The Impact of Classifier Configuration and Classifier Combination on Bug Localization

Stephen W. Thomas, *Student Member, IEEE*, Meiyappan Nagappan, Dorothea Blostein, *Member, IEEE*, and Ahmed E. Hassan

Abstract—Bug localization is the task of determining which source code entities are relevant to a bug report. Manual bug localization is labor intensive, since developers must consider thousands of source code entities. Current research builds bug localization classifiers, based on information retrieval models, to locate entities that are textually similar to the bug report. Current research, however, does not consider the effect of classifier *configuration*, i.e., all the parameter values that specify the behavior of a classifier. As such, it is unknown the effect of each parameter or which parameter values lead to the best performance. In this paper, we empirically investigate the effectiveness of a large space of classifier configurations, 3,172 in total. Further, we introduce a framework for combining the results of multiple classifier configurations, since classifier combination has shown promise in other domains. Through a detailed case study on over 8,000 bug reports from three large-scale projects, we make two main contributions. First, we show that the parameters of a classifier have a significant impact on its performance. Second, we show that combining multiple classifiers—whether those classifiers are hand-picked or randomly chosen relative to intelligently-defined subspaces of classifiers—improves the performance of even the best individual classifiers.

Index Terms—software maintenance, bug localization, information retrieval, VSM, LSI, LDA, classifier combination

1 Introduction

Developers typically use bug tracking databases, such as Bugzilla [37], to manage incoming bug reports in their software projects. For many projects, developers are overwhelmed by the volume of incoming bug reports that must be addressed. For example, in the Eclipse project, developers receive an average of 115 new bug reports every day; the Mozilla and IBM Jazz projects get 152 and 105 new reports per day, respectively. Developers must then spend considerable time and effort to read each new report and decide which source code entities are relevant for fixing the bug.

This task is known as *bug localization* [29], [40], [49], which is defined as a classification problem: given n source code entities and a bug report, classify the bug report as belonging to one of the n entities. The classifier returns a ranked list of possibly-relevant entities, along with a relevancy score for each entity in the list. An entity is considered *relevant* if it indeed needs to be modified to resolve the bug report, and *irrelevant* otherwise.

The developer uses the list of possibly-relevant entities to identify an entity related to the bug report and make the necessary modifications. After one relevant entity is identified using bug localization, developers can use change propagation techniques [1], [6] to identify any other entities that also need to be modified. Hence, the bug localization task is to find the first relevant entity; the task then switches to change propagation. Fully or partially automating bug localization can drastically reduce the development effort required to fix bugs, as much of the fixing time is currently spent manually locating the appropriate entities, which is both difficult [19] and expensive [53].

Current bug localization research uses information retrieval (IR) classifiers to locate source code entities that are textually similar to bug reports. However, current results are ambiguous and contradictory: some claim that the Vector Space Model (VSM) provides the best performance [49], while others claim that the latent Dirichlet allocation (LDA) model is best [29], while still others claim that a new IR model is needed [40]. These mixed results are due to the use of different datasets, different performance metrics, and different classifier configurations. A *classifier configuration* is a specification of all the parameters that define the behavior of a classifier, such as the way in which the source code is preprocessed, how terms are weighted, and the similarity metric between bug reports and source code entities.

In this paper, we aim to address this ambiguity by performing a large-scale empirical study to compare thousands of IR classifier configurations (see Table 2 for a description of the configurations that we use in our case studies) on a large quantity of bug reports. By using the same datasets and performance metrics, we can perform an apples-to-apples comparison of the various configurations, quantifying the impact of configuration on performance, and identifying which particular configurations yield the best performance. We find that configuration indeed has a large impact on performance:

- *S. W. Thomas is with the School of Computing, Queen's University, Canada.*
E-mail: sthomas@cs.queensu.ca
- *M. Nagappan, D. Blostein, and A. E. Hassan are with the School of Computing, Queen's University, Canada.*

some configurations are nearly useless, while others perform very well.

Further, we investigate the effect of combining the results of different classifiers, since combination has been shown to be beneficial in other domains [21] as well as in software engineering (e.g., defect prediction [60]). Classifier combinations (or *ensembles*) are known to be helpful in many situations, such as when the individual classifiers each have expertise in only a subset of the input cases, or when the performance of the individual classifiers tends to vary widely. In this paper, we present a framework for combining multiple classifiers that, as we later show, can together achieve better bug localization results than any single classifier. The main intuition behind classifier combination is that when a particular source code entity is returned high in the lists of many classifiers, then we can guess with high confidence that the entity is indeed relevant for the bug report. Our framework easily extends to any kind of bug localization classifier: we can combine IR-based classifiers with dynamic analysis classifiers, defect prediction classifiers, or any classifier that somehow solves the bug localization problem. As long as the classifiers are fairly uncorrelated in their wrong answers (i.e., the classifiers tend to make different mistakes from each other), then combining them will likely improve performance [21]. Given the nature of the bug localization problem, which has many possible wrong answers for a given bug report (i.e., source code entities that are unrelated to the bug report), combining models has the potential to perform well [35].

We perform a case study on three large-scale projects to study the performance of thousands of classifier configurations. Using these results as a baseline of performance, we investigate the performance of combinations of classifiers. We find that:

- The performance of a classifier is highly sensitive to its configuration. For example, the "average" configuration of a classifier in Eclipse achieves half the performance of the best configuration.
- Combining classifiers almost always improves performance, often by a significant amount. This is true when the best individual classifiers are combined, and also when random classifiers (which are chosen from specific subsets of classifiers, where each subset is defined to decrease the correlation between their wrong answers) are combined.

We provide our data, results, and tools online [55] to encourage others to replicate and extend our work.

The remainder of this paper is organized as follows. Section 2 introduces IR model basics and describes related work and its limitations. Section 3 presents a framework to define and analyze classifier configurations. Section 4 uses the framework to conduct an extensive case study to evaluate the performance of various classifier configurations. Section 5 presents a framework for combining classifiers, and in Section 6 we investigate the results of such combinations. We summarize and discuss key findings in Section 7 and list potential threats to the validity of our results in Section 8. We conclude and outline future work in Section 9.

2 Background and Related Work

All published bug localization research to date builds classifiers using IR models. In this section, we introduce IR models and describe existing IR-based bug localization and concept/feature location approaches.

2.1 Information Retrieval Models

Information retrieval is the study of querying for text within a collection of documents [31]. For example, the Google search engine uses IR techniques to help users find snippets of text in web pages. The "Find Documents" function in a typical operating system is based on the same theory, although applied at a much smaller scale.

IR-based bug localization classifiers use IR models to find textual similarities between a bug report (i.e., query) and the source code entities (i.e., documents). For example, if a bug report contains the words , *"Drop 20 bytes off each imgRequest object"*, then an IR model looks for entities which contain these words (*"drop"*, *"bytes"*, *"imgRequest"*, etc.). When a bug report and entity contain many shared words, then an IR-based classifier gives the entity a high relevancy score.

IR-based classifiers contain several parameters that control their behavior. Specifying a value for all parameters fully defines the ***configuration*** of a classifier. Common to all IR-based classifiers are parameters to govern how the input textual data is represented and preprocessed:

1) Which parts of the source code should be considered: the comments, identifier names, or some other representation, such as the previous bug reports linked to each source code entity?
2) Which parts of the bug report should be considered: the title only, description only, or both?
3) How should the source code and bug report be preprocessed? Should compound identifier names (e.g., *imgRequest*) be split? Should common stop words be removed? Should words be stemmed to their base form?

After these parameters are configured, each IR model has its own set of additional parameters that control term weighting, reduction factors, similarity metrics, and other aspects.

In the remainder of this section, we describe three standard and widely-used IR models: the Vector Space Model; Latent Semantic Indexing, an enhancement to the Vector Space Model; and latent Dirichlet allocation, a probabilistic topic model. We then discuss the various preprocessing steps that can be applied to the source code and bug reports.

2.1.1 Vector Space Model (VSM)

The Vector Space Model is a simple algebraic model based on the *term-document* matrix of a corpus [52]. The term-document matrix is an $m \times n$ matrix whose rows represent individual terms (i.e, words) and columns represent individual documents. The i^{th},j^{th} entry in the matrix is the weight of term w_i in document d_j. VSM represents documents by their column vector in the term-document matrix: a vector containing the weights of the words present in the document, and zeros otherwise. The similarity between two documents is calculated by comparing their two vectors. In VSM, two documents will only be deemed similar if they contain at least one shared term; the more shared terms they have, the higher their similarity score will be.

VSM uses the following parameters:

- Term weighting (*TW*): the weight of a term in a document. Popular values for this parameter are raw frequency (i.e., the number of occurrences of the term in the document), or tf-idf (term frequency, inverse document frequency) [31].
- Similarity metric (*Sim*): the similarity between two document vectors. Popular parameter values are Euclidean distance, cosine distance, Hellinger distance, or KL divergence.

2.1.2 Latent Semantic Indexing (LSI)

Latent Semantic Indexing, or Latent Semantic Analysis (LSA), is an extension to VSM in which singular value decomposition (SVD) is used to as a means to project the original term-document matrix into three new matrices: a topic-document matrix D; a term-topic matrix T; and a diagonal matrix S of eigenvalues [14]. Importantly, LSI reduces the rank of D and T to rank K, where K is a parameter provided by the user. During the projection, terms which are related by collocation (i.e., terms which often occur in the same documents) are grouped together into "concepts," or sometimes called "topics." For example, a GUI-related topic might contain the words "mouse", "click", "left", and "scroll", because these words tend to appear in the same documents. The reduced dimensionality of the topic-document matrix has increased performance over VSM when dealing with polysemy and synonymy [2]. For example, documents can be deemed similar even if they do not share any terms, but instead share terms from the same topic (e.g., document 1 contains "mouse" and document two contains "click").

In LSI, documents are still represented as column vectors, although the vectors now contain the weight of topics rather than the weights of single terms. The same similarity measures can be used to determine the similarity between two documents.

LSI uses the following parameters:

- Term weighting (*TW*): similar to the VSM *TW*.
- Number of topics (K): controls how many topics are kept during the SVD reduction.
- Similarity metric (*Sim*): similar to the VSM *Sim*.

