Explaining Software Defects Using Topic Models

Tse-Hsun Chen, Stephen W. Thomas, Meiyappan Nagappan, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
{*tsehsun, sthomas, mei, ahmed*}@*cs.queensu.ca*

***Abstract*—Researchers have proposed various metrics based on measurable aspects of the source code entities (e.g., methods, classes, files, or modules) and the social structure of a software project in an effort to explain the relationships between software development and software defects. However, these metrics largely ignore the actual functionality, i.e., the *conceptual concerns*, of a software system, which are the main technical concepts that reflect the business logic or domain of the system. For instance, while lines of code may be a good general measure for defects, a large entity responsible for simple I/O tasks is likely to have fewer defects than a small entity responsible for complicated compiler implementation details. In this paper, we study the effect of conceptual concerns on code quality. We use a statistical topic modeling technique to approximate software concerns as *topics*; we then propose various metrics on these topics to help explain the defect-proneness (i.e., quality) of the entities. Paramount to our proposed metrics is that they take into account the defect history of each topic. Case studies on multiple versions of Mozilla Firefox, Eclipse, and Mylyn show that (i) some topics are much more defect-prone than others, (ii) defect-prone topics tend to remain so over time, and (iii) defect-prone topics provide additional explanatory power for code quality over existing structural and historical metrics.**

***Keywords*-software concerns; code quality; topic modeling**

I. Introduction and Motivation

Code quality is an important issue in software engineering because the cost of fixing software defects can be prohibitively high [1]. As a result, researchers have tried to uncover the possible reasons for software defects using different classes of software metrics, such as product metrics, process metrics, and project metrics [2, Chapter 4] [3]. Indeed, such metrics have shown a certain success in explaining the defect-proneness of certain software entities (e.g., methods, classes, files, or modules) [3]. However, these classes of metrics do not take into account the actual *conceptual concerns* of the software system—the main technical concepts and business logic embedded with the entities [4].

Recent studies propose a new class of metrics based on conceptual concerns [4]–[7]. These studies approximate concerns using *statistical topic models*, such as latent Dirichlet allocation [8]. Statistical topic models discover *topics* (i.e., sets of related words) within the source code entities, which researchers use as surrogates for conceptual concerns. These studies provide initial evidence that topics in software systems are related to the defect-proneness of source code entities, opening a new perspective for explaining why some entities are more defect-prone than others.

In this paper, we build on this line of research by considering the *defect history* of topics, and propose a new set of metrics based on this history. We study the effects of our new topic metrics on code quality. We perform a detailed case study on three large, real-world software systems, with a focus on the following research questions.

RQ1: Are some topics more defect-prone than others?
We find that some topics, such as those related to new features and the core functionality of a system, have a much higher defect density than others.

RQ2: Do defect-prone topics remain defect-prone over time?
We find that defect-prone topics remain so over time, indicating that prior defect-proneness of a topic can be used to explain the future behavior of topics and their associated entities.

RQ3: Can our proposed topic metrics help explain why some entities are more defect-prone than others?
We find that including our proposed topic metrics provides additional explanatory power about the defect-proneness of entities over existing product and process metrics.

The rest of this paper is organized as follows. In Section II, we describe our approach to discover topics in source code entities, and we define the topic metrics that we use to answer our research questions. In Section III, we introduce the subject systems that we study, and outline the design of our case studies. We present our results in Section IV. We discuss the potential threats to the validity of our case studies in Section V, and describe related work in Section VI. Finally, we conclude in Section VII.

II. Proposed Approach

In this section, we outline our approach to use topics to explain defects. First, we briefly introduce topic modeling and describe how it can be applied to source code entities to approximate conceptual concerns. Next, we motivate and describe our new topic metrics.

	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).

Figure 1. Example topic model in which three topics are discovered from three entities. (a) The three discovered topics (z_1, ..., z_3) are defined by their top (i.e., highest probable) words. (b) The three original source code entities (f_1, ..., f_3) are represented by a topic membership vector.

A. Topic Modeling

Our goal is to determine which concerns are in each source code entity. This information is often not easily available, since developers do not often manually categorize each entity [6]. In this paper, we approximate concerns using statistical topics, following the work of previous research [6], [9], [10]. In particular, we extract the *linguistic data* from each source code entity, i.e., the identifier names and comments, which helps to determine the functionality of the entity [11]. We then treat the linguistic data as a corpus of textual documents, which we use as a basis for topic modeling.

In topic modeling, a *topic* is a collection of frequently co-occurring words in the corpus. Given a corpus of n documents $f_1, ..., f_n$, topic modeling techniques 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 concept. 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 operating systems and 70% about computer networks.

More formally, each topic is defined by a probability distribution over all of the unique words in the corpus. Given two Dirichlet priors, α and β, a topic model will generate a topic distribution θ_j for f_j based on α, and generate a word distribution ϕ_i for z_i based on β. Choosing the right parameter values for K, α, and β is more of an art than a science, and depends on the size of the corpus and the desired granularity of the topics [12].

B. Proposed Topic Metrics

To help explain the defect-proneness of source code entities, we propose two categories of topic metrics: *static* and *historical*. Static topic metrics use only a single snapshot of the software system, while historical metrics use the defect history of topics. In the formulation of our topic metrics, we also consider traditional software metrics:

- ***LOC(f_j)*** The lines of code of entity f_j.
- ***PRE(f_j)*** The number of pre-release defects of entity f_j, which are those defects related to f_j up to six months before a given version.
- ***POST(f_j)*** The number of post-release defects of entity f_j, which are those defects found up to six months after a given version.

