Modeling Changeset Topics for Feature Location

Christopher S. Corley, Kelly L. Kashuda
The University of Alabama
Tuscaloosa, AL, USA
{cscorley, klkashuda}@ua.edu

Nicholas A. Kraft
ABB Corporate Research
Raleigh, NC, USA
nicholas.a.kraft@us.abb.com

***Abstract*—Feature location is a program comprehension activity in which a developer inspects source code to locate the classes or methods that implement a feature of interest. Many feature location techniques (FLTs) are based on text retrieval models, and in such FLTs it is typical for the models to be trained on source code snapshots. However, source code evolution leads to model obsolescence and thus to the need to retrain the model from the latest snapshot. In this paper, we introduce a topic-modeling-based FLT in which the model is built incrementally from source code history. By training an online learning algorithm using changesets, the FLT maintains an up-to-date model without incurring the non-trivial computational cost associated with retraining traditional FLTs. Overall, we studied over 600 defects and features from 4 open-source Java projects. We also present a historical simulation that demonstrates how the FLT performs as a project evolves. Our results indicate that the accuracy of a changeset-based FLT is similar to that of a snapshot-based FLT, but without the retraining costs.**

***Index Terms*—program comprehension; feature location; topic modeling; mining software repositories; changesets**

I. Introduction

Feature location is a frequent and fundamental activity for a developer tasked with changing a software system. Whether a change task involves adding, modifying, or removing a feature, a developer cannot complete the task without first locating the source code that implements the feature. The state-of-the-practice in feature location is to use an IDE tool based on keyword or regex search, but Ko et al. [1] observed such tools leading developers to failed searches nearly 90% of the time.

The state-of-the-art in feature location [2] is to use a feature location technique (FLT) based, at least in part, on text retrieval (TR). The standard methodology [3] is to extract a document for each class or method in a source code snapshot, to train a TR model on those documents, and to create an index of the documents from the trained model. Topics models (TMs) [4] such as latent Dirichlet allocation (LDA) [5] are the state-of-the-art in TR and outperform vector-space models (VSMs) in the contexts of natural language [5], [6] and source code [7], [8]. Yet, modern TMs such as online LDA [9] natively support only the online addition of a new document, whereas VSMs also natively support online modification or removal of an existing document. So, TM-based FLTs provide the best accuracy, but unlike VSM-based FLTs, they require computationally-expensive retraining subsequent to source code changes.

Rao [10] proposed FLTs based on customizations of LDA and latent semantic indexing (LSI) that support online modification and removal. These FLTs require less-frequent retraining than other TM-based FLTs, but the remaining cost of periodic retraining inhibits their application to large software, and the reliance on customization hinders their extension to new TMs.

We envision an FLT that is: (1) accurate like a TM-based FLT, (2) inexpensive to update like a VSM-based FLT, and (3) extensible to accommodate any off-the-shelf TR model that supports online addition of a new document. Unfortunately, our vision is incompatible with the standard methodology for FLTs. Existing VSM-based FLTs fail to satisfy the first criteria, and existing TM-based FLTs fail to satisfy the second or third criteria. Indeed, given the current state-of-the-art in TR, it is impossible for a FLT to satisfy all three criteria while following the standard methodology.

In this paper we propose a new methodology for FLTs. Our methodology is to extract a document for each changeset in the source code history and to train a TR model on the changeset documents, and then to extract a document for each class or method in a source code snapshot and to create an index of the class/method documents from the trained (changeset) model. This new methodology stems from four key observations:

- Like a class/method definition, a changeset has program text.
- Unlike a class/method definition, a changeset is immutable.
- A changeset corresponds to a commit.
- An atomic commit involves a single feature.

It follows from the first two observations that it is possible for an FLT following our methodology to satisfy all three of the criteria above. The next two observations influence the training and indexing steps of our methodology, which have the conceptual effect of relating classes (or methods) to changeset topics. By contrast, the training and indexing steps of the standard methodology have the conceptual effect of relating classes to class topics (or methods to method topics).

To evaluate the new methodology, we used it to implement FLTs based on online LSI and online LDA. We next used two benchmarks to compare the accuracy of these FLTs to the accuracy of analogous FLTs following the standard methodology. Combined, the two benchmarks comprise over 600 defects and features from 4 open-source Java projects with both method- and class-level goldsets. Our evaluation results provide evidence that our new methodology is sound and that following it yields FLTs with similar accuracy to those following the standard methodology, but without the retraining costs.

The remainder of the paper is organized as follows. We first review background and related work (§II) We next present

our new methodology for FLTs (§III) and report evaluation results for the online-LDA-based FLT (§IV). We then conclude (§VII).

II. Background & Related Work

In this section, we review the standard methodology for document extraction and retrieval process used by snapshot-based FLTs, as well as related work on topic modeling and feature location.

A. Document Extraction and Retrieval Process

We use the following terminology to describe document extraction of a source code snapshot. A *word* is the basic unit of discrete data in a software lexicon and is a sequence of letters. A *token* is a sequence of non-whitespace characters containing one or more words. An *entity* is a named source element such as a method, and an *identifier* is a token representing the name of an entity. *Comments* and *literals* are sequences of tokens delimited by language-specific markers (e.g., /* */ and quotes). The *document* which corresponds to an entity is a sequence of words $d = (w_1, \ldots, w_m)$, and a *corpus* is a set of documents (e.g., methods) $D = (d_1, \ldots, d_n)$.