2.1.3 Latent Dirichlet Allocation (LDA)

Latent Dirichlet allocation [5] is a popular statistical topic model which provides a means to automatically index, search, and cluster documents that are unstructured and unlabeled [4]. Like LSI, LDA accomplishes these tasks by first discovering a set of "topics" within the documents, and then representing each document as a mixture of topics. The key difference between LSI and LDA is the method used to generate topics. In LSI, topics are a byproduct of the SVD reduction of the term-document matrix. In LDA, topics are explicitly created through a *generative* process, using machine learning algorithms (such as Gibbs sampling) to iteratively deduce which words are present in which topics, and which topics are present in which documents. While the generative process of LDA enjoys several theoretical advantages over LSI and VSM, such as model checking and no assumptions about the distribution of term counts in the corpus, the results of LDA in practical IR studies have thus far been mixed [29], [49].

LDA uses the following parameters:

- Number of topics (K): controls how many topics are created.
- α: a document-topic smoothing parameter.
- β: a word-topic smoothing parameter.
- Number of iterations (*iters*): number of sampling iterations in the generative process. (In some recent implementations of LDA, the number of iterations is not required as an input parameter.)
- Similarity metric (*Sim*): similar to the VSM *Sim*.

2.1.4 Data Preprocessing

Before IR models are applied to source code and bug reports, several preprocessing steps are generally taken in an effort to reduce noise and improve the resulting models.

- Characters related to the syntax of the programming language (e.g., "`&&`", "`->`") are removed; programming language keywords (e.g., "`if`", "`while`") are removed.
- Identifier names are split, using regular expressions, into multiple parts based on common naming conventions, such as camel case (`oneTwo`), underscores (`one_two`), dot separators (`one.two`), and capitalization changes (`ONETwo`) [15], [20]. Recently, researchers have proposed more advanced techniques to split identifiers, based on speech recognition [30], automatic expansion [27], and mining source code [17], which may be more language-independent than simple regular expressions.
- Common English-language stop words (e.g., "the", "it", "on") are removed. In addition, custom stop word lists can be used, such as domain-specific jargon lists.

- Word stemming is applied to find the root of each word (e.g., “changing” and “changes” both become “chang”), typically using the Porter algorithm [45].

The main idea behind these optional steps is to capture developers’ intentions, which are thought to be encoded within the identifier names and comments in the source code [46]. The rest of the source code (i.e., special syntax, language keywords, and stop words) are viewed as noise and will not be beneficial as input for IR models. Section 4.1 describes which preprocessing steps we consider in this paper. For replication purposes, we provide our preprocessing tool online [55].

2.2 Existing IR-based Bug Localization Approaches

Researchers have explored the use of IR models for bug localization. For example, Lukins et al. compare the performance of LSI and LDA using three small case studies [28], [29]. The authors build the two IR classifiers on the identifiers and comments of the source code and compute the similarity between a bug report and each source code entity using the cosine and conditional probability similarity metrics. By performing case studies on Eclipse and Mozilla (on a total of 3 and 5 bug reports, respectively), the authors find that LDA often outperforms LSI. We note that the authors use manual query expansion, which may influence their results.

Nguyen et al. introduce a new topic model based on LDA, called BugScout, in an effort to improve bug localization performance [40]. BugScout explicitly considers past bug reports, in addition to identifiers and comments, when representing source code documents, using the two data sources concurrently to identify key technical concepts. The authors apply BugScout to four different projects and find that BugScout improves performance by up to 20% over LDA applied only to source code.

Rao and Kak compare several IR models for bug localization, including VSM, LSI, and LDA, as well as various combinations [49]. The authors perform a case study on a small dataset, iBUGS [12], and conclude that simpler IR models often outperform more sophisticated models.

Limitations of current research. In current research, researchers only consider a single or a few configurations of the classifiers (see Table 1), often with no justification given for why each parameter value was chosen out of the large space of possible values. Worse, many parameter values are left unspecified, making replication of their results difficult or impossible. Given that there are several choices for each parameter in the configuration, and the parameters are independent, there are thousands of possible configurations for each underlying IR model. The effectiveness of each configuration—which parameters are important, and which parameter values work best—is currently unknown. As a result, researchers and practitioners are left to guess which configuration to use in their project.

2.3 IR-based Concept/Feature Location Approaches

Closely related to IR-based bug localization is the problem of IR-based concept (or feature) location. In both problems, the goal is to identify a source code entity that is relevant to a given query. The difference is that in bug localization the query is the text within a bug report, whereas in concept location the query is manually created by the developer.

LSI was first used for concept location by Marcus et al. [32]. The authors found that LSI provides better results than existing approaches at the time, such as regular expressions and dependency graphs.

Researchers have also combined various approaches to perform concept location. Poshyvanyk et al., for example, combine LSI with a dynamic feature location approach called scenario-based probabilistic ranking; these two approaches operate on different datasets and use different analysis methods [46]. The results of the combined approach are much better than either individual approach, as evidenced by two case studies on large projects. Poshyvanyk et al. combine LSI and Formal Concept Analysis to achieve similar effects [47]. Cleary et al. combine several IR models with Natural Language processing techniques and conclude that NLP techniques do not significantly improve results [10]. Finally, Revelle et al. combine LSI, dynamic analysis, and web mining algorithms for feature location and find that the combination outperforms any of the individual approaches [50].

3 Configuration Framework

As illustrated in Section 2.1, the use of IR classifiers in bug localization requires the definition of many parameters. In fact, since there are so many parameters, and some parameters can take on any numeric value, there are effectively an infinite number of possible configurations of IR classifiers. Researchers in the data mining community argue that in the ideal world, algorithms (of any kind) should require no tuning parameters, to avoid potential bias by the user [23]. However, such an ideal is not often met, and the reality of IR classifiers is that they contain many parameters. Unfortunately, as we showed in Section 2.2, current bug localization research uses manual selection of parameter values and considers only a tiny fraction of all possible configurations.

In this section, we propose a *configuration framework* for analyzing the various configurations of a classifier. We will use the framework in Section 4 for analyzing bug localization classifiers, but the framework is general so that it can be used in any domain where many possible configurations of a classifier need to be analyzed.

We summarize the framework as follows:

1) Define a set of classifier configurations.
2) Execute each configuration to perform the task at hand, and measure the performance of each configuration using some effectiveness measure.

TABLE 1
Summary of the classifier configurations used by existing bug localization work, and their performance results. Question marks indicate that the parameters were not specified.

	Classifier configuration				Studied projects	Performance metric	Results*
	Entity rep.	Bug rep.	Preprocess	IR model			
Lukins et al. [28], [29]	Idents + comments	Title + descr.	Stem	LDA (K=100, α=50/K, β=0.01, *iters*=?, *Sim*=CP)	Mozilla	Mean rank	5.8
					Rhino	Mean rank	1–1,062
					Eclipse	Mean rank	492–11,234
Nguyen et al. [40]	Idents + comments	Title + descr.	Split+stem	LDA (K= 300, α=0.01, β=0.01, *iters*=?, *Sim*=cosine)	Jazz	Top-20	0.39
					Eclipse	Top-20	0.33
					AspectJ	Top-20	0.28
					ArgoUML	Top-20	0.34
				BugScout (K= 300, α=0.01, β=0.01, *iters*=?, *Sim*=cosine)	Jazz	Top-20	0.48
					Eclipse	Top-20	0.39
					AspectJ	Top-20	0.51
					ArgoUML	Top-20	0.45
Rao and Kak [49]	Idents	?	Split	VSM (*TW*=tf-idf, *Sim*=cosine)	AspectJ	MAP	0.0796
				LSI (*TW*=tf-idf, K=500, *Sim*=cosine)	AspectJ	MAP	0.0650
				LDA (K=150, α=0.33, β=0.01, *iters*=?, *Sim*=KL)	AspectJ	MAP	0.0125

* We present results to the best of our ability. Some papers only provide results in graph form, forcing us to estimate the exact value.

3) Analyze the performance of each configuration, and the various parameter settings, using, for example, Tukey's HSD statistical test.

We describe each step in more detail.

3.1 Define Configurations

As mentioned previously, a classifier is defined by a *configuration*—a set of parameter values that specify the behavior of the classifier. To define a configuration is to specify the value of each parameter. Some parameters may be categorical (e.g., which similarity metric is used) while others might be continuous (e.g., the number of topics in LSI or LDA).

We introduce here a notation that is useful for describing an individual configuration. The notation encodes the value (whether it's categorical or numerical) of each parameter, leaving no room for ambiguity. As an example, consider an IR classifier that takes two parameters X and Y, both integers. In our notation, a configuration for this classifier is written as "X.Y": 10.3, for example, defines the configuration in which X=10 and Y=3, and 4.2 defines X=4 and Y=2. Categorical levels may be given arbitrary names, such as A1 and A2 to describe two possible levels. This notation succintly names and describes a large set of configurations.

We note that many experimental design procedures exist to help one choose which configurations need to be defined in order to allow for an accurate analysis. First, parameters with numeric values are quantized from a continuous scale to a small subset of values. Second, an experimental design is chosen. For example, a *fully factorial* design [26] dictates that all (considered) values of one parameter be examined with respect to all values of all other parameters, resulting in the maximum possible number of configurations. Smaller designs include Box Benhkens [7] and central composite [39]. No matter the chosen design, the next two steps in our framework remain the same.

3.2 Execute Configurations

The next step in our framework is to execute each configuration on the task at hand, to produce a set of tuples (C_i, $f(C_i)$), where each tuple contains the effectiveness $f(C_i)$ of configuration C_i. How exactly this is performed varies by the task. In bug localization, for example, this step entails (for each configuration): preprocessing the source code and bug reports; building the index on the source code; running the queries (bug reports) against the index to retrieve a ranked list of results; and evaluating the results using some measure of effectiveness (such as top-20 for the task of bug localization, defined later in Section 4.1.5, or F-measure or Mean Average Precision for the task of traceability linking).

3.3 Analyze Performance of Configurations

Finally, we analyze the performance of the configurations. We have two different goals. First, we wish to determine the best and worst configurations; and second, we wish to determine which parameter values are most effective.