Using these software metrics and the results of topic modeling, we propose the following topic metrics.

1) Topic Defect Density: The defect density of a source code entity is a well-known software metric, defined as the ratio of the number of defects in the entity to its size. Using this ratio as motivation, we define the ***pre-release defect density*** (***D_{PRE}***) of a topic z_i as

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

where n is the total number of source code entities and θ_{ij} is the topic membership of topic z_i in source code entity f_j. Similarly, we define ***post-release defect density*** (***D_{POST}***) of a topic z_i as

$$D_{\text{POST}}(z_i) = \sum_{j=1}^{n} \theta_{ij} * \left(\frac{\text{POST}(f_j)}{\text{LOC}(f_j)} \right). \tag{2}$$

Since the topic membership value represents the probability that a source code entity belongs to a certain topic, the topic defect density represents the possible number of defects in the topic per line of code across all entities that contain the topic.

2) Static Metrics: We propose static metrics to capture the number of topics an entity contains, and the topic membership of each entity. We define the ***Number of Topics*** (***NT***) of an entity f_j as

$$\text{NT}(f_j) = \sum_{i=1}^{K} I(\theta_{ij} \geq \delta) \tag{3}$$

where I is the indicator function that returns 1 if its argument is true, and 0 otherwise. δ is a cut-off threshold that determines if a topic plays an important role in a given entity. The NT metric measures the level of cohesion in an entity: entities with a large number of topics may be poorly designed or implemented, and thus may have higher chances to have defects [4].

We define the ***Topic Membership*** (***TM***) of an entity f_j as the topic membership values returned by the topic modeling technique:

$$\text{TM}(f_j) = \theta_j. \tag{4}$$

The intuition behind this metric is that we assume different topics have different effects on the defect-proneness of an entity. Some topics (e.g., a compiler-related topic) may

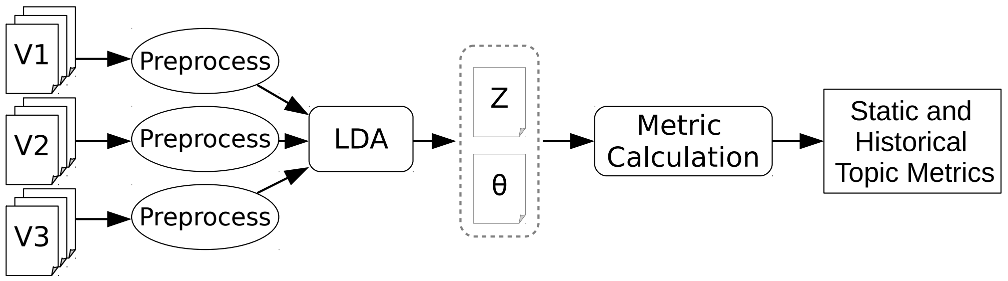

Figure 2. Process of calculating topic metrics. After preprocessing the source code, we run LDA on all versions of the source code entities together. Using the topics and topic memberships that LDA returns, we calculate the topic metrics.

increase the defect-proneness of an entity, but other topics (e.g., an I/O-related topic) may actually decrease the defect-proneness. By using all the topic membership values, the TM metric captures the full behavior of an entity.

3) Historical Metrics: We extend the static metrics by considering the defect history of each topic. In order to calculate the number of defect-prone topics in an entity, we define a *defect-prone topic* as a topic that has more defects than the average of all topics. The set of defect-prone topics, B, is defined by

$$B = \{z_i \in Z \text{ s.t. } D_{\text{PRE}}(z_i) > \mu(D_{\text{PRE}}(Z))\}, \quad (5)$$

where $\mu(D_{\text{PRE}}(Z))$ is the mean of the topic defect densities of all topics.

We define the ***Number of Defect-prone Topics*** (***NDT***) in entity f_j by

$$\text{NDT}(f_j) = \sum_{i=1}^{K} I((z_i \in B) \wedge ((\theta_{ij}) \geq \delta)). \quad (6)$$

We define the ***Defect-prone Topic Membership*** (***DTM***) metric of entity f_j as the topic memberships of defect-prone topics:

$$\text{DTM}(f_j) = \theta_{ij} \text{ where } z_i \in B. \quad (7)$$

DTM is the same as TM, except it only contains the topic memberships of defect-prone topics.

III. Case Study Design

In this section, we introduce the subject systems that we use for our case study and we describe our analysis process, depicted in Figure 2.

A. Subject Systems

We focus on three large, real-world subject systems: Mylyn, Eclipse, and Firefox (Table I). For each system, we look at three different versions (versions 1.0, 2.0, and 3.0 of Mylyn, versions 2.0, 2.1, and 3.0 of Eclipse, and versions 1.0, 1.5, and 2.0 of Firefox). Eclipse is a popular IDE, which has an extensive plugin architecture. Mylyn is a popular plugin for Eclipse that implements a task management system. Firefox is a well-known open source web browser that is used by millions of users.

Table I
Statistics of the subject systems.