The left side of Figure 1a illustrates the document extraction process. A document extractor takes a source code snapshot as input and produces a corpus as output. Each document in the corpus contains the words associated with a source code entity, such as a class or method. The text extractor is the first part of the document extractor and parses the source code to produce a token stream for each document. The preprocessor is the second part of the document extractor. It applies a series of transformations to each token and produces one or more words from the token. The transformations commonly used are [3], [11], [12]:

- *Splitting*: separate tokens into constituent words based on common coding style conventions (e.g., the use of camel case or underscores) and on the presence of non-letters (e.g., punctuation or digits)
- *Normalizing*: replace each upper case letter with the corresponding lower case letter
- *Filtering*: remove common words such as articles (e.g., 'an' or 'the'), programming language keywords, standard library entity names, or short words

The right side of Figure 1a illustrates the retrieval process. The main prerequisite of the retrieval process is to build the search engine. The search engine is constructed from a topic model trained from a corpus and an index of that corpus inferred from that model. This means that an index is no more than each input document's thematic structure (i.e., the document's inferred topic distribution).

The primary function of the search engine is to rank documents in relation to the query [13]. First, when using a TM-based approach, the engine must first infer the thematic structure of the query. This allows for a pairwise classification of the query to each document in the index and ranks the documents based on the similarities of their thematic structures.

B. Latent Dirichlet Allocation

LDA [5] is a generative topic model. LDA models each document in a corpus of discrete data as a finite mixture over a set of topics and models each topic as an infinite mixture over a set of topic probabilities. That is, LDA models each document as a probability distribution indicating the likelihood that it expresses each topic and models each topic that it infers as a probability distribution indicating the likelihood of a word from the corpus being assigned to the topic.

Hoffman et al. [9] introduce *online LDA*, an online version of LDA. Online LDA allows the model to be updated incrementally without needing to know about the documents prior to model construction. Zhai and Boyd-Graber [14] introduce an extension of LDA in which the model also does not need to know about the corpus vocabulary prior to training. Teh et al. [15] introduce an LDA counterpart, the Hierarchical Dirichlet process (HDP), that learns the appropriate number of topics from the data, rather than needing configuration. Further, Wang et al. [16] further extend HDP to bring the algorithm online.

C. Feature Location

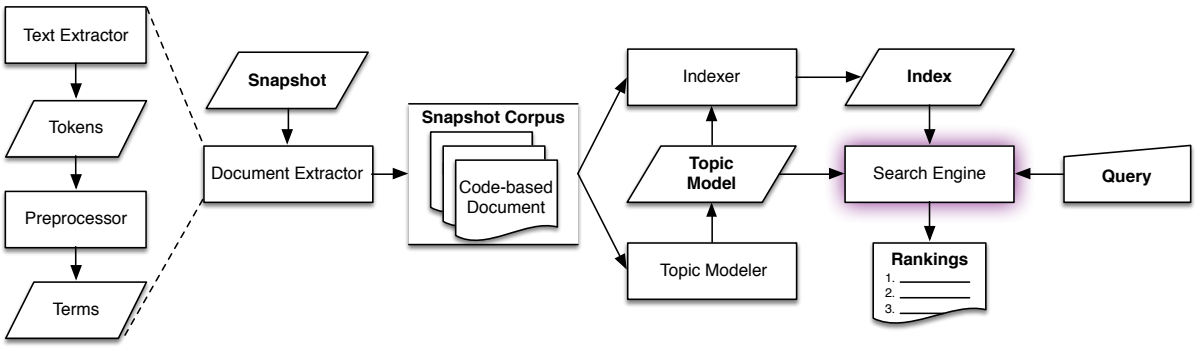
Feature location is the act of identifying the source code entity or entities that implement a feature [17]. Dit et al. [2] provide a taxonomy and survey of feature location in source code covering the scope of FLTs. They identify 89 works related to feature location in their systematic literature survey and extract 7 dimensions for their taxonomy. The primary dimension, type of analysis, can be used for categorization purposes and consists of four categories: dynamic, static, historical, and textual. Our methodology uses historical analysis (e.g., [18]) and textual analysis (e.g., [3]).

The most relevant FLTs, by Rao [10], [19], are described in the introduction. Other closely related work involves LSI-based FLTs [3], [18], [20]–[23] or LDA-based FLTs [8], [24]–[26].

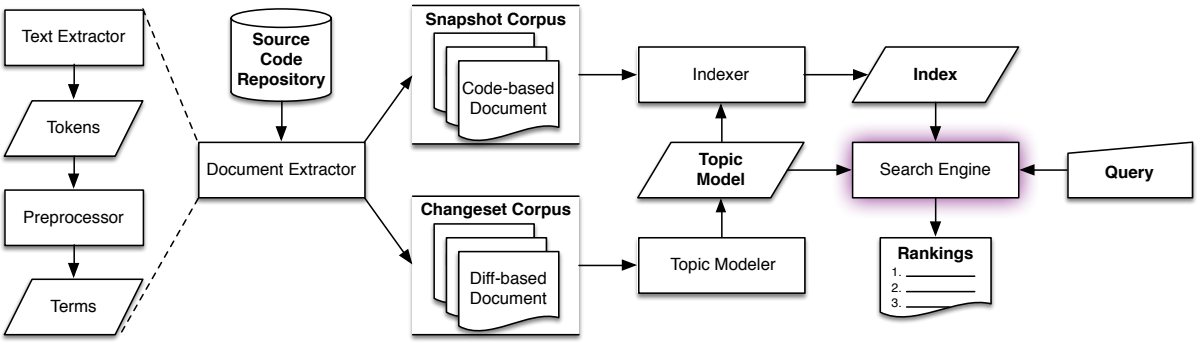
D. Modeling Software Repositories

Our work is also not the first to investigate ways to employ topic models on software repositories. Thomas et al. [27] present a study on how topics of a software project evolve over time. They present the *Diff* model, and closely resembles our work. However, their Diff model is much more coarse-grained and trains a topic model on changesets between two software snapshots, not changesets between two commits. Additionally, their goal in using this model is for modeling the evolution of topics, not for feature location.