To determine the best configurations, we simply sort the configurations by their effectiveness measure $f(C)$ in ascending or descending order, depending on whether high or low values of $f(C)$ are desirable, respectively.

To determine which parameter values are statistically most effective, we use Tukey's Honestly Significant Difference (HSD) test [58] to compare the performance of each value of each parameter. The HSD test is a statistical test on the means of the results produced for each parameter value—holding one parameter constant, and letting all other parameters vary. For a given parameter (e.g., "number of topics"), the HSD test compares the mean of each possible value with the mean of every other possible value (e.g., "32" vs. "64" vs. "128"). Using the studentized range distribution [57], the HSD test determines whether the differences between the means exceed the expected standard error. The result of HSD is a set of statistically-equivalent groups of parameter values. If two parameter values belong to the same group, then the performances of the parameter values are not statistically different, and either may be used in place of the other. Note that a parameter value can belong to multiple groups, and group memberships are not transitive: if parameter value A and parameter value B belong to the same group, and parameter value B and parameter value C belong to the same group, value A and value C do not necessarily belong to the same group.

4 Case Study 1: Which Configuration Is Best?

The goal of this case study is to evaluate the space of bug localization classifier configurations: which data representations, preprocessing steps, and other IR model parameter values result in the best bug localization performance. We use the configuration framework presented in Section 3.

4.1 Case Study Design

In this section we outline the design of our case study: which classifiers we define, which software projects we test, our data collection technique, and the performance metric (i.e., criterion function) we use.

4.1.1 Defined Classifiers

We consider two families of classifiers: IR-based classifiers and entity metric-based classifiers. Tables 2 and 3 list all the parameters and their values in our classifier evaluation, for IR-based classifiers and entity metric-based classifiers, respectively. values; indeed, an infinite number of possibilities exist. Instead, we aim to choose realistic values that are representative of those used most often in the literature, while keeping a reasonable number of configurations. We now describe each parameter and its possible values in more detail.

IR-based Classifiers. We build IR-based classifiers based on the three popular IR models described in Section 2.1: VSM, LSI, and LDA. For each IR model, we must decide which bug report representation to use for the query, which source code entity representation to use to build the index, how to preprocess the bug report and source code representation, and the remaining parameter values for the particular IR model.

For source code entity representation, we consider six values. The first three are based on the text of the source code entity itself: the identifier names (i.e., variable and method names) only (B1); comments only (B2); and both identifiers and comments (B3). As proposed by Nguyen et al. [40], we also consider the past bug reports (PBR) related to a source code entity. To do so, we represent the source code entity as a collection of the text of all of its PBRs. The idea is that a new bug report might be more textually similar to a past bug report than to the identifier names or comments of an entity, giving the IR model a better chance for success. We consider two values: using all the PBRs of an entity (B4); and using just the 10 most recent (i.e., $min(10, |\text{PBR}|)$) PBRs of an entity (B5). (We never include bug reports that were created in the future, relative to the bug report under consideration as a query.) Finally, we consider all possible data for a entity: its identifier, comments, and all PBRs (B6).

For bug report (i.e., query) representation, we consider three values: the title of the bug report only (A1); the description of the bug report only (A2); and both the title and description of the bug report (A3). In this study, we do not consider the comments or other metadata related to the bug report, as this information is usually not available at the time of bug localization. We also do not perform any query expansion techniques besides the preprocessing steps described below.

We consider three common preprocessing steps (see Section 2.1.4): splitting identifiers using simple regular expressions; removing stop words; and stemming using the Porter stemming algorithm. (We provide online the exact code and stop word lists that we use for preprocessing [55].) We remove programming language keywords and punctuation. Since the application of each preprocessing step is binary (i.e., is performed or is not performed), and all three preprocessing steps can be applied independently, we test a total of 8 possible preprocessing techniques (C0–C7).

The VSM model has two parameters: term weighting and similarity score. For term weighting, we consider the tf-idf (D1) and sublinear tf-idf (D2) weighting schemes [31], as well as the more basic Boolean (D3) weighting scheme [31]. For similarity score, we consider both the cosine (E1) and overlap similarity (E2) scores [31].

The LSI model has three parameters: term weighting, similarity score, and number of topics. We consider the same three term weighting schemes as we do for the VSM model (F1–F3). We hold the similarity score constant at cosine (H1), since research has shown this to be the best similarity score for LSI [31]. Finally, we consider four values for the number of topics: 32, 64, 128, and 256 (G32–G256). Smaller values produce coarser-grained topics, while larger values produce finer-grained topics. There is currently no automatic methodology for choosing an optimal number of topics a priori, so we select values that cover the range of typical values [56].

The LDA model has five parameters: number of topics, a document-topic smoothing parameter, a topic-word smoothing parameter, number of sampling iterations, and similarity score. We consider four values for the number of topics: 32, 64, 128, and 256 (J32–J256), to be consistent with our choices for the LSI model. The LDA implementation that we use, called MALLET [33], automatically optimizes for the document-topic and topic-word smoothing parameters, so we do not manually set values for these parameters. We also do not manually specify the number of iterations, and instead let the model run until convergence. Finally, we consider the conditional probability score (N1), as it is most relevant for IR applications [61]. Conditional probability has the advantage that it does not require the bug reports to be included in the LDA model at run-time. In contrast, other similarity measures are impractical because they require LDA to be rerun (on the source code entities and the bug reports) every time a new bug report appears.

Entity Metric-based (EM-based) Classifiers. The past decade has been very active for research in the area of bug prediction [13]. Briefly, this research aims at measuring features of the source code, such as lines of code (LOC), past bug-proneness, change proneness, and logical coupling between classes, to predict which source code entities contain bugs.

Entity metric-based bug localization classifiers use the insight from the bug prediction literature that many entity metrics are highly correlated with an entity's bug count. To this end, EM-based classifiers first calculate one or more metrics on the source code entities. Then, the classifiers rank the source code entities based on the metrics. For example, a higher LOC metric indicates more bugs, so one EM-based classifier would sort the entities by their LOC.

We note that, unlike IR-based classifiers, the rankings of EM-based classifiers are not based on the given bug report, so the same ranked list will be created for every bug report. Still, we note (and the bug prediction literature confirms) that since bugs are highly concentrated in a small number of source code entities, this list is likely to be accurate for any given bug report.

The EM-based classifiers have only a single parameter: which entity metric is used to determine the bug-proneness of an entity. We consider four metrics: the lines of code (LOC) of an entity; the churn of an entity (i.e., number of LOC that were added, deleted, or changed since the previous version); the cumulative bug count of an entity (i.e., the number of bugs that have been associated with this entity in the past); the new bug count of an entity (i.e., the number of bugs only since the previous version). Previous research has shown that these metrics are good predictors of the bug-proneness of an entity, so we expect the metrics to have reasonable performance for bug localization.

Fully Factorial Design. To quantify the performance of all possible classifiers (given our considered parameters and their possible values) we use a fully factorial design of our case study [26]. In this design, we explore every possible combination of parameter values. In our case, a fully factorial design results in a total of 3,168 IR-based

TABLE 2
The IR family of classifiers that we study. We show the configuration parameters and the values that we consider for each of the three underlying IR models: VSM, LSI, and LDA.

Parameter	Value
Parameters common to all IR classifiers	
Bug report represention	A1 (Title only) A2 (Description only) A3 (Title+description)
Entity represention	B1 (Identifiers only) B2 (Comments only) B3 (Idents+comments) B4 (PBR-All) B5 (PBR-10 only) B6 (Idents+comments+PBR-All)
Preprocessing steps	C0 (None) C1 (Split only) C2 (Stop only) C3 (Stem only) C4 (Split+stop) C5 (Split+stem) C6 (Stop+stem) C7 (Spit+stop+stem)
Parameters for VSM only	
Term weight	D1 (tf-idf) D2 (Sublinear tf-idf) D3 (Boolean)
Similarity metric	E1 (Cosine) E2 (Overlap)
Parameters for LSI only	
Term weight	F1 (tf-idf) F2 (Sublinear tf-idf) F3 (Boolean)
Number of topics	G32 (32 topics) G64 (64 topics) G128(128 topics) G256 (256 topics)
Similarity metric	H1 (Cosine)
Parameters for LDA only	
Number of iterations	I1 (Until model convergence)
Number of topics	J32 (32 topics) J64 (64 topics) J128 (128 topics) J256 (256 topics)
α	K1 (Optimized based on K)
β	L1 (Optimized based on K)
Similarity metric	N1 (Conditional probability)

TABLE 3
The EM family of classifiers that we study. We show the configuration parameters and the values we consider.

Parameter	Value
Metric	M1 (Lines of code) M2 (Churn) M3 (New bug count) M4 (Cumulative bug count)

TABLE 4
Studied projects.

	Eclipse (JDT)	Jazz (All)	Mozilla (mailnews)
Domain	IDE	IDE	Web
Language	Java	Java	C/C++/Java
License	Open	Closed	Open
Years considered	2002–2009	2007–2008	2004–2006
Snapshots	16	8	10
Bugs (preprocessed)	3,898	2,818	1,368
Source code files	1,882–2,559	756–887	319–332
LOC (K)	232–506	133–168	173–193

classifiers (VSM: 864=(3 bug data)*(6 entity data)*(8 preprocessing)*(3 term weights)*(2 similarity); LDA: 576=(3 bug data)*(6 entity data)*(8 preprocessing)*(4 no. of topics); LSI: 1728=(3 bug data)*(6 entity data)*(8 preprocessing)*(3 term weights)*(4 no. of topics)) and 4 entity metric-based classifiers. Thus, we have 3,172 classifiers under test.

4.1.2 Studied Projects

We study three software projects: Eclipse JDT, IBM Jazz, and Mozilla mailnews (Table 4). Eclipse is a large, popular integrated development environment (IDE) written in Java [16]. Eclipse JDT is the subset of Eclipse that implements Java development tools. Jazz is a proprietary IDE developed by IBM [22]. Mozilla is an application suite concerned mostly with web browsing and email clients [38]. Written mostly in C++, Mozilla is one of the largest and most active open-source projects to date; due to its size and the requirements of our case study, we only consider the largest module of Mozilla, mailnews.