	Total lines of code (K)	No. of files	Pre-release defects	Post-release defects	Programming language
Mylyn 1.0	161	1,917	1,142	775	Java
Mylyn 2.0	198	3,710	2,504	1,206	Java
Mylyn 3.0	245	1,521	2,616	624	Java
Firefox 1.0	3,008	5,862	642	457	C
Firefox 1.5	3,329	6,322	721	960	C
Firefox 2.0	3,447	6,468	1,151	456	C
Eclipse 2.0	800	6,729	7,635	1,692	Java
Eclipse 2.1	988	7,888	4,975	1,182	Java
Eclipse 3.0	1,306	10,593	7,422	2,679	Java

B. Data Preprocessing

The source code entities that we use in this paper are at the granularity level of source code files. We first collect the source code entities from each version of each subject system, and then preprocess the entities using the preprocessing steps proposed by Kuhn et al. [11]. Namely, we first extract comments and identifier names from each entity. Next, we split the identifier names according to common naming conventions, such as camel case and underscores. Finally, we stem the words and remove common English-language stop words.

C. Topic Modeling

We use a popular topic modeling technique 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 [13]. In addition, LDA has been shown to be effective for a variety of software engineering purposes, including analyzing source code evolution [14], calculating source code metrics [15], and recovering traceability links between source code and requirements documents [16]. Finally, LDA is fast and can easily scale to millions of documents.

We apply LDA to all versions of the preprocessed entities of a system at the same time, an approach proposed by

Table II
FIVE-NUMBER SUMMARY AND SKEWNESS OF DEFECT DENSITIES OF ALL SUBJECT SYSTEMS. THE DEFECT DENSITIES ARE HIGHLY SKEWED, AND MOST OF THE TOPICS HAVE A DENSITY VALUE CLOSE TO ZERO.

	Min.	1st Qu.	Median	3rd Qu.	Max.	Skewness
Mylyn 1.0	0.00	0.00	0.00	0.01	0.33	7.18
Mylyn 2.0	0.00	0.00	0.01	0.02	0.50	7.41
Mylyn 3.0	0.00	0.00	0.00	0.01	0.18	6.00
Eclipse 2.0	0.00	0.00	0.01	0.03	1.66	13.28
Eclipse 2.1	0.00	0.00	0.01	0.03	1.16	7.92
Eclipse 3.0	0.00	0.01	0.02	0.06	1.25	4.90
Firefox 1.0	0.00	0.00	0.00	0.00	0.07	6.44
Firefox 1.5	0.00	0.00	0.00	0.00	0.09	5.88
Firefox 2.0	0.00	0.00	0.00	0.00	0.07	7.96

Linstead et al. [7]. For this study, we use K=500 topics for all subject systems. Lukins et al. found that 500 topics is a good number for Eclipse and Mozilla [17], and we also feel this is a reasonable choice for Mylyn. Section V discusses this choice further.

We use MALLET [18] as our LDA implementation, which uses Gibbs sampling to approximate the joint distribution of topics and words. We run MALLET with 10,000 sampling iterations, and use the parameter optimization in the tool to optimize α and β. In addition, we build the topics using both unigrams (single words) and bigrams (pairs of adjacent words), since bigrams help to improve the performance for word assignments in topic modeling [19].

We set the membership threshold δ in Equations 3 and 6 to 1%. This value prevents topics with small, insignificant memberships in an entity from being counted in that entity's metrics.

IV. CASE STUDY RESULTS

In this section, we present the results of our case study. We present each research question with three sections: the approach we used to address the question; our experimental results; and a discussion of the results.

RQ1: Are some topics more defect-prone than other topics?

Approach: We use Equation 2 to calculate the topic defect density (D_{POST}) for each topic in the software system. We visualize the distribution of defect densities using box plots, and we provide a table of the five number summary and skewness of the densities. We then perform Kolmogorov-Smirnov non-uniformity tests to statistically determine if there is a significant difference between the defect densities of the various topics.

Results: The box plots of the density values of each software system are shown in Figure 3. Box plots show outliers and the five-number summary of the data (minimum, first quartile, median, third quartile, maximum). The

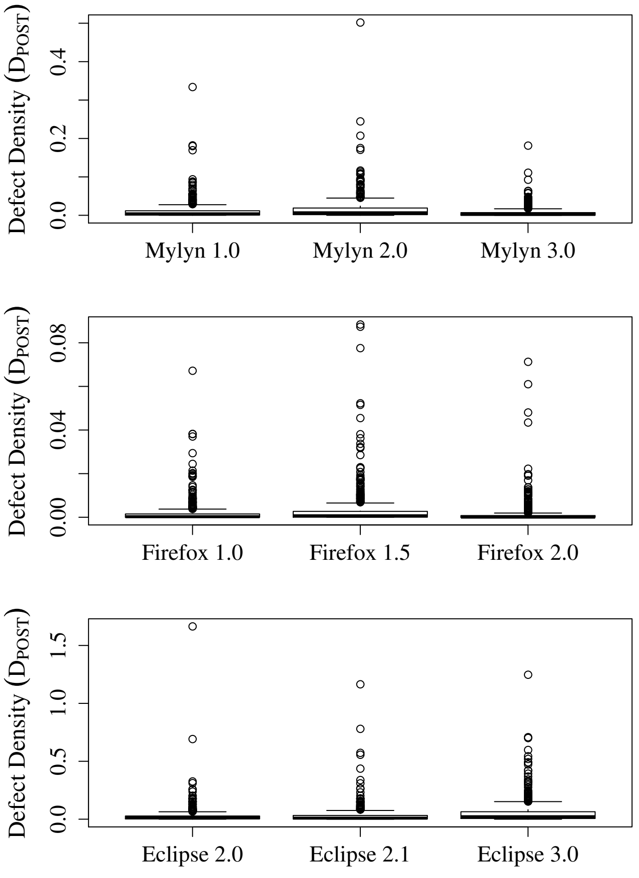

Figure 3. Box plots of the topic defect density of three versions of Mylyn, Firefox, and Eclipse. The y-axis represent the topic defect density.

actual values of the five-number summary and skewness of the defect densities is shown in Table IV. The outliers in Figure 3 are the defect-prone topics, which indicate that some topics have much higher defect densities than others. Table IV further indicates that most topics have a low (almost zero) defect density value, and the values are significantly postively skewed.

