Eclipse 2.1 (Baseline Precision = 0.79, Base Recall = 0.63)					
K=300	0.86 (+0.07)	0.83 (+0.20)	K=600	0.85 (+0.06)	0.72 (+0.09)
K=400	0.82 (+0.03)	0.63 (—)	K=700	0.74 (-0.05)	0.65 (+0.02)
II=8K	0.79 (—)	0.62 (-0.01)	II=11K	0.80 (+0.01)	0.61 (-0.02)
II=9K	0.77 (-0.02)	0.67 (+0.04)	II=12K	0.80 (+0.01)	0.65 (+0.02)
W cut-off=0.025	0.79 (—)	0.62 (-0.01)	W cut-off=0.1	0.78 (-0.01)	0.65 (+0.02)
δ=0.005	0.79 (—)	0.63 (—)	δ=0.02	0.81 (+0.02)	0.63 (—)
PCA cut-off=80%	0.79 (—)	0.63 (—)	PCA cut-off=95%	0.79 (—)	0.63 (—)
Eclipse 3.0 (Baseline Precision = 0.84, Base Recall = 0.79)					
K=300	0.73 (-0.11)	0.94 (+0.15)	K=600	0.77 (-0.07)	0.86 (+0.07)
K=400	0.78 (-0.06)	0.84 (+0.05)	K=700	0.75 (-0.09)	0.78 (-0.01)
II=8K	0.80 (-0.04)	0.77 (-0.02)	II=11K	0.84 (—)	0.78 (-0.01)
II=9K	0.80 (-0.04)	0.79 (—)	II=12K	0.81 (-0.03)	0.79 (—)
W cut-off=0.025	0.82 (-0.02)	0.78 (-0.01)	W cut-off=0.1	0.84 (—)	0.81 (+0.02)
δ=0.005	0.84 (—)	0.81 (+0.02)	δ=0.02	0.81 (-0.03)	0.79 (—)
PCA cut-off=80%	0.84 (—)	0.79 (—)	PCA cut-off=95%	0.84 (—)	0.79 (—)
NetBeans 5.0 (Baseline Precision = 0.65, Base Recall = 0.60)					
K=300	0.56 (-0.09)	0.63 (+0.03)	K=600	0.67 (+0.02)	0.58 (-0.02)
K=400	0.60 (-0.05)	0.63 (+0.03)	K=700	0.68 (+0.03)	0.60 (—)
II=8K	0.64 (-0.01)	0.62 (+0.02)	II=11K	0.64 (-0.01)	0.59 (-0.01)
II=9K	0.63 (-0.02)	0.59 (-0.01)	II=12K	0.61 (-0.04)	0.61 (+0.01)
W cut-off=0.025	0.65 (—)	0.57 (-0.03)	W cut-off=0.1	0.65 (—)	0.63 (+0.03)
δ=0.005	0.65 (—)	0.60 (—)	δ=0.02	0.63 (-0.02)	0.60 (—)
PCA cut-off=80%	0.65 (—)	0.60 (—)	PCA cut-off=95%	0.65 (—)	0.60 (—)
NetBeans 5.5.1 (Baseline Precision = 0.64, Base Recall = 0.86)					
K=300	0.58 (-0.06)	0.88 (+0.02)	K=600	0.65 (+0.01)	0.79 (-0.07)
K=400	0.67 (+0.03)	0.94 (+0.08)	K=700	0.68 (+0.04)	0.90 (+0.04)
II=8K	0.65 (+0.01)	0.88 (+0.02)	II=11K	0.65 (+0.01)	0.85 (-0.01)
II=9K	0.68 (+0.04)	0.88 (+0.02)	II=12K	0.63 (-0.01)	0.83 (-0.03)
W cut-off=0.025	0.66 (+0.02)	0.86 (—)	W cut-off=0.1	0.62 (-0.02)	0.85 (-0.01)
δ=0.005	0.65 (+0.01)	0.86 (—)	δ=0.02	0.61 (-0.03)	0.85 (-0.01)
PCA cut-off=80%	0.64 (—)	0.86 (—)	PCA cut-off=95%	0.64 (—)	0.86 (—)