Hindle et al. [28] present a technique that relates commits to requirements documents using LDA. They apply LDA to extract topics from issue reports, requirements documents, and commit messages. Their linking process relies on LDA inferencing to derive the topics of unseen documents. Hindle et al. [29] use a similar approach. Our methodology is based on the same inferencing concept and creates an index of source code entities from learned changeset topics.



(a) Topic-modeling-based feature location technique using snapshots



(b) Topic-modeling-based feature location technique using changesets

Fig. 1: Two feature location techniques side-by-side

III. Modeling Changeset Topics

In this section we describe how a TM-based FLT can use changesets.

A. Terminology

In addition to the terminology described in Section II, we use the following terminology to describe the document extraction and retrieval process of changesets.

A *diff* is a set of text which represents the differences between two texts. A *patch* is a set of instructions (i.e., diffs) that is used to transform one set of texts into another. *Context lines* denote text useful for transforming the text, but do not represent the differences. *Added lines* are lines which were added in order to transform the first text into the second. Similarly, *removed lines* are lines which are removed for this same purpose. A *changeset*, ideally, represents a single feature modification, addition, or deletion, which may crosscut many source code entities. A *commit* is a representation of a changeset in a version control system, such as Git or Subversion. Figure 2 shows an example changeset from Git.

B. Feature location using changesets

The overall difference in our methodology and the standard methodology described in Section II-A is minimal. For example, compare Figures 1a and 1b. In the changeset approach, we only need to replace the documents on which the topic model is trained while the remainder of the approach remains the same.

The changeset approach requires two types of document extraction: the snapshot of the state of source code at a commit of interest, such as a tagged release, and every changeset in the source code history leading up to the same commit of interest. The left side of Figure 1b illustrates the dual-document extraction approach.

The document extraction process for the snapshot remains the same as covered in Section II while the document extractor for the changesets parses each changeset for the removed, added, and context lines. From there, each line is tokenized by the text extractor. The same preprocessor transformations as before occur in both the snapshot and changesets. The snapshot vocabulary is always a subset of the changeset vocabulary [30].

The right side of Figure 1b illustrates the retrieval process. The key intuition to our methodology is that a topic model such as LDA or LSI can infer *any* document's topic proportions regardless of the documents used to train the model. In fact, this is also what determining the topic proportions of a user-created query relies on. Likewise, so are other unseen documents. In our approach, the seen documents are changesets and the unseen documents are the source code entities of the snapshot.

Hence, we train a topic model on the changeset corpus and use the model to index the snapshot corpus. Note that we never construct an index of the changeset documents on

```
diff --git a/src/java/net/sf/jabref/EntryEditor.java b/src/java/net/sf/jabref/EntryEditor.java
index 8c56723..6b4788e 100644
--- a/src/java/net/sf/jabref/EntryEditor.java
+++ b/src/java/net/sf/jabref/EntryEditor.java
@@ -669,7 +669,8 @@ public class EntryEditor extends JPanel implements VetoableChangeListener {
     public void storeCurrentEdit() {
         Component comp = Globals.focusListener.getFocused();
         if ((comp == source) || ((comp instanceof FieldEditor) && this.isAncestorOf(comp))) {
-            ((FieldEditor)comp).clearAutoCompleteSuggestion();
+            if (comp instanceof FieldEditor)
+                ((FieldEditor)comp).clearAutoCompleteSuggestion();
             storeFieldAction.actionPerformed(new ActionEvent(comp, 0, ""));
         }
     }
```

Fig. 2: Example of a `git diff`. This changeset addresses JabRef's Issue #2904968. Black or blue lines denote metadata about the change useful for patching. In particular, black lines represent context lines (beginning with a single space). Red lines (beginning with a single −) denote line removals, and green lines (beginning with a single +) denote line additions.

which the model is trained, nor do we use the snapshot corpus during training. In our approach, we only use the changesets to continuously update the topic model and only use the snapshot for indexing.

To leverage the online functionality of the topic models, we can also intermix the model training, indexing, and retrieval steps. First, we initialize a model in online mode. Then, as changes are made, the model is updated with the new changesets as they are committed. That is, with changesets, we incrementally update a model and can query it at any moment. Our historical simulation (§ IV-C) relies on this insight.

C. Why Changesets?

We choose to train the model on changesets, rather than another source of information, because they also represent what we are primarily interested in: program features. A single changeset provides text of an addition, removal, or modification of a single feature. A developer can to some degree comprehend what a changeset accomplishes by examining it, such as during a code review, much like examining a source file directly.

While a snapshot corpus has documents that represent a program, a changeset corpus has documents that represent programming. If we consider every changeset affecting a particular source code entity, then we gain a sliding-window view of that source code entity over time and the contexts those changes were performed in. This is akin to summarizing code snippets with machine learning [31], where in our case a changeset gives a snippet-like view of the code required to complete a task. For example, in Figure 2, we can see the entire method being changed when the context lines are considered.