The number of topics and defect-prone topics for each system is consistent across versions (Table III). We find that Mylyn has more defect-prone topics than the other two systems, while Firefox has the least number of defect-prone topics. In addition, Eclipse has the highest mean defect density among three systems.

Finally, we apply the Kolmogorov-Smirnov test on the topic defect density values of each version of each subject system to verify the non-uniformity illustrated by our visualizations. If the p-value computed using Kolmogorov-Smirnov test is high, then the data is more likely to be uniformly distributed. However, we find that the p-values for all systems are significantly small (< 0.001), indicating

Table III
FOR EACH SYSTEM, WE SHOW THE MEAN DEFECT DENSITY VALUE ACROSS ALL TOPICS ($\mu(D_{\text{POST}})$), THE NUMBER AND PERCENTAGE OF DEFECT-PRONE TOPICS (NDT), THE MEDIAN NUMBER OF TOPICS IN EACH ENTITY (MED. NT), AND THE MEDIAN NUMBER OF DEFECT-PRONE TOPICS IN EACH ENTITY (MED NDT).

	K	$\mu(D_{\text{POST}})$	NDT	Med. (NT)	Med. (NDT)
Mylyn 1.0	500	0.01	139 (27.8%)	9	7
Mylyn 2.0	500	0.02	137 (27.4%)	9	7
Mylyn 3.0	500	0.01	128 (25.6%)	9	7
Eclipse 2.0	500	0.03	122 (24.4%)	9	5
Eclipse 2.1	500	0.03	124 (24.8%)	9	5
Eclipse 3.0	500	0.06	136 (27.2%)	10	6
Firefox 1.0	500	0.00	106 (21.2%)	5	3
Firefox 1.5	500	0.00	111 (22.2%)	5	3
Firefox 2.0	500	0.00	82 (16.4%)	5	2

that the distribution of defect density values is indeed not uniform [20].

Discussion: To better understand why some topics are more defect-prone than others, we investigated the relevant words of the top three most and least defect-prone topics (Table IV).

Mylyn (previously known as Mylar) is an Eclipse plugin for task management. We find that the topics with the highest defect densities are (i) those dealing with the Eclipse integration (topic 421), likely because the Eclipse plugin API changes so often; (ii) those that are related to the core functionality of the system, i.e., tasks and the task UI (topics 164 and 168); and (iii) those dealing with the test suite of Mylyn (topic 400), likely because of adding test cases for new defect fixes.

On the other hand, the least defect-prone topics deal with images and color (topics 405 and 178) and data compression (topic 175). We postulate that the logic behind these functions may be simpler and better defined than that of the core functionality topics.

Regarding Eclipse, two of the most defect-prone topics (topics 496 and 492) in Eclipse 2.0 are about CVS plug-ins. The build notes for this release indicate that the plug-ins supporting CVS-related functionalities were first introduced in this version, making it an active area of development. (In fact, according to Eclipse's defect repository, 17 defects relating to CVS remained unfixed after the 2.0 release.) A similar story holds for Eclipse 2.1, when integration for the Apache Ant build system was actively developed, leading to many defects in topic 131.

Another set of defect-prone topics in Eclipse deals with low-level details such as memory operations and message passing (topics 462, 169, and 233). We hypothesize that the logic needed to implement these topics are more complex, leading to more defects.

The least defect-prone topics in Eclipse include those about bit-wise operations (topic 116), arrays (topic 182), and parameter parsing (topic 192). One reason that these might contain fewer defects is that errors in these topics may be observed during run time (e.g., "array out-of-bounds") and are thus more easily detected by developers during the testing phase of the project.

One of the most defect-prone topics in Firefox 1.0 and 1.5 deals with event handing (topic 101), which is responsible for dispatching events according to network protocol responses. The topic is likely more defect-prone because the network stack has been modified several times to enable the dynamic re-rendering of complex webpages as they are being loaded.

Another defect-prone topic in Firefox 2.0 deals with accessing saved states (topic 80). The release notes for this version indicate that new features were introduced that allow the browser to restore previous sessions, and that the tabbed browsing functionality is updated.

Scanner Access Now Easy (SANE), an API that enables a scanner/digital camera application to be created with JavaScript, is one of the least defect-prone topics in all versions of Firefox (topic 280). Another topic that is not defect-prone deals with Base64 encoding (topic 359—the characters are segments of encoded characters), a known character standard.

RQ2: Do defect-prone topics remain defect-prone over time?

Approach: In RQ1, we found that some topics are more defect-prone than other topics. In order to verify that these topics are consistently defect-prone over time, we compute the Spearman correlation of the topic defect density values among different versions. By ranking the density values and computing correlation on the ranks, Spearman rank correlation is able to account for skewed distributions.