We choose these projects mainly for two reasons. First, these projects are large, active, real-world projects, which allow us to perform a realistic evaluation of the classifiers under test. Second, each project carefully maintains bug tracking databases and source code version control repositories, which allows us to build our ground-truth datasets to evaluate the classifiers. Table 5 gives example bug reports from each studied project.

4.1.3 Data Collection and Ground Truth

We begin by obtaining the raw bug data from the bug tracking database and the source code from the version control system (VCS) for each studied project.

4.1.3.1 Creating the Ground Truth: To create the ground truth data that we use to evaluate relevancy, we use an approach similar to Fischer et al. [18] to link resolved bug reports to the version control system that were changed to resolve the bug [54]. Our approach parses the commit log messages from the source code repository (e.g., CVS, SVN, Git), looking for messages such as “Fixed Bug #45433” or similar variations. If found, the algorithm establishes a link between all the source code entities in the commit transaction with the identified bug ID, if it exists in the bug database. The approach uses several heuristics that in concert find fairly accurate links [18], [54]. The result is a reliable set of links between bug reports and source code entities, which we use to evaluate the classifiers under test. We provide our ground truth dataset online [55].

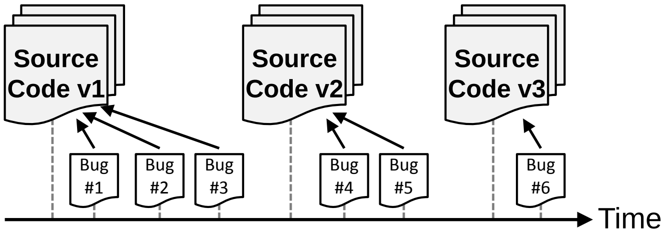

Fig. 1. Source code snapshots used in our evaluation procedure. First, we take snapshots of the project’s source code at six month intervals to build the classifiers. For each bug report under test, we use the classifiers built on the most recent (relative to the bug report) snapshot.

4.1.3.2 Source Code Preprocessing: We work at the file level of granularity. We preprocess the source code entities at each snapshot according to the specified classifier configuration (C0 to C7 in Table 2).

4.1.3.3 Bug Report Preprocessing: After collecting bug reports from the bug repositories, we preprocess them in the following manner. First, we remove bug reports that meet one or more of the following conditions.

- Bug report is not marked as “FIXED”
- Bug report does not result in the change of at least one entity. Some bugs, such as bug #6994 in Eclipse, deal with meta-source code issues, such as configuring an IDE correctly to build the project. Fixing these bugs results in no actual entity changes, and thus no links are created.
- Bug report has empty title field after pre-processing.
- Bug report links to build or configuration files (we are only interested in bugs linked to source code entities).

We preprocess the remaining bug reports according to the specified classifier configuration. For a given classifier, the same preprocessing steps are applied to the source code and bug reports.

4.1.4 Evaluation Procedure

Ideally, for each bug report in our testing set, we would take a snapshot of the source code at the exact time the bug report was created, build all classifiers using data from that snapshot, and perform the classification. This process would provide the most accurate and realistic evaluation scenario, because it uses the source code that the classifiers would have available in a real-world situation. However, doing so would be too computationally expensive for our evaluation, as we are evaluating thousands of bugs and thousands of classifiers, each taking upwards of several minutes to build using our unoptimized research prototypes. As a compromise, we first take snapshots of each project’s source code from the studied projects’ VCS at six month intervals over the

TABLE 5
Example bug reports in the three studied projects, and their relevant source code entities.

Project	Bug ID	Title	Relevant entity(s) from ground truth
Eclipse	102645	"[open type] Open Type history shows stale visibility info in type history"	`util.TypeInfo.java`
	106638	"[misc] Support BiDi chs in logical expr in java editor"	`JavaSourceViewer.java`
	100062	"[formatting] Code formatter is broken on test case from bug 99999"	`CodeFormatterVisitor.java`
	100302	"StackOverflowError during completion"	`CompletionParser.java`
Jazz	28284	"Create button enabled for process iterations while editor is loading"	`processpart.java`
	18317	"Move team area dialog should not allow to show archived areas."	`teamareamovewizard.java`
	21247	"Offer to switch to edit mode when I start typing in the over page"	`wikiformpage2.java`
	30963	"Error messages in bluesdev server log for I20070912-2000"	`auditableserver.java`
Mozilla	105964	"Drop 20 bytes off each imgRequest object"	`imgRequest.cpp`
	24668	"[DOGFOOD]Crash when clicking on Finish on Account Setup"	`xpcwrappedjsclass.cpp`
	11001	"[4.xP] Table spacing borders incorrect at http://www.choochem.com"	`nsElementTable.cpp`
	222023	"regression: pref parser should accept single-quote delimited strings"	`prefread.cpp`

duration of the project and precompute the classifiers at each snapshot (Figure 1). Given a bug report during evaluation, we determine the most recent past snapshot of the source code and use the corresponding precomputed classifiers. This process allows us to consider temporally appropriate source code for each bug report without requiring substantial computation.

For each classifier under test, we perform the following procedure. For every bug report in each studied project, we determine the nearest snapshot occuring in the past, and use the classifier to obtain the ranked list of source code entities. Using the ground truth, we determine the rank of the first relevant entity on the list.

4.1.5 Performance Metrics

To measure the performance of a classifier, we use the top-k accuracy metric, as others have [40]. The ***top-k accuracy*** metric measures the percentage of bug reports in which at least one relevant source code entity was returned in the top k results. Formally,

$$\text{top-}k(C_j) = \frac{1}{|Q|} \sum_{i=1}^{|Q|} I\big(\exists d \in D \mid \text{rel}(d, q_i) \wedge r(d \mid C_j, q_i) \leq k\big),$$

where $|Q|$ is the number of queries (i.e., bug reports), q_i is an individual query, rel(d, q_i) returns whether entity d is relevant (using truth information) to query q_i, $r(d|C_j, q_i)$ is the rank of d given by C_j in relation to q_i, and I is the indicator function, which returns 1 if its argument is true and 0 otherwise. For example, a top-20 accuracy value of 0.25 indicates that for 25% of the bug reports, at least one relevant source code entity was returned in the top 20 results. Previous work has set k to 20, with the idea that 20 is a reasonable number of entities for a developer to search through before growing impatient and resorting to other means of bug localization [40].

Other metrics which are commonly used for evaluating IR models, such as precision, recall, and Mean Average Precision (MAP), are inappropriate for our purposes. In bug localization, the task is to locate the *first* relevant entity for a given bug report, and therefore there is at most one correct answer in the list of returned entities; precision, recall, and MAP metrics are appropriate only when we expect to find several possible correct answers for each query.

4.2 Results

Table 6 shows the best and worst four configurations of each of the four classification families: three IR classification techniques (VSM, LSI, and LDA) and one EM classification technique, for each of the three studied projects, ordered by the top-20 metric. (We provide the full set of results for all configurations online [55].) For all three studied projects, the VSM classification technique achieves the best overall performance, consistent with previous findings [49]. The best configuration of VSM varies slightly from project to project, and many configurations have comparable performance.

Table 7 shows the dispersion of the performance of the various configurations: the worst, average, and best configuration performances for each of the four classification techniques. We find that *configuration matters*: for all studied projects and all classification techniques, the differences between the worst configuration and best configuration is significant. The differences between the "average" configuration and the best configuration is also significant: for example, in Eclipse, using the VSM classification technique, the median top-20 performance is 21% (configuration VSM.A2.B2.C4.D1.E2), while the best performance is 55% (configuration VSM.A3.B6.C7.D1.E1). This suggests that the choice of bug report representation, entity representation, preprocessing steps, and IR model parameters has a large effect on the overall performance of a classifier, no matter the underlying classification technique.

We now present the results of Tukey's HSD statistical test on each parameter listed in Table 2. We note that all of our results meets the homoscedasticity condition required by Tukey's HSD test, as determined by the `bartlett.test()` method in R.

4.2.1 Bug Report Representation

Table 8 shows the HSD results for the bug report representation parameter. (We provide the HSD results for

TABLE 6
The best four and worst four configurations, for each classifier family (VSM, LSI, LDA, and EM) and each studied project. The configurations are ordered according to their top-20 performance.