TABLE XII
CROSS-VERSION CLASSIFICATION RESULTS WHEN CHANGING THE CLASSIFICATION THRESHOLD *from the third quantile to median*. THE TABLE SHOWS THE PRECISION, RECALL, % OF OVERLAPPING FILES THAT ARE IDENTIFIED BY OUR TOPIC-BASED APPROACH AND LM, AND THE DEFECT DENSITY OF THE IDENTIFIED FILES.

Predicted System	Precision	Recall	% overlapping files	Defect Density (Defect/KLOC) Topic-based	LM
Mylyn 2.0	0.98	0.95	0%	7.33	5.76
Mylyn 3.0	0.85	0.85	0%	3.28	3.10
Eclipse 2.1	0.85	0.74	2.1%	2.63	1.01
Eclipse 3.0	0.73	0.75	3.5%	3.50	1.86
NetBeans 5.0	0.73	0.68	5.5%	0.09	0.23
NetBeans 5.5.1	0.79	0.76	7.6%	0.32	0.45

TABLE XIII
CROSS-VERSION CLASSIFICATION RESULTS WHEN CHANGING THE CLASSIFICATION THRESHOLD *from the third quantile to the 90 percentile*. THE TABLE SHOWS THE PRECISION, RECALL, % OF OVERLAPPING FILES THAT ARE IDENTIFIED BY OUR TOPIC-BASED APPROACH AND LM, AND THE DEFECT DENSITY OF THE IDENTIFIED FILES.

Predicted System	Precision	Recall	% overlapping files	Defect Density (Defect/KLOC) Topic-based	LM
Mylyn 2.0	0.67	0.78	0%	0	5.76
Mylyn 3.0	0.76	0.72	0%	4.33	1.02
Eclipse 2.1	0.79	0.79	1.6%	5.33	1.11
Eclipse 3.0	0.63	0.84	1.5%	7.79	1.88
NetBeans 5.0	0.55	0.67	0.7%	0.60	0.25
NetBeans 5.5.1	0.60	1.00	7.6%	0.71	0.47

topics is reduced, the precision and recall decrease (class imbalance increases). However, we find that after changing the threshold, our topic-based approach can identify files with much higher defect density, compared to the same number of files identified by LM. One exception is Mylyn 2.0, where our topic-based approach does not locate any defect-prone files. After manual investigation, we find that there is only one file that belongs to the LTHD class, and this file contains no defect. This may imply that the threshold is too high for Mylyn 2.0, and other threshold values should be used. In short, after changing the threshold to median and the 90% quantile, the results follow the same trend. However, higher threshold values may result in identifying fewer files (but these files have higher defect density), which may reduce the manual inspection effort required by the practitioners.

2) Different Source Code Preprocessing Steps: The source code preprocessing steps may affect the resulting topics and thus, our results. Thus far, the effects of the preprocessing steps have received limited empirical attention from our community, so it is not yet clear which steps are the best. Some researchers split identifiers and some do not; some researchers stem and some do not. There are many papers that even propose advanced splitting techniques [55], [56]. A study by the second author [57] explicitly considers these issues, and finds that indeed, splitting, stopping, and stemming are all beneficial for topic models in the context of bug localization. Nonetheless, much more research is needed to quantify these effects for other software engineering tasks.

TABLE XIV
PRECISION OF OUR HEURISTIC ON IDENTIFYING SOURCE CODE AND TEST FILES.