Results and Discussion: Table V shows the correlation among different versions of a system. The correlation values are consistently medium to high between different versions, which indicates that a defect-prone topic is still likely to be defect-prone in the later versions. We also see evidence of this in Table IV, as several of the top defect-prone topics are listed for each of the versions of a software system.

Therefore, it would be better to allocate more testing resources to previously-identified defect-prone topics, as they are likely to remain defect-prone in later releases.

RQ 3: Can our proposed topic metrics help explain why some entities are more defect-prone than others?

In this research question, we examine how much more deviance in post-release defects our topic metrics can explain, with respect to traditional baseline metrics. This type of analysis allows us verify our empirical theory that topic metrics provide additional explanatory power.

Table IV
TOP WORDS AND DEFECT DENSITIES OF THE MOST/LEAST DEFECT-PRONE TOPICS IN OUR SUTS.

Most Defect Prone			Least Defect Prone		
	Top words	Density		Top words	Density
Mylyn 1.0					
421	mylar, eclips, eclips_mylar, mylar_intern, mylar_task	0.334	405	src, dest, base, imag, fragment, imag_pattern	<0.001
164	task, list, task_list, task_ui, ui, plugin	0.182	178	lower_color, part, put_light green_lower, jface, medium	<0.001
400	test, suit, test_suit, add_test, add, suit_add	0.180	175	monitor, gzip, configur, key, bugzilla_attribut, iter	<0.001
Mylyn 2.0					
143	task, eclips, eclips_mylyn, mylyn, ui, task_ui	0.502	405	src, dest, base, imag, fragment, imag_pattern	<0.001
457	eclips, mylyn, eclips_mylyn, intern, mylyn_intern, core	0.244	178	lower_color, part, put_light, green_lower, jface, medium	<0.001
164	task, list, task_list task_ui, ui, plugin	0.207	175	monitor, gzip, configur, key, bugzilla_attribut, iter	<0.001
Mylyn 3.0					
143	task, eclips, eclips_mylyn, mylyn, ui, task_ui	0.181	178	lower_color, part, put_light, green_lower, jface, medium	<0.001
457	eclips, mylyn, eclips_mylyn, intern, mylyn_intern, core	0.111	310	aa, comparison_check, comparison, check, check_aa, aa_comparison	<0.001
168	repositori, task_repositori, task_core, repositori	0.092	6	select, caller, calle, editor, part, foo	<0.001

Most Defect Prone			Least Defect Prone		
	Top words	Density		Top words	Density
Eclipse 2.0					
174	express, method, declar, ast, node, astnod	1.663	116	0xff, 0xff_0xff, src, dst, 0xa, 0xf	<0.001
496	option, local, seccion, folder, ccv_core, local option	0.691	146	printer, data, printer_data, code, error, dispos	<0.001
492	team, eclips_team, eclips, ccv, intern_ccv, team_intern	0.325	316	run, line, offset, style, length, item	<0.001
Eclipse 2.1					
143	form, toolkit, dfm, nfm, ui, eclips_ui	1.164	192	arg, vtbl, arg_arg, guid, iidfrom, system	<0.001
131	ant, eclips, task, eclips_ant, ui, intern	0.779	325	token, scribe, align, print, scribe_print, space	<0.001
233	bundl, recourc, resourc_bundl, kei, bundl_resourc, messag	0.572	116	0xff, 0xff_0xff, src, dst, 0xa, 0xf	<0.001
Eclipse 3.0					
462	memori, block,render, memori_block, view, address	1.247	330	packet, print, id, command, stream, spy	<0.001
169	transfer,data,code, transfer_data, java, object	0.708	182	array, constant, array_dim, dim, pixbuf, paramet	<0.001
131	ant, eclips, task, eclips_ant, ui, intern	0.700	270	pt, ph, pt_arg, arg, pg, wm	<0.001

Most Defect Prone			Least Defect Prone		
	Top words	Density		Top words	Density
Firefox 1.0					
462	list, val, isvgvalu, modifi, observ, imethodimp	0.067	359	ghhd, sbz_yxkgd, yxkgd, sbz, yxkgd_ghhd, vghle_sbz	<0.001
381	frame, svgframe, comptr, queri, kid, add	0.038	280	sane, plugin, zoom, sane_plugin, instanc, error	<0.001
101	rv, rv_rv, comptr, nsresult, fail, fail_rv	0.038	361	child, border, spec, num, color, col	<0.001
Firefox 1.5					
305	elem, rv, length, map, rv_rv, comptr	0.088	359	ghhd, sbz_yxkgd, yxkgd, sbz, yxkgd_ghhd, vghle_sbz	<0.001
413	xform, elem, model, wrapper, instanc, xform_xpath	0.087	280	sane, plugin, zoom, sane_plugin, instanc, error	<0.001
101	rv, rv_rv, comptr, nsresult, fail, fail_rv	0.078	335	ck, rv, pr, log, modlog, log_modlog	<0.001
Firefox 2.0					
168	param, info, pruint, xpttype, val, count	0.071	359	ghhd, sbz_yxkgd, yxkgd, sbz, yxkgd_ghhd, vghle_sbz	<0.001
305	elem, rv, length, map, rv_rv, comptr	0.061	100	frame, pfd, span, psd, width, line	<0.001
80	access, state, retval, shell, node, comptr	0.048	280	sane, plugin, zoom, sane_plugin, instanc, error	<0.001

Table V
SPEARMAN CORRELATION COEFFICIENTS OF EACH TOPIC'S DEFECT DENSITY ACROSS SOFTWARE VERSIONS.

