CoLUA: Automatically Predicting Configuration Bug Reports and Extracting Configuration Options

Wei Wen, Tingting Yu, Jane Huffman Hayes
Department of Computer Science and Engineering, University of Kentucky, Lexington, KY 40508, USA
wei.wen0@uky.edu, {tyu, hayes}@cs.uky.edu

***Abstract*—Configuration bugs are among the dominant causes of software failures. Software organizations often use bug-tracking systems to manage bug reports collected from developers and users. In order for software developers to understand and reproduce configuration bugs, it is vital for them to know whether a bug in the bug report is related to configuration issues; this is not often easily discerned due to a lack of easy to spot terminology in the bug reports. In addition, to locate and fix a configuration bug, a developer needs to know which configuration options are associated with the bug. To address these two problems, we introduce CoLUA, a two-step automated approach that combines natural language processing, information retrieval, and machine learning. In the first step, CoLUA selects features from the textual information in the bug reports, and uses various machine learning techniques to build classification models; developers can use these models to label a bug report as either a configuration bug report or a non-configuration bug report. In the second step, CoLUA identifies which configuration options are involved in the labeled configuration bug reports. We evaluate CoLUA on 900 bug reports from three large open source software systems. The results show that CoLUA predicts configuration bug reports with high accuracy and that it effectively identifies the root causes of configuration options.**

I. Introduction

Modern software systems are highly-configurable, allowing users to customize a large number of configuration options while retaining a core set of functionality. For example, a user can augment their browser with sophisticated add-ins, change their Eclipse build settings (i.e., configuration options) to use different versions of the JDK or specified libraries depending on the project, build a specific Linux kernel configuration, etc. While such customizability provides benefit to users, the complexity of the configuration space and the sophisticated constraints among configuration settings complicates the process of testing and debugging. Thus, it is not surprising that many configuration bugs remain undetected and later surface in the field. A study by Yin et al. [47] shows that up to 31% of bugs are related to misconfiguations in several open source and commercial software systems [47], where a majority of misconfigurations (up to 85.5%) are due to mistakes in setting configuration options.

Software organizations use bug-tracking systems to manage bug reports collected from developers and users. A developer who is assigned to a given bug report first needs to determine the type of the bug, i.e., whether this bug is configuration-related or not. For example, in Apache, bug reports related to configurations are explicitly labeled as configuration bugs. The next step is to use the anomalous configuration options to reproduce the bug. However, developers with insufficient domain knowledge may incorrectly label a bug report or spend time determining the bug type (time that could have been well spent elsewhere). In addition, to understand the bug, developers often need to look through the bug descriptions, which can be lengthy, verbose, and involve multiple developers and users. In fact, Rastkar et al. found that almost one-third of the bug reports in the open source projects Firefox, Thunderbird, and Eclipse Platform in the 2011-2012 period were 300 words or longer (deemed lengthy) [37].

Further, it is often non-trivial to determine which configuration options are relevant in order to reproduce a bug. For example, if a developer knows that a bug report describes a configuration bug related to javascript in a browser application, he or she may not be able to quickly determine what the real name of the configuration option is in the configuration database (e.g., `Browser.urlbar.filter.javascript`). If the developer wants to fix this bug, he or she may spend an exorbitant amount of time searching through the configuration database to find which option is relevant. Recent work by Wiklund et al. reported that the majority of the impediments for developers stemmed from the failure to figure out the correct configuration options and environment settings [42]. A user-study conducted by Wang et al. [41] indicates that the average person time for finding the cause of a bug from reading a bug report is 9-15 minutes.

Therefore, there is a need for an effective technique to reduce the manual effort required to label configuration bug reports and to identify the root cause configuration options. There has been some research on debugging and diagnosing configuration bugs [36], [49]. For example, Zhang et al. [49] propose a technique to diagnose crashing and non-crashing errors related to software misconfigurations. These techniques, however, have focused on the implementation with the assumption that the bugs are already labeled as configuration bugs. Xia et al. [44] use text mining to label configuration bug reports related to system setting and compatabilities. Nevertheless, their technique neither predicts bug reports related to misuse of configuration options nor extracts such configuration options.