Studied System	Precision of finding source files	Precision of finding test files
Mylyn	1.00	0.94
Eclipse	1.00	0.90
NetBeans	0.98	0.76

3) Correctness of the Heuristic for Finding Test Files: Our approach relies on the accuracy of the heuristic that we use to identify test files in a system (Section III-B). Although we define the heuristic for finding test files based on previous research [23], it is not clear how this heuristic performs in our studied systems. Therefore, we randomly select 50 source code files and 50 test files from each studied system. The fourth author manually checks whether these files are classified correctly, and the first author verifies the classification result. The files that we randomly sampled are tagged and made available online [16].

We use the following criteria for determining whether a file is a source code or test file: a file is classified as a test file if it is testing some functionality of the system. Table XIV shows the precision of our heuristic on identifying source code and test files. The heuristic has high precision in Mylyn and Eclipse, but a relatively lower precision in NetBeans. We found that NetBeans provides tools for users to create test cases, which our approach identifies these files as test files. In addition, some test files that we randomly sampled in NetBeans are very short, and only provide class signatures (e.g., interfaces). These files again do not provide any means of code coverage. The relatively lower precision of finding test files in NetBeans may explain why our approach is not performing as well in NetBeans. Nevertheless, the heuristic still has a high precision value for the studied systems.

4) Classifier Choice: The goal of this paper is to provide initial evidence that it is possible to analyze code coverage using topic models, and to identify topics of the source code files that require more testing. We choose a naïve Bayes classifier, which is shown to have a good performance in classifying defective files [37], [38]. However, other classifiers may also be used, and different classifiers may have different classification performance.

B. Internal Validity

1) Assumption of the Topic Metrics: In our topic testedness metric, we made the assumption that if a topic appears more in test files, then this topic is well-tested. However, this may not always be true, since there may be some set-up code in the test files that is not meant to be tested. This threat is not specific to our approach, because traditional coverage-based approaches (e.g., [58]) make similar assumptions. For example, in traditional block coverage approaches, a code block is assumed to be tested if a test case covers it, but the code block may be executed only for setting up the test case. Nevertheless, the success of the test case depends on the fact that the setup code being executed correctly. As a result, in

TABLE XV
PERCENTAGE OF SOURCE CODE FILES THAT ARE TESTED BY THE TEST FILES THAT BELONG TO THE SAME TOPIC (WHEN USING DIFFERENT TOPIC MEMBERSHIP THRESHOLDS).

Studied Systems	% Source Code Covered		
	$\theta = 0.1$	$\theta = 0.3$	$\theta = 0.5$
Mylyn	24	43	35
Eclipse	37	45	60
NetBeans	37	43	55

coverage-based testing, a test case may be explicitly testing one functionality but also implicitly testing other functionality.

The CCDT and CDDT metrics may also have such problem. For example, if 80% of a file belongs to topic A, then it may not be the case that 80% of its defects are about topic A; these defects may be related to other 20% of topics. Obtaining code-level metrics is a difficult task, and most researchers only focus on file-level defect analysis. Fortunately, although such issues may be present in our metrics, the results of our approaches are still promising. Moreover, one of the advantages of our topic metrics is that we are taking the cumulated values of each topic across all the files. By considering the information in all the files, hopefully the problem will be averaged out. We plan to redefine our topic metrics using code-level information and study its improvement over using file-level information in the future.

Finally, the quality of the test cases, and words used in source code and tests may also affect the performance of our approach.

2) Studying Code Coverage Using Topics: We made the assumption that a source code file is tested by a test file if they share the same topic (i.e., have a covering relationship). We conducted an experiment to verify the covering relationship between test code and source code. The experiment is described as follows. We first identify the topics that are shared by both source code and tests (i.e., shared topics). Then, we identify the source code and test files that belong to the same topic in order to examine the covering relationship. Note that since we need to use a threshold (i.e., topic membership value) to find files that belong to the same topic, we have tried out three different threshold values: 0.1, 0.3, 0.5. For example, a topic membership of 0.5 for a file means that 50% of the file is related to the topics.