VSM			LSI			LDA			EM		
Rank	Configuration	Top-20	Rank	Configuration	Top-20	Rank	Configuration	Top-20	Rank	Configuration	Top-20
Eclipse											
1	VSM.A3.B6.C7.D1.E1	0.548	1	LSI.A3.B6.C4.F2.G256	0.462	1	LDA.A1.B4.C7.J128	0.290	1	EM.M3	0.405
2	VSM.A3.B6.C4.D1.E1	0.535	2	LSI.A3.B6.C7.F2.G256	0.444	2	LDA.A1.B4.C6.J128	0.290	2	EM.M4	0.323
3	VSM.A3.B3.C7.D1.E1	0.524	3	LSI.A3.B6.C1.F2.G256	0.438	3	LDA.A1.B5.C7.J128	0.282	3	EM.M1	0.226
4	VSM.A3.B6.C5.D1.E1	0.512	4	LSI.A3.B6.C2.F2.G256	0.434	4	LDA.A1.B5.C6.J128	0.282	4	EM.M2	0.172
861	VSM.A2.B3.C0.D3.E1	0.012	1725	LSI.A2.B6.C0.F1.G32	0.065	573	LDA.A2.B3.C3.J64	0.009		–	
862	VSM.A2.B3.C0.D3.E2	0.012	1726	LSI.A2.B3.C3.F1.G32	0.061	574	LDA.A3.B6.C0.J64	0.009		–	
863	VSM.A2.B3.C1.D3.E2	0.011	1727	LSI.A3.B3.C0.F1.G32	0.056	575	LDA.A3.B3.C0.J64	0.008		–	
864	VSM.A2.B3.C1.D3.E1	0.011	1728	LSI.A2.B3.C0.F1.G32	0.044	576	LDA.A3.B6.C0.J32	0.007		–	
Jazz											
1	VSM.A3.B6.C6.D1.E1	0.686	1	LSI.A3.B6.C2.F2.G256	0.588	1	LDA.A3.B1.C7.J256	0.336	1	EM.M3	0.330
2	VSM.A3.B6.C7.D1.E1	0.673	2	LSI.A3.B6.C6.F2.G256	0.584	2	LDA.A3.B3.C7.J256	0.331	2	EM.M4	0.301
3	VSM.A3.B6.C2.D1.E1	0.669	3	LSI.A3.B6.C6.F2.G128	0.576	3	LDA.A3.B6.C7.J256	0.330	3	EM.M1	0.181
4	VSM.A3.B6.C4.D1.E1	0.657	4	LSI.A3.B6.C2.F2.G128	0.574	4	LDA.A1.B6.C7.J256	0.318	4	EM.M2	0.161
861	VSM.A2.B6.C1.D3.E2	0.069	1725	LSI.A2.B3.C0.F2.G32	0.144	573	LDA.A2.B3.C0.J64	0.066		–	
862	VSM.A2.B6.C1.D3.E1	0.068	1726	LSI.A2.B3.C0.F1.G64	0.139	574	LDA.A2.B3.C0.J256	0.065		–	
863	VSM.A2.B6.C0.D3.E2	0.068	1727	LSI.A2.B3.C3.F1.G32	0.121	575	LDA.A2.B3.C3.J32	0.057		–	
864	VSM.A2.B6.C0.D3.E1	0.068	1728	LSI.A2.B3.C0.F1.G32	0.098	576	LDA.A2.B3.C0.J32	0.051		–	
Mozilla											
1	VSM.A3.B6.C6.D1.E1	0.802	1	LSI.A3.B3.C2.F2.G128	0.794	1	LDA.A3.B6.C7.J128	0.523	1	EM.M3	0.667
2	VSM.A3.B6.C7.D1.E1	0.796	2	LSI.A3.B3.C6.F2.G128	0.782	2	LDA.A3.B6.C7.J256	0.522	2	EM.M4	0.576
3	VSM.A3.B6.C4.D1.E1	0.794	3	LSI.A3.B3.C2.F2.G256	0.781	3	LDA.A3.B6.C4.J256	0.517	3	EM.M1	0.528
4	VSM.A3.B6.C5.D1.E1	0.788	4	LSI.A3.B6.C2.F2.G128	0.778	4	LDA.A3.B6.C7.J64	0.505	4	EM.M2	0.383
861	VSM.A2.B2.C1.D3.E2	0.067	1725	LSI.A2.B4.C1.F3.G32	0.242	573	LDA.A1.B5.C4.J256	0.058		–	
862	VSM.A2.B2.C1.D3.E1	0.066	1726	LSI.A2.B4.C0.F3.G32	0.240	574	LDA.A1.B5.C2.J256	0.058		–	
863	VSM.A2.B2.C0.D3.E2	0.062	1727	LSI.A2.B4.C5.F3.G32	0.228	575	LDA.A1.B5.C7.J256	0.000		–	
864	VSM.A2.B2.C0.D3.E1	0.061	1728	LSI.A2.B4.C3.F3.G32	0.225	576	LDA.A1.B5.C6.J256	0.000		–	

TABLE 8
The results of Tukey's HSD test for the bug report representation parameter, for each IR classification technique and each studied project. If two values appear in the same group, then their top-20 performances were not statistically different.

VSM			LSI			LDA		
Group	Mean	Parameter value	Group	Mean	Parameter value	Group	Mean	Parameter value
Eclipse								
a	0.238	A1 (title)	a	0.259	A3 (title+descr.)	a	0.131	A1 (title)
a	0.215	A3 (title+descr.)	b	0.227	A1 (title)	b	0.070	A3 (title+descr.)
b	0.180	A2 (descr.)	b	0.217	A2 (descr.)	b	0.062	A2 (descr.)
Jazz								
a	0.354	A1 (title)	a	0.399	A3 (title+descr.)	a	0.214	A1 (title)
a	0.351	A3 (title+descr.)	b	0.362	A1 (title)	a	0.206	A3 (title+descr.)
b	0.270	A2 (descr.)	c	0.314	A2 (descr.)	b	0.158	A2 (descr.)
Mozilla								
a	0.458	A1 (title)	a	0.555	A3 (title+descr.)	a	0.346	A1 (title)
ab	0.439	A3 (title+descr.)	a	0.542	A1 (title)	a	0.325	A3 (title+descr.)
b	0.402	A2 (descr.)	b	0.515	A2 (descr.)	b	0.282	A2 (descr.)

TABLE 7
Performance dispersion amongst classifier configurations, using the top-20 performance metric.

	Configs.	Min.	1st Qu.	Med.	Mean	3rd Qu.	Max.
Eclipse							
VSM	864	0.011	0.083	0.209	0.211	0.321	0.548
LSI	1728	0.044	0.175	0.228	0.234	0.287	0.462
LDA	576	0.007	0.053	0.074	0.088	0.101	0.290
EM	4	0.172	0.213	0.275	0.281	0.343	0.405
Jazz							
VSM	864	0.068	0.214	0.322	0.325	0.430	0.686
LSI	1728	0.098	0.306	0.354	0.358	0.403	0.588
LDA	576	0.051	0.149	0.195	0.192	0.238	0.336
EM	4	0.161	0.176	0.241	0.243	0.308	0.330
Mozilla							
VSM	864	0.061	0.271	0.441	0.433	0.604	0.802
LSI	1728	0.225	0.441	0.536	0.537	0.647	0.794
LDA	576	0.000	0.228	0.322	0.318	0.418	0.523
EM	4	0.383	0.492	0.552	0.538	0.599	0.667

the remaining parameters online [55].) A2 (description) always i.e., for the three IR models and the three studied projects, belongs to the bottom group and never belongs to the top group, meaning that A2 is always significantly worse than at least one other value. A3 (title and description) almost always belongs to the top group: in one instance (LDA model, Eclipse), A3 is in the second group. A1 (title) is similar in that it is usually in the top group (2 exceptions: LSI model, Eclipse; and LSI model, Jazz). Since both A1 and A3 include the title of the bug report, and A2 does not, we theorize that including the title from the bug report representation is most important; whether you also include the description is of secondary importance. This result is likely because the bug descriptions introduce noise into the IR models, since the descriptions often include stack traces. Stack traces have the names of several files and methods, which can lead the IR model in the wrong direction. In contrast, the title is typically carefully constructed to exactly summarize the problem.

4.2.2 *Source Code Entity Representations*

The best choice of source code entity representation differs between the studied projects. In Mozilla and Jazz, B6 (all available data: identifiers, comments, and past bug reports) belongs to the top group no matter the IR model. Therefore, in these studied projects, more information is better. In addition, B4 (all past bug reports) and B5 (last 10 bug reports) do not belong to the top group in any of the IR models, indicating that identifiers and comments play a more important role in these studied projects.

In Eclipse, on the other hand, B4 and B5 belong to the top group for all three IR models. B6 appears in the top group for VSM, and the second of two groups for LDA, and the second of four groups for LSI. Therefore, in Eclipse, including past bug reports is most important; including identifiers and/or comments is of secondary importance.

One possible explanation for the difference between studied projects is the number of snapshots in our analysis. Our Eclipse dataset contains 16 snapshots, compared to the 8 and 10 of Jazz and Mozilla, respectively. Given the nature of our evaluation procedure, any configuration using only PBRs will achieve low performance during the first snapshot of the project, since by definition the source code entities have not yet been linked to any past bug reports. The IR models are built on empty representations of the entities, and bug localization is simply random. While all three studied projects share this characteristic, it may be the case that the additional snapshots in Eclipse help to mitigate the effects of the poor performance of the first snapshot, since the top-20 metric is averaged across all snapshots. Further investigation is needed to verify this possible explanation.

Overall, we conclude that B6 (all) is the best: For Mozilla and Jazz, B6 is always in the top group; in Eclipse, B6 is always in one of the top 2 groups. Note that B6 is suited to cases when there are no past bug reports to which to link the entities, since identifiers and comments are also included.

4.2.3 *Preprocessing Steps*

C7 (stopping, stemming, and splitting) is in the top group for all studied projects and all IR models; it is the only parameter value that is. C4 (split and stop) is almost always in the top group (one exception: LDA applied to Jazz). C6 (stop and stem) is almost always in the top group (two exceptions). C2 (stop) is in the top group in all but four cases. C1 (split) is almost never in the top group; it is only twice. C3 (stem) is in the top group only once. Finally, C0 (none) is always in the bottom group, never in the top group. We conclude that performing all three preprocessing steps is most beneficial, and removing stop words is the most important of the three steps. Removing stop words helps to reduce the size of the vocabulary and thus concentrates the information content of each document. Splitting and stemming do the same, yet are slightly less effective for source code.

4.2.4 *IR Model Parameters*

4.2.4.1 VSM parameters: For VSM, tf-idf (D1) weighting is always (i.e., for each studied project and each IR model) best, and sublinear tfi-df (D2) is always second best, and boolean (D3) weighting is always last. These results are consistent with research in other domains [31], as tf-idf is the most popular weighting scheme. Cosine similarity (E1) is always better than overlap similarity (E2).

4.2.4.2 LSI parameters: For LSI, sublinear tf-idf (F2) is always best, tf-idf (F1) is always best or second best, and Boolean (F3) is always last.

The best number of topics depends on the studied project. In Mozilla, 64 and 128 topics are best. In Eclipse, 256 topics is best. In Jazz, 128 and 256 topics are best.

These are roughly proportional to the number of lines of code in each studied project, suggesting that the number of topics should be selected based on the size of the studied project.

4.2.4.3 LDA parameters: For LDA, it appears that more topics are better, but there are no significant differences in any studied project between 64, 128, and 256 topics. Thirty-two topics, however, is always significantly worst.

4.2.5 Entity Metrics

The HSD statistical test is not appropriate for the EM-based classifiers, since there are only four configurations total. Instead, we use Table 6 to discuss the results.

In all three studied projects, the new bug count metric (EM.M3) has the best performance. This is consistent with previous research [48]. In fact, in all three studied projects, the EM.M3 classifier has comparable or better performance than the best LDA configuration, even though EM.M3 does not make use of the textual data of the bug report or source code in any way. This somewhat surprising result agrees with current research in defect prediction, which has repeatedly shown that prior bug counts are good predictors of future bugs [36]. Coupled with the observation that the majority of bugs in any given project are found in only a small subset of the entities [42], [43], it is understandable that for most new bug reports, the relevant entities will have a history of bug-proneness. The second best performing EM-based classifier also incorporates bug information: EM.M4 is the cumulative bug count metric. The other two entity metrics, EM.M1 and EM.M2 perform worse in each studied project.