In this paper, we propose a framework, CoLUA (**Co**nfiguration bug **L**earner **U**ncovers **A**pproximated options), that aims to improve configuration-aware techniques and help ease developers' process of debugging and reproducing bugs that need specific configurations for exposition. CoLUA focuses on configuration bugs due to incorrect settings of configuration options. Given a bug report, CoLUA determines whether it is a configuration bug, and if it is, the approach automatically suggests configuration options to help developers reproduce the bug. CoLUA is comprised of two steps, combining machine learning and natural language processing (NLP) techniques. First, CoLUA trains classification models on the historical bug reports with known labels to classify a new bug report as either a configuration bug report or a non-configuration bug report. In the second step, CoLUA applies Information Retrieval (IR) and NLP to the configuration bug reports. It then parses both configuration options and the query (the bug report of interest). The queries and configuration options are then matched, ranked, and returned to the developer.

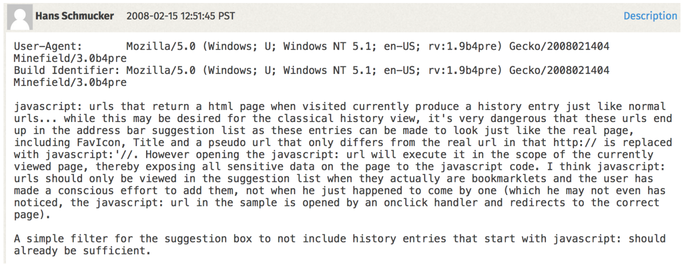
CoLUA provides at least two benefits. First, developers can label configuration bug reports in an automated and timely manner. Second, with the configuration query component, CoLUA allows developers to approximate configuration options that are relevant to the bugs. This can improve the configuration debugging and diagnosis process. Note that CoLUA does not substitute for, but complements, existing configuration-aware debugging techniques [36], [49]. Such techniques can use CoLUA to predict whether a bug is configuration-related, and then decide the appropriate course of action. Also, the configuration options identified by CoLUA can be used to narrow down the list of candidate root causes.

To evaluate CoLUA, we apply the approach to three popular open source projects (i.e., Mozilla, Apache, and MySQL). In total, we analyze 900 bug reports using three classification techniques (Naive Bayes, Decision Tree, and Logistic Regression). We measure the performance of the approaches in terms of the F-measure. Our results show that CoLUA is effective at discriminating configuration bug reports from non-configuration bug reports. The average F-measure is 0.73 over three techniques across all three subjects, compared to 0.33 for the baseline method of ZeroR. In addition, CoLUA is able to extract configuration options from configuration bug reports with a high success rate.

In summary, this paper contributes the following:

- A framework for learning natural-language models to automate the classification of configuration bug reports. The techniques used to build models are domain-specific thus accounting for the unique characteristics of the configuration software systems.
- A first approach that can automatically extract configuration options that are relevant to bugs from bug report descriptions.
- An extensive empirical evaluation on 900 bug reports from three large open source software systems that shows that our approach is effective.

In the next section we present a motivating example and background. We then describe the CoLUA approach in Section III. Our empirical study follows in Sections IV and V, followed by discussion in Section VI. We present related work in Section VII, and end with conclusions in Section VIII.


Hans Schmucker 2008-02-15 12:51:45 PST Description

```
User-Agent:       Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9b4pre) Gecko/2008021404
Minefield/3.0b4pre
Build Identifier: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9b4pre) Gecko/2008021404
Minefield/3.0b4pre

javascript: urls that return a html page when visited currently produce a history entry just like normal
urls... while this may be desired for the classical history view, it's very dangerous that these urls end
up in the address bar suggestion list as these entries can be made to look just like the real page,
including FavIcon, Title and a pseudo url that only differs from the real url in that http:// is replaced
with javascript:'//. However opening the javascript: url will execute it in the scope of the currently
viewed page, thereby exposing all sensitive data on the page to the javascript code. I think javascript:
urls should only be viewed in the suggestion list when they actually are bookmarklets and the user has
made a conscious effort to add them, not when he just happened to come by one (which he may not even has
noticed, the javascript: url in the sample is opened by an onclick handler and redirects to the correct
page).

A simple filter for the suggestion box to not include history entries that start with javascript: should
already be sufficient.
```

Fig. 1. **A configuration bug report.**

II. Motivation

A configurable system is a software system with a core set of functionality and a set of variable features which are defined by a set of configuration options [23]. A configuration option can be specified in a configuration file, source code, and/or in a user input option. A configuration database (also called a configuration model) consists of all the configuration options in an application. Constructing an effective configuration model has been well discussed in recent work [23]. In this work, we assume that the configuration model is known. Changes to the value of a configuration option may change the program's behavior in some way. If such changes cause the system to behave incorrectly, a *configuration bug* occurs. In fact, there are several categories of configuration-related issues, including 1) software bug in the function of handling configuration parameters (e.g., missing implementation for case items), 2) configuration error due to wrong value selection by users, 3) configuration error due to default setting assigned by developers, and 4) configuration error in external software, libraries, or operating systems. In this work, configuration bugs refer to categories 2) and 3). We use Firefox, a popular web browser and also a highly-configurable system, to motivate our approach.