Table XV shows the percentage of source code files that are tested by the test files that belong to the same topic. In general, we find that there is a covering relationship if two types of files (source code and test files) belong to the same topic. Around 30% to 60% of the source code files are tested by the test files that belong to the same topic. The finding further suggests that it is possible to apply the suggested approach to study code coverage using topic models. The finding further supports our results in the paper. Our proposed approach is indeed capturing the covering relationship and we are able to accurately identify less tested topics and more defect-prone topics.

3) Interpreting Topics: Although Hindle *et al.* [29] show that topics generated by LDA make sense to practitioners, people without domain knowledge still have some difficulties understanding the topics. In cases where developers have hard time interpreting the LTHD topics that our approach points out, developers can use our approach to map the topics to the corresponding files. Another benefit of recovering the mapping is that developers may find it easier to work with files when it comes down to writing test cases. Thus, our approach can still help developers when the topics are difficult to interpret.

C. External Validity

1) Studied Systems: In our case study, we considered in detail three versions of Mylyn, Eclipse, and NetBeans. However, these three systems may not be representative. We try to choose systems with various sizes to minimize the threats, but more case studies are needed to verify the generalizability of our approach.

VII. RELATED WORK

Our work is mostly related to applications of topic models in software engineer, defect prediction using topic models, and software testing.

A. Applications of Topic Models in Software Engineering

Researchers have applied topic models to a wide range of tasks in software engineering. Kuhn *et al.* [18] use Latent Semantic Indexing (LSI) to help developers better understand software systems by clustering source code files based on the similarity of word usage. Linstead *et al.* [12] and Thomas *et al.* [17], [26] use topics to study the evolution in the source code. Maskeri *et al.* [11] apply LDA to source code to uncover its business logic. Other researchers apply topic models to software engineering problems such as recovering traceability links between documents and source code [28], locating concepts in the source code [59], and searching relevant code in a system [60]. Bavota *et al.* [61] use relational topic models (RTM) to study software coupling. In another work, Bavota *et al.* [62] use RTM to help software refactoring. Gethers *et al.* [63] develop an Eclipse plugin for viewing the similarity between source code and requirements. Gethers *et al.* [64] combine several information retrieval approaches for traceability link recovery. Finally, Chen *et al.* [65] survey over one hundred papers that are related to the use of topic models on software engineering tasks.

The most relevant work to this paper is about prioritizing test cases using topic models [66]. Thomas *et al.* show that topic models can be used to help software testing process by considering the semantic difference among test cases. In this paper, we study how testing may help improve software quality at the topic level, and show that our approach can complement current existing approaches.

B. Defects and Topic Models

Recent studies also apply topic models to understand software quality. These approaches differ from traditional approaches [36], [67], [68], since they consider the topics in source code files. Nguyen *et al.* [10] use LDA to propose

a topic-based defect prediction approach. The authors first apply LDA to the studied systems using K=5 topics, and propose a topic metrics based on these five topics. Liu *et al.* [9] propose a metric called Maximal Weighted Entropy (MWE) to model class cohesion using topics, and study the effect of topic cohesion on software defects.

In earlier work [13], [69], we consider the defect history of topics, and show that each topic should be treated differently. Some topics have much higher CDDT than others, and topics can help explain defects in software system. In this paper, we go one step further: we study the effect of topic testedness on CDDT, and show that we can help practitioners allocate testing resources through our case studies.

C. Testing and Software Quality

Some researchers, such as Zaidman *et al.* [23], study the co-evolution between production code and test code using three different views: change history, growth history, and test quality evolution. Instead of using the topics in source code, Zaidman *et al.* create links between source code and test files using class signature. Nagappan *et al.* [5] use traditional test metrics to predict software defects at the file level. One major difference between our study and that of Zaidman *et al.* and Nagappan *et al.* is that we study the relationship between software testing and quality using topics. By using topics, we can provide a more intuitive overview about which features may require more testing to testers. Previous researches have studied code coverage using criteria such as statement coverage, path coverage, function converge etc. [2]–[4]. These criteria study code coverage from the structure and function call graphs to evaluate the testing approaches. However, they do not identify which part of the system is defect-prone, hence requiring more testing. In this paper, we use topics to study topic-level code coverage and identify those part of the system that are defect-prone and require more testing.