Additionally, Vasa et al. [32] observe that code rarely changes as software evolves. The implication is that the topic modeler will see changesets containing the same source code entity only a few times, perhaps only once. Since topic modeling a snapshot only sees an entity once, topic modeling a changeset can miss no information.

Using changesets also implies that the topic model may gain some noisy information from these additional documents, especially removals. However, Vasa et al. also observe that code is less likely to be removed than it is to be changed. This implies that the noisy information would likely remain in both snapshot-based models and changeset-based models.

Indeed, it appears desirable to remove changesets from the model that are old and no longer relevant to the current snapshot of the system. There would be no need for this because online LDA already contains features for increasing the influence newer documents have on the model, thereby decaying the effect of the older documents on the model.

IV. Study

In this section we describe the design of a study in which we compare our new methodology with the current practice. We describe the case study using the Goal-Question-Metric approach [33]. We discuss the results of using LDA as our topic modeler, and exclude the LSI discussion for brevity. Further, the data and source code for the full case study is available in this paper's online appendix[1].

A. Definition and Context

Our *goal* is to evaluate the effectiveness of TM-based FLTs trained on changesets. The *quality focus* of the study is on informing development decisions and policy changes that could lead to software with fewer defects. The *perspective* of the study is of a researcher, developer, or project manager who wishes to gain understanding of the concepts or features implemented in the source code. The *context* of the study spans the version histories of 14 open source systems.

Toward the achievement of our goal, we pose the following research questions:

RQ1 Is a changeset-based FLT as accurate as a snapshot-based FLT?

With our new methodology, we also gain the opportunity to simulate how the FLT would perform in a real development environment, an evaluation technique not previously feasible due to the run-time of the experiment.

RQ2 Does the accuracy of a changeset-based FLT fluctuate as a project evolves?

[1] http://christop.club/publications/data/Corley-etal_2015

TABLE I: Subject Systems and Goldset Sizes

Subject System	Features	Classes	Methods
ArgoUML v0.22	91	287	701
ArgoUML v0.24	52	154	357
ArgoUML v0.26.2	209	706	1560
Jabref v2.6	39	131	280
jEdit v4.3	150	361	748
muCommander v0.8.5	92	303	717
Total	633	1942	4363

At a high level, our goal is to determine the feasibility of using changesets to train topic models for feature location, especially in realistic development scenarios.

In the remainder of this section we introduce the subjects of our study, describe the setting of our study, our data collection and analysis procedures, and report the results of the study using LDA.

B. Subject Software Systems

Each of our subject software systems come from two publicly-available datasets. The first is a dataset of six software systems by Dit et al. [34] and contains method-level goldsets. This dataset was automatically extracted from changesets that relate to the queries (issue reports). The second is a dataset of 14 software systems by Moreno et al. [35] and contains class-level goldsets. The six software systems in the first dataset also appear in the second, supplying us with both class- and method-level goldsets for the queries. We only consider the systems where the datasets overlap.

ArgoUML is a UML diagramming tool[2]. jEdit is a text editor[3]. JabRef is a BibTeX bibliography management tool[4]. muCommander is a cross-platform file manager[5].

C. Methodology

For snapshots, the process is straightforward and corresponds to Figure 1a. First, we train a model on the snapshot corpus using batch training. That is, the model can see all documents in the corpus at once. Then, we infer an index of topic distributions with the snapshot corpus. For each query in the dataset, we infer the query's topic distribution and rank each entity in the index with pairwise comparisons.

In terms of changesets, the process varies slightly from a snapshot approach, as shown in Figure 1b. First, we train a model on the changeset corpus using batch training. Second, we infer an index of topic distributions with the snapshot corpus. Note that we *do not* infer topic distributions with the changeset corpus on which the model was built. Finally, for each query in the dataset, we infer the query's topic distribution and rank each entity in the snapshot index with pairwise comparisons.

For the historical simulation, we take a slightly different approach. We first determine which commits relate to each query (or issue) and partition mini-batches out of the changesets. We then proceed by initializing a model for online training. Using each mini-batch, or partition, we update the model. Then, we infer an index of topic distributions with the snapshot corpus at the commit the partition ends on. We also obtain a topic distribution for each query related to the commit. For each query, we infer the query's topic distribution and rank each entity in the snapshot index with pairwise comparisons. Finally, we continue by updating the model with the next mini-batch.

Since the Dit et al. dataset was extracted from the commit that implemented the change, our partitioning is inclusive of that commit. That is, we update the model with the linked commit and infer the snapshot index from that commit. This allows our evaluations to capture any entities added to address the issue report, as well as changed entities, but does not capture any entities that were removed by the change.

D. Setting

Our document extraction process is shown on the left side of Figure 1b. We implemented our document extractor in Python v2.7 using the Dulwich library[6] for interacting with the source code repository and Teaser[7] for parsing source code. We extract documents from both a snapshot of the repository at a tagged snapshot and each commit reachable from that tag's commit. The same preprocessing steps are employed on all extracted documents.

For our document extraction from a snapshot, we first parse each Java file using our tool, Teaser, which is a text extractor implemented in Java using an open source Java 1.5 grammar and ANTLR v3. The tool extracts documents from the chosen source code entity type, either methods or classes. We consider interfaces, enumerations, and annotation types to also be a class. The text of inner an entity (e.g., a method inside an anonymous class) is only attributed to that entity, and not the containing one. Comments, literals, and identifiers within a entity are considered as text of the entity. Block comments immediately preceding an entity are also included in this text.