	Mylyn 1.0	Mylyn 2.0	Mylyn 3.0
Mylyn 1.0	1.000	–	–
Mylyn 2.0	0.673	1.000	–
Mylyn 3.0	0.483	0.493	1.000

	Eclipse 2.0	Eclipse 2.1	Eclipse 3.0
Eclipse 2.0	1.000	–	–
Eclipse 2.1	0.529	1.000	–
Eclipse 3.0	0.438	0.530	1.000

	Firefox 1.0	Firefox 1.5	Firefox 2.0
Firefox 1.0	1.000	–	–
Firefox 1.5	0.536	1.000	–
Firefox 2.0	0.473	0.564	1.000

Approach: As previously mentioned, software metrics can be classified as *static* or *historical*. Static metrics, such as lines of code (LOC), are obtained from a single snapshot of the system [21]. On the other hand, historical metrics require past information about the system, and include pre-release defects (PRE) and code churn (i.e, changes to the code) [22]. As such, in this research question, we build two sets of models: those based on static metrics, and those based on historical metrics. For a baseline static metric, we choose LOC, because LOC is a good general software metric and has been used for benchmarking [23], [24]. For baseline historical metrics, we choose PRE and code churn because they are a good measurement for defects [25], [26], and have also been used as a baseline model for comparing metrics [22].

Our goal here is not to predict post-release defects. Instead, we want to see how much improvement on explaining deviance (i.e., model fitness) in defects our topic metrics can bring to the baseline metrics.

We use logistic regression with post-release defects as our dependent variable, and report the percent deviance explained (D^2) for each combination of independent variables (i.e., metric combinations). (To eliminate any skew in the metric values, we apply a log transformation on the metrics.) Here, the D^2 measure is similar to the adjusted R^2 measure in linear regression, except that D^2 quantifies how much deviance a logistic regression model can explain. A higher D^2 value generally indicates a better model fit, but when the number of independent variables is large, D^2 may not be a good measure. As the number of independent variables increases, D^2 will always increase regardless of the quality of the model. Thus, we also use another measure called the Akaike information criterion (AIC). AIC can be used to compare the fitness of different models, and it penalizes more complex models [27] [28]. Models with lower AIC scores are better.

Recall that by the definition of our TM and DTM metrics (Equations 4 and 7), each metric will produce many values for each entity (K values in the case of TM, and $|B|$ values in the case of DTM). To avoid the problems of overfitting and multicollinearity, we use Principal Component Analysis (PCA) to reduce the dimensionality of the metrics [29]. PCA transforms the data into a smaller set of uncorrelated variables while still capturing the patterns of the original data [29]. We choose 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.

We perform stepwise regression on the PCs of TM and DTM metrics to make our model more robust [30] [31]. Stepwise regression is a variable selection technique, which adds or removes variables to the model according to some criteria, which, in this paper, we choose to use the AIC score.

Results: We present the results in Tables VI and VII. Table VI shows the results for static metrics. We find that adding NT gives a significant improvement in the deviance explained. All models, except Firefox 2.0, have statistically significant (p-value $\leq$ 0.05) improvement when NT is added to the model. In all the versions of Mylyn and Firefox, NT gives at least 18% increase in D^2. However, we find that the performance of NT is not as high in Eclipse.

Both TM and DTM, on the other hand, give significant improvements in all three subject systems. We find that the TM metric improves the deviance explained by 61–162%, compared to the baseline model.

Table VII shows the results for historical metrics. We find that NDT gives a promising improvement in D^2 over the base model. This implies that topics with high pre-release defects are more likely to have post-release defects, and having more defect-prone topics will have negative effects on the code quality. The improvement of NDT is not as large for Eclipse 2.1 and 3.0. However, the improvements achieved by the DTM metric are consistent across all versions of all systems. We find that, overall, DTM improves the explanatory power over the baseline model by 22–76%.

Discussion: One possible explanation as to why the improvement of NDT in Eclipse is not as high as in the other systems is because topics in Eclipse have higher defect density (Table III). Since more topics are defect-prone, the overall explanatory power of NDT decreases. On the other hand, DTM contains a more general information about all the defect-prone topics, which better explains defects.

To see the effects of NT and NDT in our logistic regression models, Table VIII shows the average coefficients of these metrics across the three versions of each subject system. Both NT and NDT have positive coefficients in all the subject systems, which implies that as the number of topics or defect-prone topics increases, an entity will have higher chances to be defect-prone. However, the coefficients of NDT in Firefox and Eclipse, and NT in Eclipse are smaller than one, which means the effects are minor.

Table VIII
AVERAGE VALUE OF THE REGRESSION COEFFICIENTS OF NT AND NDT METRICS.

	Mylyn	Firefox	Eclipse
NT	1.73	1.26	0.06
NDT	1.71	0.85	0.24

Our findings show that the number of topics in an entity has a strong relationship with defects, and entities having more defect-prone topics will more likely to be defect-prone.

V. Threats to Validity

A. Parameter and Threshold Choices

The choice of the optimal number of topics in LDA is a difficult task [12], [32]. Although we use the guidance of a previous study to inform our choices [17], we still cannot be sure that our results are optimal. To alleviate this concern, we investigated two additional values of K, namely K=50 and K=250. We found our overall results to be comparable to K=500, suggesting that our approach is not particularly sensitive to the exact number of topics.