VIII. Conclusions and Future Work

In this paper, we studied the effect of topic testedness, i.e., the extent to which a topic is tested, on code quality. We proposed new topic metrics to study this relationship. We performed a detailed case study on three large, real-world systems: Mylyn, Eclipse, and NetBeans. The summary and highlights of our findings include:

- Test files and source code files share a significant number of topics.
- The cumulative defect density of a topic (CDDT) and topic testedness have a non-coexistence relationship, i.e., do not both have high values. Well-tested topics are unlikely to have high CDDT; high CDDT topics have low testedness.
- We are able to predict, with high recall and precision, whether an under-tested topic is at risk of containing defects, which can help practitioners allocate testing sources more effectively.
- Although our approach is done at the topic level, we can still map from topics to files and help practitioners allocate testing resources.
- Our approach outperforms tradition prediction-based resource allocation approaches in terms of allocating testing and code inspection resources.

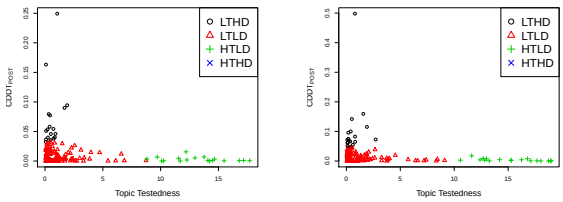
In future work, we plan to compare traditional code coverage approaches with our topic-based approach. We also plan to use traditional code coverage analysis approaches in conjunction with the topic-based approach proposed in this paper, to better identify those parts of the system that require more testing, and to examine more systems.

Acknowledgment

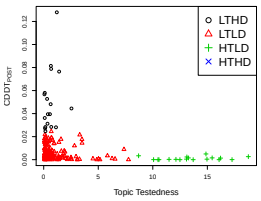
We thank Dr. Yasutaka Kamei for providing us the bug datasets of the studied systems that are used in this paper.

Appendix A
CDDT$_{post}$ v.s. Topic Testedness

This appendix shows scatter plots of cumulative post-release defect density of a topic (CDDT$_{post}$) against topic testedness for all versions of the studied systems (Figure 5, 6, and 7). Each point represents a topic. Black points (class LTHD) are under-tested and defect-prone topics; red points (class LTLD) are under-tested and less defect-prone topics; green points (class HTLD) are well-tested and less defect-prone topics; and blue points (class HTHD) are well-tested and highly defect-prone topics.



(a) Mylyn 1.0 (b) Mylyn 2.0

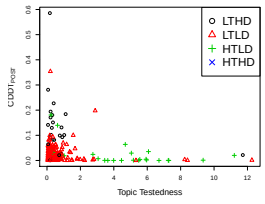


(c) Mylyn 3.0

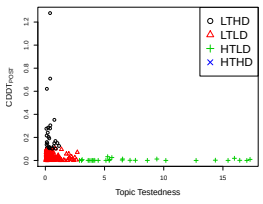
Fig. 5. CDDT$_{post}$ v.s. topic testedness for Mylyn.

References

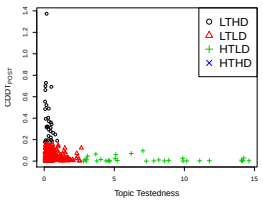
[1] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, Feb. 2007.
[2] S. Ntafos, "A comparison of some structural testing strategies," *IEEE Transactions on Software Engineering*, vol. 14, pp. 868 –874, jun 1988.
[3] J. C. Huang, "An approach to program testing," *ACM Comput. Surv.*, vol. 7, pp. 113–128, Sep. 1975.
[4] G. Myers, T. Badgett, T. Thomas, and C. Sandler, *The Art of Software Testing*, 2nd ed., 2004.