To extract text from the changesets, we look at the `git diff` between two commits. In our changeset text extractor, we extract all text related to the change, e.g., context, removed, and added lines; metadata lines are ignored. Note that we do not consider where the text originates from, only that it is text changed by the commit.

After extracting tokens, we split the tokens based on camel case, underscores, and non-letters. We only keep the split tokens; original tokens are discarded. We normalize to lower case before filtering non-letters, English stop words [36], Java keywords, and words shorter than three characters long. We do not stem words.

We implemented our modeling using the Python library Gensim [37], version 0.10.3. We use the same configurations on each subject system. We do not try to adjust parameters between the different systems to attempt to find a better, or best, solution; rather, we leave them the same to reduce confounding

[2] http://argouml.tigris.org/ [3] http://www.jedit.org/
[4] http://jabref.sourceforge.net/ [5] http://www.mucommander.com/
[6] http://www.samba.org/~jelmer/dulwich/ [7] https://github.com/nkraft/teaser

variables. We do realize that this may lead to topic models that may not be best-suited for feature location on a particular subject system. However, this constraint gives us confidence that the measurements collected are fair and that the results are not influenced by selective parameter tweaking. Again, our goal is to show the performance of the changeset-based FLT against snapshot-based FLT under the same conditions.

Gensim's LDA implementation is based on an online LDA by Hoffman et al. [9] and uses variational inference instead of a collapsed Gibbs sampler. Unlike Gibbs sampling, in order to ensure that the model converges for each document, we allow LDA to see each mini-batch 5 times by setting Gensim's initialization parameter `passes` to this value and allowing the inference step 1000 iterations over a document. We set the following LDA parameters for all 6 systems: 500 topics, a symmetric $\alpha = 1/K$, and a symmetric $\eta = 1/K$. These are default values for α and η in Gensim, and have been found to work well for the FLT task [25].

For the historical simulation, we found it beneficial to consider two other parameters: κ and τ_0. As noted in Hoffman et al. [9], it is beneficial to adjust κ and τ_0 to higher values for smaller mini-batches. These two parameters control how much influence a new mini-batch has on the model when training. We follow the recommendations in Hoffman et al. choosing $\tau_0 = 1024$ and $\kappa = 0.9$ for all systems, because the historical simulation often has mini-batch sizes in single digits.

E. Data Collection and Analysis

To evaluate the performance of a TM-based FLT we cannot use measures such as precision and recall. This is because the FLT creates the rankings pairwise, causing every entity being searched to appear in the rankings. Poshyvanyk et al. define an effectiveness measure that can be used for TM-based FLTs [7]. The effectiveness measure is the rank of the first relevant document and represents the number of source code entities a developer would have to view before reaching a relevant one. The effectiveness measure allows evaluating the FLT by using the mean reciprocal rank (MRR) [38]:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{e_i} \tag{1}$$

where Q is the set of queries and e_i is the effectiveness measure for some query Q_i.

To answer RQ1, we run the experiment on the snapshot and changeset corpora as outlined in Section IV-C. We then calculate the MRR between the two sets of effectiveness measures. We use the Wilcoxon signed-rank test with Holm correction to determine the statistical significance of the difference between the two rankings. To answer RQ2, we run the historical simulation as outlined in Section IV-C and compare it to the results of batch changesets from RQ1. Again, we calculate the MRR and use the Wilcoxon signed-rank test.

F. Results

RQ1 asks how well a topic model trained on changesets performs against one trained on source code entities. Table II and Table III summarize the results of each subject system when evaluated at the class and method granularity, respectively. In each of the tables, we bold which of the two MRRs is greater. Since our goal is to show that training with changesets is just as good, or better than, training on snapshots, we only care about statistical significance when the MRR is in favor of snapshots.

TABLE II: **RQ1**: MRR and p-values of class-level Batch

Subject System	Snapshot	Changeset	p-value
ArgoUML v0.22	0.059154	**0.099735**	$p < 0.01$
ArgoUML v0.24	0.168083	**0.188884**	$p = 0.013930$
ArgoUML v0.26.2	**0.186212**	0.144408	$p < 0.01$
JabRef v2.6	**0.292574**	0.190469	$p = 0.501962$
jEdit v4.3	**0.268085**	0.173586	$p = 0.019540$
muCommander v0.8.5	**0.276343**	0.183420	$p = 0.011480$
All	**0.205412**	0.157036	$p < 0.01$

TABLE III: **RQ1**: MRR and p-values of method-level Batch

Subject System	Snapshot	Changeset	p-value
ArgoUML v0.22	0.038286	**0.064189**	$p = 0.268417$
ArgoUML v0.24	**0.068024**	0.064706	$p = 0.699388$
ArgoUML v0.26.2	**0.077683**	0.055174	$p = 0.445880$
JabRef v2.6	0.069135	**0.081123**	$p = 0.076284$
jEdit v4.3	0.039041	**0.068860**	$p = 0.078042$
muCommander v0.8.5	0.038749	**0.050925**	$p = 0.160519$
All	0.055945	**0.061465**	$p = 0.067154$

For LDA at the class-level we note an improvement in MRR for 2 of the 6 systems when using changesets. Additionally, 1 of these 2 systems were statistically significant at $p < 0.01$. Only 1 of the 4 systems with MRR in favor of snapshots were statistically significant. Hence, changeset topics perform just as well as snapshot topics at the class-level 5 of the 6 times.