From these results, we make two conclusions. First, some EM-based classifiers can perform at least as well as some IR-based classifiers. Second, entity metrics based on previous bug counts are better than those based on LOC or churn metrics.

> *The best individual IR-based classifier uses the Vector Space Model, with the index built using tf-idf term weighting on all available data in the source code entities (i.e., identifiers, comments, and past bug reports for each entity), which has been stopped, stemmed, and split, and queried with all available data in the bug report (i.e., title and description) with cosine similarity.*

5 Combination Framework

There is a rich literature in the pattern recognition and data mining domains for combining classifiers, also known as classifier ensembles, voting experts, or hybrid methods [21], [24], [35]. No matter the name used, the fundamental idea is the same: individual classifiers often excel on different inputs and make different mistakes. For example, one IR-based classifier might be good at finding links between bug reports and source code entities if they share one or more exact-match keywords, which might happen if the bug report explicitly mentions the variable names or method names of the relevant source code entity. Another IR-based classifier might be good at finding links between general concepts, such as "GUI" or "networking", even if no individual keywords are shared between the bug report and relevant entity. An entity metrics-based classifier might be good at determining which source code entities are likely to be bug-prone in the first place, regardless of the specific bug report in question. By combining these disparate classifiers, the truly-relevant files are likely to "bubble up" to the top of the combined list, providing developers with fewer false positive matches to investigate. While classifier combination has been successful in other domains and recently even other areas of software engineering [25], classifier combination has not been investigated in the context of bug localization.

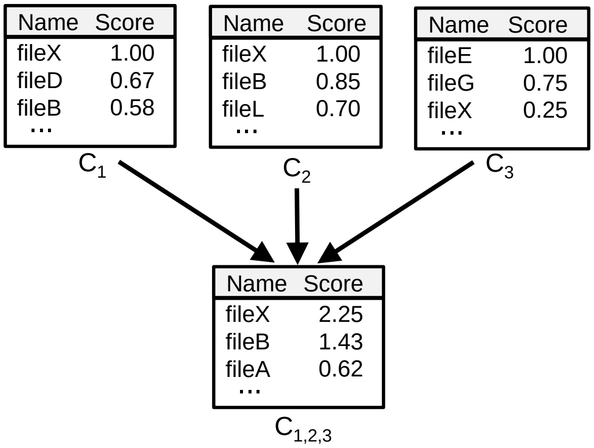

Fig. 2. An illustration of the classifier combination framework. Here, three classifiers are created, based on the available input data and the given bug report. Then, the classifiers are combined, using score addition, to produce a single ranked list of source code entities. In this example, fileX bubbles up to the top of the combined list since it is high on each of the three classifiers' lists.

We present a framework to combine multiple bug localization classifiers, based largely on methods from other domains and illustrated in Figure 2. The framework consists of two main constituents.

1) Any number of classifiers are created, based on the available input data and the given bug report. Exactly which classifiers are used can have an effect on the performance of combination; we discuss this issue more in Section 5.2.
2) The classifiers are combined using any of several combination techniques, such as Borda Count [59], score addition, or reciprocal rank fusion [11].

We now describe each constituent in more detail.

5.1 Creation of Classifiers

As mentioned in Section 3, classifiers can come in many forms. IR-based classifiers attempt to find textual similarities between the given bug report and the source code entities. Entity metric-based (EM-based) use entity metrics [62], such as lines of code, to classify which source code entities are likely to have the largest number of bugs, independent of the given bug report (which, as we found in Section 4.2, has surprisingly good performance). In fact, we consider any variant of any technique that returns a ranked list of source code entities as a classifier. Formally, we define the *result set* of a classifier C_i, which operates on a given bug report q_j, as

$$C_i(q_j) = \Big\{ \Big(r(d_k), s(d_k)\Big) \quad \forall d_k \in D \Big\}$$

where $r(d_k)$ is the rank of entity d_k, $s(d_k)$ is the score of entity d_k, according to C_i, and D is the set of all entities in the project. The result set of a classifier consists of a rank and a (relevancy) score for every source code entity in the project. Note that scores need not be unique; in fact, many entities may be assigned a score of 0. In this case, they all share a rank of $M + 1$, where M is the number of entities that received a non-zero score.

It is well established that the choice of classifiers that are used will effect the performance of the combination [21]. As classifier combination works best when the individual classifiers err in different ways, choosing classifiers that are likely to result in the most uncorrelated mistakes—such as classifiers based on different input data representations—is likely to achieve the best result.

Defining Classifier Subspaces. Recall that for IR-based classifiers, two data sources must be represented: that of the source code entity, and that of the bug report. Accordingly, we define four IR *classifier subspaces* that are based on four different input data representations, using the notation from the configuration framework (Section 3), as follows. The first subspace consists of those classifiers that use the entities' textual content (B3: identifiers and comments) and the bug reports' titles (A1). The classifiers in the second subspace use the entities' textual content (B3) and the bug reports' descriptions (A2). The classifiers in the third subspace use the entities' past bug reports (B5) and the bug reports' titles (A1), the bug reports' descriptions (A2).

Given that there are thousands of individual classifiers in the context of bug localization, and each classifier is computationally expensive to create, it is in our interest to choose a subset of classifiers in each subspace to create and combine. We describe two basic techniques for choosing which classifiers to combine: combining the best performing individual classifiers in each subspace, and combining random sets of classifiers from each subspace.

Combining the Best Classifiers. In order to combine the best IR classifiers, then, we select the best classifier from each of the four subspaces. Should we want to combine the best eight IR classifiers, for example, then we select the best two classifiers from each subspace, and so on.

Combining Random Classifiers. If we do not know in advance which classifiers from each subspace perform best, we still use the subspaces to create so-called *intelligently random* sets of classifiers. Here, we select random classifiers with an equal probability from each of the four subspaces, so that we decrease the likelihood that the wrong answers from each classifier will be correlated in any way.

5.2 Combination Techniques

Given a set of $|C|$ classifiers, we can combine them in any of several ways. A simple rank-only combination is the Borda Count method [59], which was originally devised for political election systems. For each source code entity d_k, the Borda Count method assigns points based on the rank of d_k in each classifier's result set. For example, if classifier C_i assigned d_k a rank of 1, then the Borda Count for d_k would be M-1, where M is the number of entities that received a non-zero score in C_i. The entity as rank 2 would receive a score of M-2, and so on. The Borda Count scores for all $|C|$ classifiers are tallied for each entity, and the entity with the highest total Borda Count score is ranked first, and so on. Formally, the score for an entity d_k is defined as

$$\text{Borda}(d_k) = \sum_{C_i \in C} M_i - r(d_k \mid C_i), \tag{1}$$

where M_i is the number of entities that received a non-zero score in C_i, and $r(d_k \mid C_i)$ is the rank of entity d_k in C_i. The entities are then ranked according to their total Borda score.

Instead of using the ranks, the scores of each classifier can also be used. For example, the score of each entity d_k and each classifier C_i is summed to produce a total score for each entity:

$$\text{ScoreAddition}(d_k) = \sum_{C_i \in C} s(d_k \mid C_i). \tag{2}$$

Usually, the scores of each classifier are scaled to be in the same range (e.g., [0,1]) before combination to avoid unintentionally weighting the importance of certain classifiers. However, Equations (1) and (2) can be modified to explicitly weight certain classifiers differently from others, for example to scale the best-performing classifiers in the range of [0,2]. We leave the investigation of such weighting schemes for future work.

We note that the result of combining $|C|$ classifiers defines a new classifier. This classifier can itself be combined with other classifiers. In this way, a hierarchy of classifiers can be constructed.

6 Case Study 2: Does Combination Help?

This case study investigates the performance improvements that are achieved by combining classifiers, using

TABLE 9
Classifier sets under consideration.

	$\lvert C\rvert$	C = Classifiers
CS1	5	VSM.A1.B3.C7.D1.E1, VSM.A2.B3.C7.D1.E1, VSM.A1.B5.C7.D1.E1, VSM.A2.B5.C7.D1.E1, EM.M3
CS2	5	LDA.A1.B3.C7.J256, LDA.A2.B3.C7.J256, LDA.A1.B5.C7.J256, LDA.A2.B5.C7.J256, EM.M3
CS3	5	LSI.A1.B3.C7.F2.G128, LSI.A2.B3.C7.F2.G128, LSI.A1.B5.C7.F2.G128, LSI.A2.B5.C7.F2.G128, EM.M3
CS4	13	CS1 ∪ CS2 ∪ CS3

the combination framework introduced in Section 5. We present two experiments: one to test the performance of combining the best-performing individual classifiers, and another to test the performance of combining "intelligently-random" sets of classifiers. In both experiments, we use the same projects, data collection algorithm, performance metrics, and evaluation technique as case study 1 (Section 4).

6.1 Experiment 1: Combining the Best Individual Classifiers

In this experiment, we investigate whether combining the best-performing classifier configurations can improve performance.

6.1.1 Experiment Design

We define four classifiers sets, CS1–CS4, shown in Table 9. CS1 contains the best VSM classifier from each of the four subspaces identified in Section 5.1. For each subspace, we use the results of case study 1 (Section 4) to determine the best classifier in the subspace. CS1 also includes the best-performing EM classifier, EM.M3.

We define similar sets based on LDA (CS2 in Table 9) and LSI (CS3). Finally, we take the union of the sets in CS1–CS3 to create a new set, CS4, with 13 classifiers.

The classifier sets CS1–CS3 each contain five classifiers that (a) operate on independent data representations (e.g., the identifiers and comments will be very different from the past bug reports) and (b) have optimal values for the other parameters. Thus, we expect that combining these classifiers will increase overall performance.

For each of the classifier sets defined above, we consider two combination techniques: the Borda Count method (BRD) and the score addition method (ADD), each described previously in Section 5.2. We run each classifier set on all 8,084 bugs reports in the three studied projects, and calculate the top-20 performance of each classifier set.