In Firefox, when the configuration option O_1 = `Browser.urlbar.filter.javascript` is set to false, it allows "javascript:" URLs to appear in the autocomplete dropdown of the location bar. This can cause potential security threats. Figure 1 shows a bug report associated with the configuration option O_1. This bug involves 22 comments and took 21 months to fix. In fact, at the end of the 21 months, a single change to the value of this configuration option fixed the problem.

Suppose that an inexperienced developer is assigned to work on this bug. He or she may spend much time figuring out that it is a configuration bug. In such a case, the developer has to inspect the source code, try various inputs and configurations to hopefully reproduce, locate, and fix the bug. Even if an experienced developer is assigned to work on this bug and notices that it is related to configurations based on the system's specific behavior (e.g., mouse scrolling events), she may not be able to quickly determine the real configuration option from the configuration database (i.e., O_1) as there are approximately 1650 possible configuration options in the configuration model of Firefox. Therefore, to ease the process of configuration debugging and diagnosis, we need

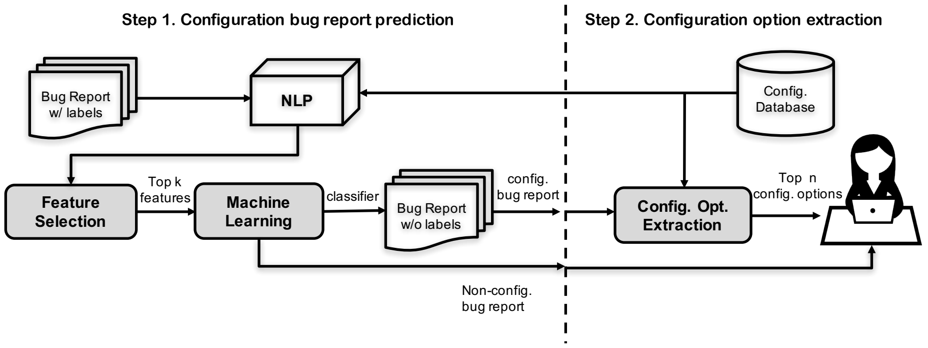


Fig. 2. **The overview of our CoLUA framework.**

new techniques that can identify a configuration bug report and link the bug to specific configuration options.

We have observed that natural language descriptions of a bug report provide information to indicate whether a bug is related to configuration options. In the running example, the word "bookmarklets" is likely to be an indicator of a configuration problem. The word "javascript" ties to the name of a configuration option O_1 obtained from the configuration model. Based on these observations, we decided to use natural language processing (NLP) techniques that process text reports and converts them into individual words to be used as features in machine learning. Developers can use the trained machine learning classifiers to label a bug report as either a configuration bug report or a non-configuration bug report. Also, with the help of NLP and information retrieval (IR), the classifier returns a list of ranked configuration options extracted from the configuration bug reports to the developers. In our running example, O_1 is ranked at the top of the list, followed by the `Javascript.allow.mailnews` option and `Mailnews.start page.url` option.

III. Approach

Figure 2 shows an overview of the CoLUA framework. CoLUA consists of two steps. In the first step, CoLUA applies NLP to extract textual features from a set of training bug reports, i.e., bug reports with known labels (configuration vs. non-configuration). CoLUA then uses Chi-Square [38], [45], a state-of-the-art feature selection technique, to select a subset of relevant textual features to improve the prediction performance and avoid overfitting [45]. Next, CoLUA applies machine learning to train classifiers based on the selected features and labeled bug reports. The trained classifiers are then applied on the unlabeled bug reports to classify a bug report as either a configuration bug report (CBR) or a non-configuration bug report (NCBR). The CBRs are retained for further processing (i.e., configuration option extraction), whearas the NCBRs are sent to the developers. In this work, we investigate three classification techniques: naive Bayes, decision tree, and logistic regression.