We set our δ threshold to 1%, so topics with a membership value less than 1% will be filtered out. However, further analysis is required to understand the effect of δ on the results of our study.

Additionally, we choose PCs until either 90% of variances are explained or the increase in variance explained by the next PC is less than the mean of the variance explained by all PCs. These thresholds may also affect the result, and different numbers may give slightly different results.

B. Choosing Baseline Metrics

Our goal is to examine the improvement in the explanatory power of topics over traditional static and historical metrics. Since topics are derived from the source code entities and can also be combined with pre-release defects to discover defect-prone topics, we examine the improvement on static and historical metrics separately. It is possible to combine different topics metrics, such as NT and NDT, in our defect explanation models. However, we leave the full investigation of all possible metric combinations to future work.

VI. Related Work

Recently, many researchers used topic modeling techniques to understand software systems from a different point of view than from the traditional structural and historical views. For example, Kuhn et al. used Latent Semantic Indexing (LSI) to cluster the entities in a software system

Table VI
D^2 IMPROVEMENT AND AIC SCORES FOR STATIC SOFTWARE METRICS. NUMBERS IN THE PARENTHESES ARE THE D^2 IMPROVEMENT OR AIC SCORE DECREASEMENT IN PERCENTAGE OF THE BASE MODEL. THE BEST MODEL OF EACH VERSION OF THE SOFTWARE IS MARKED IN BOLD. NT IS STATISTICALLY SIGNIFICANT IN ALL SYSTEMS EXCEPT FIREFOX 2.0.

	Mylyn 1.0		**Mylyn 2.0**		**Mylyn 3.0**	
Model	D^2	AIC	D^2	AIC	D^2	AIC
Base(LOC)	0.09	1047.36	0.14	1078.35	0.13	1159.74
Base+NT	0.14 (+56%)	990.85 (-5%)	0.19 (+36%)	1020.46 (-5%)	0.20 (+54%)	1071.71 (-8%)
Base+TM	***0.21 (+133%)***	***957.83 (-9%)***	***0.27 (+93%)***	***956.73 (-11%)***	***0.34 (+162%)***	***949.17 (-18%)***

	Firefox 1.0		**Firefox 1.5**		**Firefox 2.0**	
Model	D^2	AIC	D^2	AIC	D^2	AIC
Base(LOC)	0.12	2374.85	0.15	3474.84	0.14	2224.76
Base+NT	0.16 (+33%)	2256.41 (-5%)	0.21 (+40%)	3241.09 (-7%)	0.17 (+18%)	2152.19 (-3%)
Base+TM	***0.20 (+67%)***	***2168.37 (-9%)***	***0.28 (+87%)***	***2969.09 (-15%)***	***0.24 (+71%)***	***1973.27 (-11%)***

	Eclipse 2.0		**Eclipse 2.1**		**Eclipse 3.0**	
Model	D^2	AIC	D^2	AIC	D^2	AIC
Base(LOC)	0.18	4584.26	0.11	4804.87	0.14	7591.93
Base+NT	0.18 (+0%)	4575.72 (-0%)	0.11 (+0%)	4793.48 (-0%)	0.14 (+0%)	7589.50 (-0%)
Base+TM	***0.29 (+61%)***	***4003.58 (-13%)***	***0.19 (+73%)***	***4401.56 (-8%)***	***0.24 (+71%)***	***6800.06 (-10%)***

Table VII
D^2 IMPROVEMENT AND AIC SCORES FOR HISTORICAL SOFTWARE METRICS. NUMBERS IN THE PARENTHESES ARE THE D^2 IMPROVEMENT OR AIC SCORE DECREASEMENT IN PERCENTAGE OF THE BASE MODEL. THE BEST MODEL OF EACH VERSION OF THE SOFTWARE IS MARKED IN BOLD. NDT IS STATISTICALLY SIGNIFICANT IN ALL SYSTEMS EXCEPT MYLYN 3.0, ECLIPSE 2.1, AND ECLIPSE 3.0.

	Mylyn 1.0		**Mylyn 2.0**		**Mylyn 3.0**	
Model	D^2	AIC	D^2	AIC	D^2	AIC
Base(PRE+Churn)	0.21	917.38	0.22	987.04	0.28	957.31
Base+NDT	0.24 (+14%)	885.72 (-3%)	0.23 (+4%)	971.01 (-2%)	0.29 (+4%)	955.98 (-0%)
Base+DTM	***0.30 (+43%)***	***824.24 (-10%)***	***0.34 (+55%)***	***882.19 (-11%)***	***0.36 (+29%)***	***909.83 (-5%)***

	Firefox 1.0		**Firefox 1.5**		**Firefox 2.0**	
Model	D^2	AIC	D^2	AIC	D^2	AIC
Base(PRE+Churn)	0.14	2299.70	0.20	3255.54	0.23	2008.67
Base+NDT	0.18 (+29%)	2204.47 (-4%)	0.25 (+25%)	3081.35 (-5%)	0.25 (+9%)	1951.41 (-3%)
Base+DTM	***0.20 (+43%)***	***2152.08 (-6%)***	***0.27 (+35%)***	***3005.55 (-8%)***	***0.28 (+22%)***	***1892.53 (-6%)***

	Eclipse 2.0		**Eclipse 2.1**		**Eclipse 3.0**	
Model	D^2	AIC	D^2	AIC	D^2	AIC
Base(PRE+Churn)	0.17	4605.37	0.15	4586.50	0.17	7310.04
Base+NDT	0.20 (+18%)	4477.51 (-3%)	0.15 (+0%)	4586.19 (-0%)	0.17 (+0%)	7309.30 (-0%)
Base+DTM	***0.30 (+76%)***	***3930.40 (-15%)***	***0.19 (+27%)***	***4366.09 (-5%)***	***0.24 (+41%)***	***6729.03 (-8%)***

according to the similarity of word usage [11]. Maskeri et al. were the first to apply LDA to source code to uncover its conceptual concerns [6]. Linstead et al. and Thomas et al. used topics to study the evolution of concerns in the source code [7], [10], [33].