6.1.2 Results

Table 10 shows the top-20 performance of the four classifier sets, as well as their relative performance improvements over the best individual classifier in the sets, for both the Borda Count and score addition methods. The relative improvement of classifier set CS is calculated as

$$\mathrm{RI}(CS) = \frac{\text{top-20}(CS) - \max_{C_i \in CS}\big(\text{top-20}(C_i)\big)}{\max_{C_i \in CS}\big(\text{top-20}(C_i)\big)}.$$

In all three studied projects, classifier combination improves performance for all classifier sets, often significantly. In Jazz, for example, classifier set CS2, which combines four LDA configurations and EM.M3, results in a 95% relative improvement over the best individual classifier, going from a top-20 performance of 33% to a top-20 performance of 64%. We find similar results for each of the other studied projects, classifier sets, and combination techniques, indicating that combining the best performing classifiers can improve bug localization performance.

6.2 Experiment 2: Combining Random Classifier Sets

We conduct a second experiment to investigate whether classifier combination helps in situations where the best configuration of each classifier is not known in advance.

6.2.1 Experiment Design

Similar to Experiment 1, we define classifier sets with five classifiers: one from each of the four IR subspaces, and one based on entity metrics. However, in this experiment, we do not use the best classifier from each of the four IR subspaces. Instead, we choose one at random.

Specifically, we build a random classifier set with the following classifiers (using regular expression notation): VSM.A1.B3.*.*.*, VSM.A2.B3.*.*.*, VSM.A1.B5.*.*.*, VSM.A1.B5.*.*.*, and EM.*. We build 50 such sets, each time randomly choosing values for the varying parameters (i.e., "*"). We do the same for LDA and LSI, yielding a total of 150 randomly-generated classifier sets. (We enumerate the sets online [55].) By randomly choosing classifiers from each subset, we can examine the effects of many situations: combining good classifiers with bad classifiers; bad with bad; and so on.

For each classifier set, we again consider two combination techniques: the Borda Count method (BRD) and the score addition method (ADD). We run each classifier set on all 8,084 bugs reports in the three studied projects, and calculate the top-20 performance metric.

6.2.2 Results

Table 11 shows the percentage of the 150 classifier sets that were improved by combination, in terms of top-20 performance. Overall, classifier combination helps in the large majority of sets. In Eclipse, 93% or 84% of the classifier sets had a better performance after combination, depending on whether the Borda Count or score addition methods were used, respectively. In Mozilla, the results were even better: 88% or 96% of the classifier sets improved after combination. The best results came

TABLE 10
Top-20 performance of the four manually-created classifier sets, CS1–CS4 (see Table 9), and their relative improvements over the best individual classifier in the sets.

	Performance of best individual classifier	Performance of combined classifiers			
		Borda count	Relative improvement (%)	Score addition	Relative improvement (%)
Eclipse					
CS1: 4 VSM + Best EM	0.467	0.706	+51.2	0.666	+42.7
CS2: 4 LDA + Best EM	0.405	0.522	+28.9	0.464	+14.8
CS3: 4 LSI + Best EM	0.405	0.708	+75.1	0.661	+63.3
CS4: 4 VSM + 4 LDA + 4 LSI + Best EM	0.467	0.699	+49.5	0.640	+37.0
Jazz					
CS1: 4 VSM + Best EM	0.576	0.739	+28.4	0.737	+28.1
CS2: 4 LDA + Best EM	0.330	0.643	+95.0	0.568	+72.2
CS3: 4 LSI + Best EM	0.438	0.732	+67.3	0.721	+64.7
CS4: 4 VSM + 4 LDA + 4 LSI + Best EM	0.576	0.697	+21.1	0.694	+20.7
Mozilla					
CS1: 4 VSM + Best EM	0.739	0.844	+14.2	0.837	+13.3
CS2: 4 LDA + Best EM	0.667	0.787	+18.0	0.717	+7.6
CS3: 4 LSI + Best EM	0.703	0.836	+18.9	0.820	+16.6
CS4: 4 VSM + 4 LDA + 4 LSI + Best EM	0.739	0.815	+10.3	0.818	+10.7

in Jazz, where 97% and 100% of classifiers sets were improved.

Table 11 also quantifies the amount of improvement the classifier sets experienced after combination. The median relative improvements for Eclipse were 39% and 37% for the Borda Count and score addition methods, respectively. This means that for a random combination of classifiers, one can expect performance to improve by at least 37%. The same is true for the other two studied projects: in Jazz, the mean relative improvements were 57% and 54%, and in Mozilla they were 14% and 18%.

> *Classifier combination—based on the four subspaces of classifiers—helps in almost all cases, no matter the underlying classifiers used, or the specific combination technique used.*

7 Main Findings and Discussion

We highlight the main findings from each case study, and provide a discussion of results.

7.1 Classifier Configuration

From our analysis of 3,172 classifiers (Section 4), each evaluated on 8,084 bug reports, we make the following conclusions.

- Configuration matters: the performance difference between one classifier configuration and another is often significant.
- The VSM classifier can achieve the best overall top-20 performance. LSI is second, and LDA is last.
- Using both the bug report's title and description results in the best overall performance, for all IR models.
- Using the source code entities' identifiers, comments, and past bug reports results in the best overall performance, for all IR models.
- Stopping, stemming, and splitting all improve performance, for all IR models.
- New bug count is the best-performing entity metric.

No single classifier is best for all three studied projects. However, a configuration of the VSM classifier has the best performance for all studied projects, suggesting that VSM is the overall best classification technique for bug localization. For all studied projects, VSM > LSI > LDA, when considering the performance of each technique's best configuration.

Interestingly, for all studied projects, if we consider the worst configurations of the various IR models (VSM, LSI, and LDA), LSI achieves the highest performance, often significantly so. In Mozilla, for example, the worst LSI configuration has a performance of 23%, compared to VSM's 6% and LDA's 0%.

Bug localization in Mozilla is relatively easy compared to the other studied projects: the best top-20 for Mozilla is 80%, compared to Jazz's 69% and Eclipse's 55%. We also note that Mozilla has the smallest number of LOC, Jazz the second smallest, and Eclipse the largest.

In general, we find that more is better: take all the bug report data you can (A3); take all the entity data you can (B6); and do all the preprocessing steps you can (C7). Each of these parameter values had the best overall performance across the various IR models and studied projects.

We stress that the configuration of a classifier has a significant impact on its results. The difference between the best and worst configuration is large; even small differences (e.g., deciding not to remove stop words) can result in large performance variations. Researchers

TABLE 11
Improvement of classifier combination in 150 randomly-generated classifier sets. We show the percentage of classifier sets in which the performance of the combination was better than the performance of the best individual classifier in the set. We also report summary statistics of the relative improvement that combination provides.

		% of sets improved	Summary of relative improvements provided by combination					
		by combination	Min	1st Qu.	Med.	Mean	3rd Qu.	Max.
Eclipse	Borda Count	92.7	-20.0	17.7	39.4	37.2	55.1	121.0
	Score addition	84.0	-33.9	14.7	36.9	35.1	55.9	95.0
Jazz	Borda Count	97.3	-16.3	37.6	56.8	56.1	72.8	142.2
	Score addition	100.0	0.3	43.0	54.4	54.3	65.7	114.3
Mozilla	Borda Count	88.0	-37.7	7.3	14.4	13.9	22.2	53.7
	Score addition	96.0	-6.1	12.5	17.8	19.2	24.3	56.5

and practitioners should be careful when configuring their bug localization classifiers. Overall, we recommend the VSM.A3.B6.C7.D1.E1 classifier (VSM using the bug report title and description, the source code entities' identifiers, comments, and past bug reports, preprocessed by splitting, stopping, and stemming, using tf-idf weighting, and using the cosine similarity metric), for two reasons. First, it achieves the best performance in Eclipse and the second best performance in Jazz and Mozilla. In addition, all the configuration settings (i.e., A3, B6, C7, D1, and E1) were shown to be optimal by Tukey's HSD statistical test.

7.2 Classifier Combination

Based on the results of our case study on classifier combination, we conclude that:

- Combining the best-performing classifiers always improves performance, by at least 10% and up to 95% in top-20 performance, compared to the best performing individual classifier.
- Combining "intelligently-random" sets of classifiers almost always helps (84–100% of the time), and usually helps by a large amount: a median of 14–56% improvement in top-20 performance, compared to the best performing individual classifier..
- The Borda Count combination method is always comparable or better than the score addition method.

These results provide strong evidence that classifier combination is a valuable method for improving bug localization performance.

We saw the smallest relative improvements in Mozilla, and the largest in Eclipse. We note that individual classifiers in Mozilla already have high performance (i.e., top-20 values above .80), leaving little room for improvement for combination. Individual classifiers in Eclipse, on the other hand, have relatively worse performance (a maximum top-20 value of 0.54).

In general, the Borda Count combination method performed better than the score addition method. In all four manually-created classifier sets, and all studied projects (Table 10), the Borda Count offered a greater improvement than score addition (with one exception: the Borda Count method in CS4 in Mozilla had a relative improvement of 10.3%, compared to score addition's 10.7%). In addition, when considering the 150 randomly-created classifier sets, the Borda Count method offered better mean and median relative improvements over score addition, for the Eclipse and Jazz projects (Table 11). In Mozilla, the mean and median relative improvements of Borda Count and score addition were comparable.

In both experiments, we combined sets of five component classifiers, based on the logic that classifiers using different data sources as input will result in uncorrelated errors. We also investigated combining all 3,172 component classifiers of Case Study 1. We found that the top-20 performance of their combination to be comparable to the top-20 performance of the best individual classifier. (Specifically, the relative improvements ranged from -2% to +12%, depending on the studied project and combination method used.) Given that the set of 3,172 contains many classifiers with very low performance, it is encouraging that their combination still achieves such high performance. This combination can be practically useful, not to improve on the best individual classifier, but to allow us to achieve close to the best performance without needing to identify the best-performing individual classifier.

8 Threats to Validity

This section discusses potential threats to the validity of our case studies.