(a) Eclipse 2.0

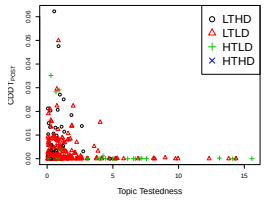


(b) Eclipse 2.1

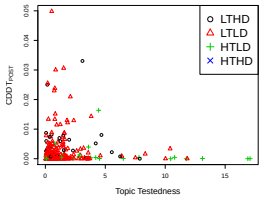


(c) Eclipse 3.0

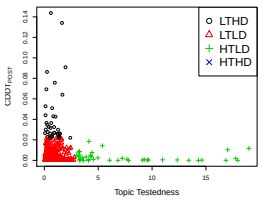
Fig. 6. CDDT$_{post}$ v.s. topic testedness for Eclipse.



(a) NetBeans 4.0



(b) NetBeans 5.0



(c) NetBeans 5.5.1

Fig. 7. CDDT$_{post}$ v.s. topic testedness for NetBeans.

[5] N. Nagappan, L. Williams, M. Vouk, and J. Osborne, "Using in-process testing metrics to estimate post-release field quality," in *Proceedings International Symposium on Software Reliability Engineering*, 2007, pp. 209–214.

[6] E. Weyuker, "Testing component-based software: a cautionary tale," *IEEE Software*, vol. 15, no. 5, pp. 54 –59, sep 1998.

[7] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 2008, pp. 543–562.

[8] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[9] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, and N. Chrisochoides, "Modeling class cohesion as mixtures of latent topics," in *Proceedings of the 25th International Conference on Software Maintenance*, 2009, pp. 233 –242.

[10] T. T. Nguyen, T. N. Nguyen, and T. M. Phuong, "Topic-based defect prediction," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 932–935.

[11] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent Dirichlet allocation," in *Proceedings of the 1st India Software Engineering Conference*, 2008, pp. 113–120.

[12] E. Linstead, C. Lopes, and P. Baldi, "An application of latent Dirichlet allocation to analyzing software evolution," in *Proceedings of Seventh International Conference on Machine Learning and Applications*, 2008, pp. 813–818.

[13] T.-H. Chen, S. W. Thomas, M. Nagappan, and A. Hassan, "Explaining software defects using topic models," in *Proceedings of the 9th Working Conference on Mining Software Repositories*, 2012, pp. 189–198.

[14] J. R. Horgan, S. London, and M. R. Lyu, "Achieving software quality with testing coverage measures," *Computer*, vol. 27, no. 9, pp. 60–69, Sep. 1994.

[15] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," vol. 334, no. 6062, 2011, pp. 1518–1524.

[16] T.-H. Chen, http://petertsehsun.github.io/topicTesting.html, 2015.

[17] S. Thomas, B. Adams, A. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 55–64.

[18] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, pp. 230–243, 2007.

[19] D. Posnett, P. Devanbu, and V. Filkov, "MIC check: A correlation tactic for ESE data," in *Proceedings of the 9th Working Conference on Mining Software Repositories*, 2012, pp. 22–31.

[20] D. Reshef, Y. Reshef, P. Sabeti, and M. Mitzenmacher, "Mine: maximal information-based nonparametric exploration," 2012. [Online]. Available: http://www.exploredata.net/

[21] J. Stewart, *Calculus: Concepts and Contexts*, ser. Stewart's Calculus Series. Cengage Learning, 2009.

[22] I. Titov and R. McDonald, "Modeling online reviews with multi-grain topic models," in *Proceedings of the 17th international conference on World Wide Web*, 2008, pp. 111–120.

[23] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v. Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008, pp. 220–229.