For LDA at the method-level we note an improvement in MRR for 4 of the 6 systems when using changesets. None of these were statistically significant at $p < 0.01$. This suggests that changeset topics are as accurate as snapshot topics at the method-level, especially since there is a lack of statistical significance for *any* of the cases.

RQ1: Changeset-based FLTs are as accurate as snapshot-based FLTs.

RQ2 asks how well a simulation of using a topic model would perform as it were to be used in real-time. This is a much closer evaluation of an FLT to it being used in an actual development environment. Table IV and Table V summarize the results of each subject system when evaluated at the class and method granularity, respectively. In each of the tables, we bold which of the two MRRs is greater. Again, since our goal is to show that temporal considerations must be given during FLT

TABLE IV: **RQ2**: MRR and p-values of class-level Temporal

Subject System	Batch	Temporal	p-value
ArgoUML v0.22	0.099735	**0.121756**	$p = 0.280057$
ArgoUML v0.24	**0.188884**	0.182267	$p = 0.449599$
ArgoUML v0.26.2	0.149223	**0.157971**	$p < 0.01$
JabRef v2.6	0.203930	**0.232776**	$p < 0.01$
jEdit v4.3	0.174738	**0.219397**	$p < 0.01$
muCommander v0.8.5	0.185228	**0.261738**	$p < 0.01$
All	0.159593	**0.189059**	$p < 0.01$

TABLE V: **RQ2**: MRR and p-values of method-level Temporal

Subject System	Batch	Temporal	p-value
ArgoUML v0.22	**0.065629**	0.054402	$p = 0.025760$
ArgoUML v0.24	0.063462	**0.087268**	$p = 0.025119$
ArgoUML v0.26.2	0.059563	**0.072729**	$p = 0.127751$
JabRef v2.6	**0.101983**	0.064746	$p = 0.069284$
jEdit v4.3	0.068859	**0.071710**	$p = 0.466658$
muCommander v0.8.5	0.052569	**0.066859**	$p < 0.01$
All	0.064204	**0.069749**	$p < 0.01$

evaluation, we only care about statistical significance when the MRR is in favor of batch.

At the class-level we note an improvement in MRR for 5 of the 6 systems with 4 of these 5 statistically significant at $p < 0.01$. The one result in favor of batch changesets, for ArgoUML v0.24, was not statistically significant. A the method-level we note an improvement in MRR for 4 of the 6 systems with 2 of the 4 statistically significant at $p < 0.01$.

RQ2: Historical simulation reveals that the accuracy of the changeset-based FLT is consistent as a project evolves and is actually higher than indicated by batch evaluation.

V. Discussion

The results outlined in the previous section warrants some qualitative discussion. In particular, our analysis shows significant affects between snapshots and changesets, and between batch changesets and changesets in the simulated environment. The results are mixed between each and are not conclusive. However, we argue this is desirable to show that the accuracy of a changeset-based FLT is similar to that of a snapshot-based FLT but without the retraining cost.

A. RQ1

Figure 3 shows the effectiveness measures for methods and classes across all systems. The figure suggests that snapshot-based models and changeset-based models have similar results overall with changesets performing slightly better, but does not help to understand how each feature query performs for each model. With respect to RQ1, we will investigate the queries and effectiveness measures between the batch snapshot and batch changesets in detail.

For the 632 successful queries of classes, each query returns the same effectiveness measure 28 out of 632 times, or about 4.4% of the time. Of these 28, 17 of them all return an effectiveness measure of 1 (the best possible measure). For 159 queries (25.2%), the effectiveness measure is within 10 ranks of each other. For 291 queries (46.0%), the effectiveness measure is within 50 ranks of each other. The remaining 341 queries (53.9%) perform noticeably different (> 50 ranks apart).

For the 629 successful queries of methods, each query returns the same effectiveness measure 12 out of 629 times, or about 1.9% of the time. Of these 12, 7 return an effectiveness measure of 1 (the best possible measure). For 65 queries (10.3%), the effectiveness measure is within 10 ranks of each other. For 151 queries (24.0%), the effectiveness measure is within 50 ranks of each other. The remaining 478 queries (75.9%) perform noticeably different (> 50 ranks apart).

B. RQ2

Figure 3 also shows the effectiveness measures for methods and classes across the 6 systems considered in RQ2. The figure shows that the historical simulation outperforms both batch evaluations, but does not help to understand how each feature query performs for each model. With respect to RQ2, we will investigate the queries and effectiveness measures between the historical simulation and the batch evaluations in detail.

For the 603 successful queries of classes, each query returns the same effectiveness measure 8 out of 603 times, or about 1.3% of the time. Of these 8, 7 return an effectiveness measure of 1 (the best possible measure). For 111 queries (18.4%), the effectiveness measure is within 10 ranks of each other. For 230 queries (38.1%), the effectiveness measure is within 50 ranks of each other. The remaining 373 queries (61.8%) perform noticeably different (> 50 ranks apart).

For the 595 successful queries of methods, each query returns the same effectiveness measure 3 out of 595 times, or about 0.5% of the time. Of these 3, all return an effectiveness measure of 1 (the best possible measure). For 23 queries (3.9%), the effectiveness measure is within 10 ranks of each other. For 77 queries (12.9%), the effectiveness measure is within 50 ranks of each other. The remaining 518 queries (87.0%) perform noticeably different (> 50 ranks apart).