Internal Validity. One potential threat to the internal validity of our case studies is our ground truth data collection algorithm, which was based an algorithm proposed by Fischer et al. [18]. Even though the algorithm is the state-of-the-art algorithm for linking bug reports to source code entities, recent research has questioned the algorithm's linking biases [3], since some change sets in the revision control system may not explicitly mention which bug reports are related. However, Nguyen et al. find that this bias exists even in a near-ideal dataset with high quality linking, indicating that the bias is a symptom of the underlying development process rather than the data collection methodology [41]. Another potential threat to the internal validity of our study is that of false negatives in our ground truth data. Specifically, our truth data contains links between bug reports and those source code entities that were actually changed to resolve the bug report. However, it may be the case that other source code entities could have been changed instead to resolve the same bug report. Both internal threats are mitigated in our work because whatever biases or false negatives exist, we use the same truth dataset in the evaluation of all classifiers and combination techniques, providing an equal platform for comparison.

External Validity. A potential threat to the external validity of our study is that even though we performed three extensive case studies on large, active, real-world projects, our results still must be considered in context. In particular, our studied projects represent only a fraction of all real-world projects, domains, programming languages, and development paradigms, so we cannot definitively say that our results will hold for all possible projects. We have provided our data and tools online to encourage others to replicate our work on other projects.

Construct Validity. A potential threat to the construct validity of our case studies is our use of the top-20 metric (Section 4.1.5) as a proxy for developer effectiveness. While the top-20 metric is the standard metric for evaluating bug localization performance, additional research is needed to see how well it correlates with developer effectiveness in real-world situations.

9 Conclusions and Future Work

Solving the bug localization problem has major implications for developers, since it can dramatically reduce the time and effort required to maintain software. In this paper, we cast the bug localization problem as one of classification, and analyzed the effect classifier configuration had on bug localization performance, as well as whether classifier combination could help. We summarize our main findings as follows.

- The configuration of an IR-based classifier matters.
- The best individual IR-based classifier uses the Vector Space Model, with the index built using tf-idf term weighting on all available data in the source code entities (i.e., identifiers, comments, and past bug reports for each entity), which has been stopped, stemmed, and split, and queried with all available data in the bug report (i.e., title and description) with cosine similarity.
- The best EM-based classifier uses the new bug count metric to rank source code entities.
- Classifier combination helps in almost all cases, no matter the underlying classifiers used, or the specific combination technique used.

We proposed two frameworks: one for defining and analyzing classifier configurations, and one for combining the results of disparate classifiers. We used these frameworks to conduct our empirical case studies on bug localization; researchers can use our frameworks to conduct similar analyses in other areas that consider multiple configurations of classifiers.

We found that the configuration of a classifier has a significant impact on its performance. In Eclipse, for example, the difference between a poorly configured IR classifier and a properly configured IR classifier is the difference between having a 1 in 100 chance of finding a relevant entity in the top 20 results and having better than a 1 in 2 chance. We found that prior bug localization research does not use optimal classifier configurations. Fortunately, we also found consistent results of the various configurations across all three studied projects, suggesting that the proper configuration is completely project specific, and can be applied in a general context.

By identifying which classifier configurations are best, and how to combine them in the most effective way, our results substantially advance the state-of-the-art in bug localization. Practitioners can use our results to accelerate the task of finding and fixing bugs, resulting in increased software quality and decreased maintenance costs. We provide our data, tools, and results online [55].

Since IR-based bug localization has only recently gained the attention of researchers, there are many exciting avenues to explore in future work. The most obvious avenue is the addition of classifier families to the combination framework presented in this paper, such as those used for concept location: PageRank [44], [50]; formal concept analysis [47]; dynamic analysis [46]; and static analysis [34]. Additional IR models can be considered, such as BM25F [51], BugScout [40], and other variants of LDA, such as the Relational Topic Model [9]. Recently, researchers have proposed query expansion techniques [8], which may be a useful preprocessing step to any IR-based classifiers. We also wish to investigate whether preprocessing bug reports by removing noise in the form of stack traces and code snippets could be beneficial to bug localization results. Finally, we have yet to fully investigate many possible combination techniques, such as variants of the Borda Count and Reciprocal Rank Fusion [11].

Acknowledgments

This work was funded in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Research in Motion (RIM). We would like to thank Nguyen et al. for providing us with an early version of their recent work [40] and Nicholas A. Kraft for a helpful discussion about his bug localization experience. We would also like to thank Thanh H. D. Nguyen and Yasutaka Kamei for their help with data collection. Finally, we would like to thank the anonymous reviewers for their very constructive feedback on early drafts of this work.

References

[1] R. Arnold and S. Bohner. Impact analysis — Towards a framework for comparison. In *Proceedings of the International Conference on Software Maintenance*, pages 292–301, 1993.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[3] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in Bug-Fix datasets. In *Proceedings of the 7th European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, pages 121–130, 2009.

[4] D. M. Blei and J. D. Lafferty. Topic models. In *Text Mining: Classification, Clustering, and Applications*, pages 71–94. Chapman & Hall, London, UK, 2009.

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

[6] S. Bohner and R. Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[7] G. Box and D. Behnken. Some new three level designs for the study of quantitative variables. *Technometrics*, 2(4):455–475, 1960.

[8] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1):1–50, Jan. 2012.

[9] J. Chang and D. M. Blei. Relational topic models for document networks. *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, pages 81–88, 2009.

[10] 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*, 14(1):93–130, 2008.

[11] G. V. Cormack, C. L. Clarke, and S. Buettcher. Reciprocal Rank Fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of the 32nd International Conference on Research and Development in Information Retrieval*, pages 758–759, 2009.

[12] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 433–436, 2007.

[13] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4), 2012.

[14] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[15] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol. Can better identifier splitting techniques help feature location? In *Proceedings of the 2011 International Conference on Program Comprehension*, pages 11–20, 2011.

[16] Eclipse Foundation. Eclipse. http://www.eclipse.org/, 2011.

[17] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

[18] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. *Proceedings of the 19th International Conference on Software Maintenance*, 2003.

[19] R. L. Glass. *Facts and fallacies of software engineering*. Addison-Wesley Professional, 2003.

[20] E. Hill, S. Rao, and A. Kak. On the use of stemming for concern location and bug localization in java. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, pages 1–10, 2012.

[21] T. Ho, J. Hull, and S. Srihari. Decision combination in multiple classifier systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(1):66–75, 1994.

[22] IBM. Jazz. http://www-01.ibm.com/software/rational/jazz/, 2011.

[23] E. Keogh, S. Lonardi, and C. Ratanamahatana. Towards parameter-free data mining. In *Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining*, pages 206–215, 2004.

[24] J. Kittler, M. Hatef, R. P. Duin, and J. Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.

[25] E. Kocaguneli, T. Menzies, and J. W. Keung. On the value of ensemble effort estimation. *IEEE Transactions on Software Engineering*, 38(6):1403–1416, 2012.

[26] R. Kuehl. *Design of experiments: Statistical principles of research design and analysis*. Brooks/Cole, 2000.

[27] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the 27th International Conference on Software Maintenance*, pages 113–122, 2011.

[28] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent Dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 155–164, 2008.

[29] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[30] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, and G. Antoniol. Recognizing words from source code identifiers using speech recognition techniques. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 68–77, 2010.

[31] C. D. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, UK, 2008.

[32] 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*, pages 214–223, 2004.

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

[34] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, 2011.

[35] A. T. Misirli, A. B. Bener, and B. Turhan. An industrial case study of classifier ensembles for locating software defects. *Software Quality Journal*, 19(3):515–536, 2011.

[36] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, pages 181–190, 2008.

[37] Mozilla Foundation. Bugzilla. http://www.bugzilla.org/, 2012.

[38] Mozilla Foundation. Mozilla. http://www.mozilla.org/, 2012.

[39] R. Myers, A. Khuri, and W. Carter. Response surface methodology: 1966–l988. *Technometrics*, 31(2):137–157, 1989.

[40] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 263–272, 2011.

[41] T. H. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in Bug-Fix datasets. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 259–268, 2010.

[42] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 86–96, 2004.

[43] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[44] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. *Stanford InfoLab*, 1999.

[45] M. Porter. An algorithm for suffix stripping. *Program*, 14:130, 1980.

[46] 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*, 33(6):420–432, 2007.

[47] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 37–48, 2007.

[48] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. BugCache for inspections: Hit or miss? In *Proceedings of the 19th Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 322–331, 2011.

[49] 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*, pages 43–52, 2011.

[50] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proceedings of the 18th International Conference on Program Comprehension*, pages 14–23, 2010.

[51] S. Robertson, H. Zaragoza, and M. Taylor. Simple BM25 extension to multiple weighted fields. In *Proceedings of the 13th International Conference on Information and Knowledge Management*, pages 42–49, 2004.

[52] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):620, 1975.

[53] R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Transactions on Software Engineering*, 31(6):495–510, 2005.

[54] J. Sliwerski, T. Zimmerman, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2nd Working Conference on Mining Software Repositories*, 2005.

[55] S. W. Thomas. http://sailhome.cs.queensu.ca/replication/sthomas/TSE2013, 2012.

[56] S. W. Thomas. Mining software repositories with topic models. Technical Report 2012-586, School of Computing, Queen's University, 2012.

[57] J. Tukey. The philosophy of multiple comparisons. *Statistical Science*, 6(1):100–116, 1991.

[58] J. Tukey and H. Braun. *The collected works of John W. Tukey: Multiple comparisons, 1948-1983*, volume 8. Chapman & Hall/CRC, 1994.

[59] M. Van Erp and L. Schomaker. Variants of the Borda Count method for combining ranked classifier hypotheses. In *Proceedings of the 7th International Workshop on Frontiers in Handwriting Recognition*, pages 443–452, 2000.

[60] H. Wang, T. M. Khoshgoftaar, and A. Napolitano. Software measurement data reduction using ensemble techniques. *Neurocomputing*, 92(0):124–132, 2012.

[61] X. Wei and W. B. Croft. LDA-based document models for ad-hoc retrieval. In *Proceedings of the 29th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 178–185, 2006.

[62] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering*, 2007.

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

Meiyappan Nagappan Biography text here.

Dorothea Blostein Biography text here.

Ahmed E. Hassan Biography text here.