[24] S. W. Thomas, "Mining software repositories with topic models," School of Computing, Queen's University, Tech. Rep. 2012-586, 2012.

[25] T. Hofmann, "Probabilistic Latent Semantic Indexing," in *Proceedings of the 22nd International Conference on Research and Development in Information Retrieval*, 1999, pp. 50–57.

[26] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Modeling the evolution of topics in source code histories," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 173–182.

[27] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *Proceedings of the 26th International Conference on Software Maintenance*, 2010, pp. 1–10.

[28] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 95–104.

[29] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?" in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012, pp. 243–252.

[30] H. Wallach, D. Mimno, and A. McCallum, "Rethinking LDA: Why priors matter," *Proceedings of Neural Information Processing Systems*, pp. 1973–1981, 2009.

[31] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, pp. 972–990, September 2010.

[32] A. K. McCallum, "Mallet: A machine learning for language toolkit," 2002. [Online]. Available: http://mallet.cs.umass.edu

[33] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational Linguistics*, vol. 18, pp. 467–479, Dec. 1992.

[34] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '07, 2007.

[35] F. Provost, "Machine learning from imbalanced data sets 101 (extended abstract)," in *Proceedings of the AAAI 2000 Workshop on Imbalanced Data Sets*, 2000.

[36] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.
[37] B. Turhan and A. Bener, "Analysis of naive bayes assumptions on software fault data: An empirical study," *Data and Knowledge Engineering*, vol. 68, no. 2, pp. 278–290, 2009.
[38] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan 2007.
[39] I. Jolliffe, *Principal Component Analysis*. Springer, 2002.
[40] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference*, 2011, pp. 4–14.
[41] T. Mende and R. Koschke, "Effort-aware defect prediction models," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 107–116.
[42] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the 26th International Conference on Software Maintenance*, 2010, pp. 1–10.
[43] M. DAmbros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceedings of the 7th Conference on Mining Software Repositories*, 2010, pp. 31–41.
[44] J. Rosenberg, "Some misconceptions about lines of code," in *Proceedings of the 4th International Symposium on Software Metrics*, 1997, pp. 137–142.
[45] G. Jay, J. E. Hale, R. K. Smith, D. P. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship." *Journal of Software Engineering and Applications*, vol. 2, pp. 137–143, 2009.
[46] A. Oram and G. Wilson, *Making Software: What Really Works, and Why We Believe It*, ser. O'Reilly Series. O'Reilly Media, Incorporated, 2010.
[47] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of 27th International Conference on Software Engineering*, 2005, pp. 284–292.
[48] S. Biyani and P. Santhanam, "Exploring defect data from development and customer usage on software modules over multiple releases," in *Proceedings of the 9th International Symposium on Software Reliability Engineering*, 1998, pp. 316–320.
[49] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, Oct. 2005.
[50] R. Moser, W. Pedrycz, and G. Succi, "Analysis of the reliability of a subset of change metrics for defect prediction," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2008, pp. 309–311.
[51] "Eclipse bug report 71888," https://bugs.eclipse.org/bugs/show_bug.cgi?id=71888, "Last accessed July 2016".
[52] 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 *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 522–531.
[53] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 837–847.
[54] T. Hofmann, "Unsupervised learning by probabilistic Latent Semantic Analysis," *Machine Learning*, vol. 42, no. 1, pp. 177–196, 2001.
[55] D. Binkley, D. Lawrie, and C. Uehlinger, "Vocabulary normalization improves IR-based concept location," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, 2012, pp. 588–591.
[56] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, 2011, pp. 113–122.
[57] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, "The impact of classifier configuration and combination on bug localization," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1427–1443, 2013.
[58] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.
[59] B. Cleary, C. Exton, J. Buckley, and M. English, "An empirical analysis of information retrieval based concept location techniques in software comprehension," *Empirical Software Engineering*, vol. 14, no. 1, pp. 93–130, 2008.
[60] K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent Dirichlet allocation for automatic categorization of software," in *Proceedings of the 6th International Working Conference on Mining Software Repositories*, 2009, pp. 163–166.
[61] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, pp. 4:1–4:33, 2014.
[62] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, July 2014.
[63] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "CodeTopics: Which topic am I coding now?" in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1034–1036.
[64] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, 2011, pp. 133–142.
[65] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1–77, 2016.
[66] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2012.
[67] T. Graves, A. Karr, J. Marron, and H. Siy, "Predicting fault incidence using software change history," *Transactions on Software Engineering*, vol. 26, no. 7, pp. 653 –661, 2000.
[68] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 78–88.
[69] T.-H. Chen, W. Shang, M. Nagappan, A. E. Hassan, and S. W. Thomas, "Topic-based software defect explanation," *Journal of Systems and Software*, pp. –, 2016.