Other uses of topic models in software engineering tasks include concept location [17], [34]–[37], traceability link recovering [16], and building source code search engines [38].

A few recent studies have tried to establish a link between topics and defects. For example, Liu et al. propose a new metric, called Maximal Weighted Entropy (MWE) [4], to measure the level of cohesion in a software system. MWE, for each topic, captures the topic occupancy and distribution of each entity, i.e., how many different topics an entity contains. While this metric focuses on the cohesiveness of topics in an entity, our proposed metrics focus on the defect-prone topics in an entity.

Nguyen et al. use LDA to predict defects [5]. The authors first apply LDA to the subject systems using K=5 topics, and for each source code entity they multiply the topic memberships by the entity's LOC. As a result, the authors obtain five topic variables for each entity, and use these variables to build a prediction model. In this way, the authors provide initial evidence, that it is possible to explain defects using topic metrics. In this paper, we are interested in explaining defects while also controlling for the standard defect explainers, i.e., LOC, churn, and pre-release defects. In addition, we consider a larger number of topics in order to capture more accurate and detailed conceptual concerns. We use PCA to extract the most effective topics and avoid the possible problem of multicollinearity and minimize the effects of overfitting.

VII. Conclusions and Future Work

In this paper, we have endeavored to understand the relationship between the *conceptual concerns* in source code entities, i.e., their technical content, with their defect-proneness. To do so, we approximated the concerns in each entity with *statistical topics*, and proposed new metrics on these topics. In particular, we considered the defect history of each topic, which we hypothesized would help our metrics to better explain the defect-proneness of the entities.

To evaluate our new metrics, we performed a detailed case study on three large, real-world systems: Mylyn, Mozilla Firefox, and Eclipse. The highlights of our analysis include:

- Some topics are much more defect-prone than others.
- A topic's defect-proneness holds over time.
- The more topics an entity has, the higher the chances it has defects.
- The more *defect-prone* topics an entity has, even higher are the chances that it has defects.
- Our proposed topic metrics provide better explanatory power for defect-proneness over existing static (i.e., LOC) and historical (i.e., churn) metrics, suggesting they provide additional information about the quality of the code. Further study should consider using such metrics alongside traditional metrics for building defect prediction models.

In future work, we plan to combine our proposed topic metrics with existing topic metrics [4], [5], [10] to understand their accuracy in predicting future defects. Additionally, we plan to consider using other information sources, such as defect reports or mailing lists, to help explain defects.

References

[1] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, pp. 67–73, August 1998.

[2] S. H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic review of fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, 2011.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] H. Wallach, D. Mimno, and A. McCallum, "Rethinking LDA: Why priors matter," *Proceedings of NIPS-09, Vancouver, BC*, 2009.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] J. Stapleton, *Models for probability and statistical inference: theory and applications*, 2008.

[21] S. G. Crawford, A. A. McIntosh, and D. Pregibon, "An analysis of static metrics and faults in c software," *Journal of Systems and Software*, vol. 5, pp. 37–48, 1985.

[22] 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.

[23] M. DAmbros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proceeding of the 7th Conference on Mining Software Repositories*, 2010, pp. 31–41.

[24] J. Rosenberg, "Some misconceptions about lines of code," in *Proceedings of the 4th International Symposium on Software Metrics*, 1997, pp. 137–142.

[25] 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.

[26] S. Biyani and P. Santhanam, "Exploring defect data from development and customer usage on software modules over multiple releases," in *Proceedings of the The 9th International Symposium on Software Reliability Engineering*, 1998, pp. 316–320.

[27] A. Raftery, "Bayesian model selection in social research (with discussion)," *Sociological Methodology*, vol. 25, pp. 111–163, 1995.

[28] A. D. R. Burnham Kenneth P., "Multimodel inference: Understanding AIC and BIC in model selection," *Sociological Methods Research*, vol. 33, pp. 467–479, Nov. 2004.

[29] I. Jolliffe, *Principal component analysis*. Springer-Verlag, 2002.

[30] C. Haan, *Statistical methods in hydrology*. Iowa State University Press, 1977.

[31] E. Cureton and R. D'Agostino, *Factor Analysis: An Applied Approach*. Lawrence Erlbaum Associates, 1993.

[32] S. Grant and J. Cordy, "Estimating the optimal number of latent concepts in source code analysis," in *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010, pp. 65–74.

[33] 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.

[34] 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.

[35] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, pp. 420–432, 2007.

[36] M. Revelle, M. Gethers, and D. Poshyvanyk, "Using structural and textual information to capture feature coupling in object-oriented software," *Empirical Software Engineering*, vol. 16, no. 6, 2011.

[37] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proceeding of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 43–52.

[38] 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.