C. Situations

In this study, we've also asked two research questions which lead to two distinct comparisons. First, we compare a batch TM-based FLT trained on the changesets of a project's history to one trained on the snapshot of source code entities. Second, we compare a batch TM-based FLT trained on changesets to a online TM-based FLT trained on the same changesets over time. Our results are mixed between the research questions, hence we end up with four possible situations; we will now discuss each of these situations in detail.

1) Batch changesets are better than batch snapshot and *batch changesets are better than changesets in the simulated environment:* This situation occurs in 1 out of 6 systems at the class-level, and 1 out of 6 systems at the method-level. We

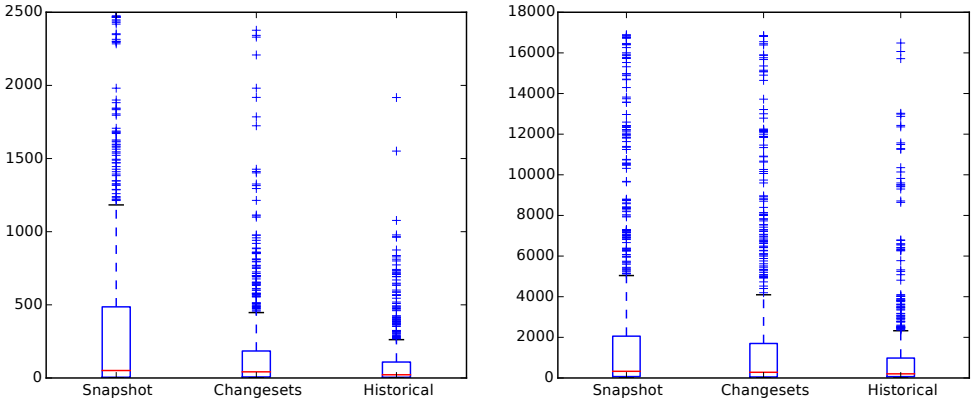


Fig. 3: Effectiveness measures for classes (left) and methods (right) across all 6 systems

hypothesize that this is due to the nature of the batch evaluation versus the historical simulation. In the batch evaluation, the model is trained on all data before being queried, while in the historical simulation the model is trained on partial data before being queried. This allows for the batch model to be more accurate because it is trained on more data and reveals feature location research evaluations may not be accurately portraying how an FLT would perform in a real scenario.

2) Batch changesets are better than batch snapshot and *changesets in the simulated environment are better than batch changesets:* This situation occurs in 1 out of 6 systems at the class-level, and 3 out of 6 systems at the method-level. We hypothesize that this is due to the same previous reason, but that historical simulation more accurately captures the correct state of the system (i.e., the source code entities) at the point in time when querying is done. Since querying on the batch models is after the model is completely trained, there may be source code entities that do not exist in the system anymore that were at one time changed to complete a certain task. Again, the historical simulation better captures this scenario.

3) Batch snapshot are better than batch changeset and *changesets in the simulated environment are better than batch changesets:* This situation occurs in 4 out of 6 systems at the class-level, and 2 out of 6 systems at the method-level. Similarly, this could be because of how the models are trained. Although the batch changesets performed worse in both cases, it does improve during historical simulation. This does not mean that changesets are bad, but more accurately model the system over time.

4) Batch snapshot are better than batch changeset and *batch changesets are better than changesets in the simulated environment:* We note that this situation never occurs. This also supports the hypothesis that historical simulation more accurately portrays the system over time. However, we cannot conclude this without also historically simulating snapshot TM-based FLTs.

VI. Threats To Validity

Our study has limitations that impact the validity of our findings, as well as our ability to generalize them. We describe some of these limitations and their impacts.

Threats to internal validity include possible defects in our tool chain and possible errors in our execution of the study procedure, the presence of which might affect the accuracy of our results and the conclusions we draw from them. We controlled for these threats by testing our tool chain and by assessing the quality of our data. Because we applied the same tool chain to all subject systems, any errors are systematic and are unlikely to affect our results substantially.

Another threat to internal validity pertains to the value of parameters such as K that we selected for all models trained. We decided that the changeset and snapshot models should have the same parameters to help facilitate evaluation and comparison. We argue that our study is not about selecting the best parameters, but to show that our changeset TM-based FLT approach is reasonable.

Threats to external validity concern the extent to which we can generalize our results. The subjects of our study comprise fourteen open source projects in Java, so we cannot generalize our results to systems implemented in other languages. However, the systems are of different sizes, are from different domains, and have characteristics in common with those of systems developed in industry.

Threats to construct validity concern measurements accurately reflecting the features of interest. A possible threat to construct validity is our benchmarks. Errors in the datasets could result in inaccurate effectiveness measures. The datasets were produced by other researchers, are publicly available, and have been used in previous research [34], [35], [39]. While both datasets extracted source code entities automatically from changesets and patches, previous work shows this approach is on par with manual extraction [40].

VII. Conclusions & Future Work

In this paper we conducted a study on modeling the topics of changesets in comparison to the traditional snapshot approach.

We use latent Dirichlet allocation (LDA) to extract linguistic topics from changesets and snapshots (releases).