Tse-Hsun Chen Tse-Hsun Chen is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained his BSc from the University of British Columbia, and MSc and PhD from Queen's University. Besides his academic career, Dr. Chen also worked as a software performance engineer at BlackBerry for over four years. His research interests include performance engineering, database performance, program analysis, log analysis, and mining software repositories. Early tools that are developed by Dr. Chen are integrated into industrial practice for ensuring the quality of large-scale enterprise systems. More information at: http://petertsehsun.github.io/

Stephen W. Thomas Stephen W. Thomas received an MS degree in computer science from the University of Arizona in 2009 and a PhD from Queens University in 2012. His research interests include empirical software engineering, temporal databases, and unstructured data mining.

Hadi Hemmati Dr. Hadi Hemmati is an assistant professor at the department of electrical and computer engineering University of Calgary. Before joining University of Calgary in 2017, Dr. Hemmati was an assistant professor at the department of computer science, University of Manitoba, Canada (2013-2016), where he still has an adjunct position, and a postdoctoral fellow at University of Waterloo (2012-2013), and Queens University (2011-2012). He received his PhD from Simula Research Laboratory, Norway. His main research interest is Automated Software Engineering with a focus on software testing and quality assurance.His research has a strong focus on empirically investigating software engineering practices in large-scale systems, using model-driven engineering and data science. He has/had industry research collaborations with SEB, Latvia; Micropilot, Canada; CA Technologies, USA; Blackberry, Canada; and Cisco Systems, Norway.

Meiyappan Nagappan Meiyappan Nagappan is an Assistant Professor in the David R. Cheriton School of Computer Science at the University of Waterloo. His research is centered around the use of large-scale Software Engineering (SE) data to address the concerns of the various stakeholders (e.g., developers, operators, and managers). He received a PhD in computer science from North Carolina State University. Dr. Nagappan has published in various top SE venues such as TSE, FSE, EMSE, and IEEE Software. He has also received best paper awards at the International Working Conference on Mining Software Repositories (MSR 12, 15). He is currently the Editor of the IEEE Software Blog and the Information Director for the IEEE Transactions on Software Engineering. He continues to collaborate with both industrial and academic researchers from the US, Canada, Japan, Germany, Chile, and India. You can find more at mei-nagappan.com.

Ahmed E. Hassan Ahmed E. Hassan is a Canada Research Chair in Software Analytics and the NSERC/Blackberry Industrial Research Chair at the School of Computing in Queen's University. Dr. Hassan serves on the editorial board of the IEEE Transactions on Software Engineering, the Journal of Empirical Software Engineering, and PeerJ Computer Science. He spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community.

Early tools and techniques developed by Dr. Hassan's team are already integrated into products used by millions of users worldwide. Dr. Hassan industrial experience includes helping architect the Blackberry wireless platform, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. Dr. Hassan is the named inventor of patents at several jurisdictions around the world including the United States, Europe, India, Canada, and Japan. More information at: http://sail.cs.queensu.ca/