We addressed two research questions regarding the performance of a TM-based FLT trained on changesets. First, we compare a batch TM-based FLT trained on the changesets of a project's history to one trained on the snapshot of source code entities. We found that changesets can perform as well as or better than snapshots. Second, we compare a batch TM-based FLT trained on changesets to a historical simulation of a TM-based FLT trained on the same changesets over time. We show that the historical simulation more accurately portrays how a FLT would execute in a real environment.

Our results encourage the idea that there is still much to explore in the area of feature location. What other untapped resources might be available? We show changesets are yet another viable resource researchers and practitioners should be taking advantage of for the feature location task. Our results also show that research remains not only in improving accuracies of FLTs, but also in solving the practical aspects of building FLTs that are robust *and* agile enough to keep up with fast-changing software.

Future work includes deploying this approach in a development environment. Since the source code to our approach is online, we encourage other researchers to investigate this future work as well. We also would like to expand the simulation parts of this study to include both snapshots and changesets. It would be particularly useful to compare results between batch snapshots and simulated snapshots.

Additional future work exists in regard to configuration. In a changeset it may be desirable to parse further for source code entities using island grammar parsing [41]. It may also be desirable to only use portions of the changeset, such as only using added or removed lines, or extracting changes between the abstract syntax trees [42]. Most importantly, like previous work [25], it would be wise to further investigate the effects of the two online LDA variables, τ_0 and κ. We leave these options for future work.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments and helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. 1156563.

REFERENCES

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[3] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the 11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 214–223.

[4] D. Blei, "Probabilistic topic models," *Communications of the ACM*, vol. 55, no. 4, pp. 77–84, Apr. 2012.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[6] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society of Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[7] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *Software Engineering, IEEE Transactions on*, vol. 33, no. 6, pp. 420–432, 2007.

[8] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[9] M. Hoffman, F. R. Bach, and D. M. Blei, "Online learning for latent dirichlet allocation," in *Advances in Neural Information Processing Systems 23*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, pp. 856–864.

[10] S. Rao, "Incremental update framework for efficient retrieval from software libraries for bug localization," Ph.D. dissertation, Purdue University, 2013.

[11] A. Marcus and T. Menzies, "Software is data too," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 229–232.

[12] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 3–12.

[13] B. Croft, D. Metzler, and T. Strohman, *Search engines : information retrieval in practice*. Boston: Addison-Wesley, 2010.

[14] K. Zhai and J. Boyd-Graber, "Online topic models with infinite vocabulary," in *Proc. Int'l Conf. on Machine Learning*, ser. JMLR: Workshop and Conference Proceedings, vol. 28, no. 1, 2013, pp. 561–569.

[15] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, "Hierarchical dirichlet processes," *Journal of the american statistical association*, vol. 101, no. 476, 2006.

[16] C. Wang, J. W. Paisley, and D. M. Blei, "Online variational inference for the hierarchical dirichlet process," in *International conference on artificial intelligence and statistics*, 2011, pp. 752–760.

[17] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002, pp. 271–278.

[18] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 446–465, 2005.

[19] S. Rao, H. Medeiros, and A. Kak, "An incremental update framework for efficient retrieval from software libraries for bug localization," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 62–71.

[20] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*. IEEE, 2006, pp. 137–148.

[21] D. Poshyvanyk and A. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*. Washington, DC, USA: IEEE, Jun. 2007, pp. 37–48.

[22] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature location via information retrieval based filtering of a single scenario execution trace," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 234–243.

[23] G. Scanniello and A. Marcus, "Clustering support for static concept location in source code," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*. IEEE, 2011, pp. 1–10.

[24] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*. IEEE, 2008, pp. 155–164.

[25] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[26] B. Bassett and N. A. Kraft, "Structural information based term weighting in text retrieval for feature location," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 133–141.

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

[28] A. Hindle, M. W. Godfrey, and R. C. Holt, "What's hot and what's not: Windowed developer topic analysis," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 339–348.

[29] A. Hindle, C. Bird, T. Zimmermann, and N. Nagappan, "Do topics make sense to managers and developers?" *Empirical Software Engineering*, pp. 1–37, 2014.

[30] C. S. Corley, K. L. Kashuda, D. S. May, and N. A. Kraft, "Modeling changeset topics," in *Proc. 4th Wksp. on Mining Unstructured Data*, 2014.

[31] A. T. T. Ying and M. P. Robillard, "Code fragment summarization," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, 2013.

[32] R. Vasa, J.-G. Schneider, and O. Nierstrasz, "The inevitable stability of software change," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 4–13.

[33] V. Basili, G. Caldiera, and H. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, vol. 2, no. 1994, pp. 528–532, 1994.

[34] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi, "A dataset from change history to support evaluation of software maintenance tasks," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 2013, pp. 131–134.

[35] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, 2014, pp. 151–160.

[36] C. Fox, "Lexical analysis and stoplists," in *Information Retrieval: Data Structures and Algorithms*, W. Frakes and R. Baeza-Yates, Eds. Prentice-Hall, 1992, pp. 102–130.

[37] R. Řehůřek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[38] E. M. Voorhees, "The trec-8 question answering track report." in *TREC*, vol. 99, 1999, pp. 77–82.

[39] M. Revelle, B. Dit, and D. Poshyvanyk, "Using data fusion and web mining to support feature location in software," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*. IEEE, 2010, pp. 14–23.

[40] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 2011, pp. 31–37.

[41] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 13–22.

[42] B. Fluri, M. Wursch, and H. Gall, "Do code and comments co-evolve? On the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.