In the second step, the configuration option extraction component takes the labeled CBRs and the configuration database as inputs. A configuration database contains all configuration options for an application. Next, the configuration extraction component invokes the NLP procedures to find the similarities between the bug reports and each of the configuration options. The output of this step is a ranked list of configuration options that are likely the root cause of the configuration bug (with a value showing the score). We describe the two steps in the sections that follow.

A. Feature Selection

The feature selection component first creates the *term frequency* matrix by analyzing the texts of bug reports. To do this, CoLUA performs several basic NLP text parsing steps to retain terms that are most likely to be the indicators of CBRs and NCBRs. CoLUA next applies chi-Square and bigram to select high informative terms as features for training classifiers.

1) Basic Text Parsing: The basic steps involved in NLP are word tokenization, stopword removal, stemming, lemmatization, part-of-speech tagging, and chunking and chinking. In this work, the *basic text parser* chooses several commonly used NLP steps [44], [50], including *word tokenization*, *stopword removal*, and *lemmatization*, to convert bug reports into a "bag of words" as preparation for feature selection. The process of splitting paragraphs into sentences or splitting sentences into words is referred to as *word tokenization*. Since words are frequently used as features in machine learning, specifically text mining, word tokenization is often necessary.

CoLUA uses NLP to parse terms (i.e., words and phrases) from each bug report to create a *term frequency* matrix. A bug report (b) contains a bag of words and each word in b is a term t. *Term frequency* ($tf_{t,d}$) is defined as the number of occurrences of a term t in the document d. If t is not in d, the value of $tf_{t,d}$ is zero. For example, the term "config" occurs one time in bug report 1 , zero times in bug report 2, and two times in bug report 3.

After this step, however, there still exist some words, such as *the*, *this*, *when*, *me*, that are common in English but do not provide domain relevant information in our context. These words are commonly referred to as *stop words*. The parser filters stop words prior to further processing, using a stop words list. The terms in the stop words list are not entered into the term frequency matrix. However, the default stopword corpus in the NLP component (e.g., NLTK package) is not sufficient. To address this problem, CoLUA plugins a domain-specific dictionary, including stopwords that are specific to bug reports (e.g., user names) and applications (e.g., firefox, org).

These words carry little discriminating power when it comes to configurations.

While the stopword removal can shorten the texts without losing the core information, the texts may still fail to match if they have the same concept using words in different forms or use a different word with similar meaning. As such, we apply *stemming* and *lemmatization*. Stemming is the process of removing the ending of a derived word to get its root form. For example, "configurations" becomes "configur.". Lemmatization, on the other hand, always returns the true root form of a word. For example, developers may write "preference" as "pref," and "configuration" as "config." We return such words to their original form. Therefore, the term frequency matrix is updated with respect to the converted words.

2) Feature Selection Using Chi-Square: After textual features are extracted from the training bug reports, We use Chi-Square [38], [45] to select top k features that are specific for predicting configuration bug reports. Chi-Square measures how common a feature is in a particular class (label) compared to how common it is in other classes (labels). The higher the Chi-Square score, the more likely the feature is to be associated with the class. For simplicity, assume that there are bug reports of two classes CBR and NCBR. Let n_{ii} be the counts that the feature (say w) in consideration occurs in CBR, n_{io} be the counts that w occurs in reports of NCBR, n_{oi} be the counts of all features except w that occur in reports of class CBR, n_{oo} be the counts of all features except w that occur in class NCBR, and n_{xx} be the counts of all features that occur in reports of both types. The Chi-Square score that shows how likely this feature is associated with type CBR is calculated as:

$$Chi\_sq\_score = \frac{n_{xx} \times (n_{ii} \times n_{oo} - n_{oo} \times n_{oi})^2}{(n_{ii}+n_{io}) \times (n_{ii}+n_{io}) \times (n_{io}+n_{oo}) \times (n_{io}+n_{oo})}$$

The score that shows how likely w is associated with class NCBR can be calculated similarly.

A feature (term) is an n-gram: a contiguous sequence of n items from a given sequence of text. In this work, we investigate the effectiveness of using unigrams (single word tokens) and bigrams (two consecutive word tokens) as features (i.e., $n < 3$) because they are commonly used in bug report mining techniques [6], [32], [39]. Bigram identifies two words that are likely to co-occur. For example, a bug report containing "not configuration" indicates that it is not related to configuration bugs. If individual words are used as the only features, this report may be incorrectly classified as configuration-related. Thus, including bigrams may increase the chance of correctly classifying bug reports. The likelihood of two words occurring together is calculated using Chi-Square. The only change is that now instead of the association between a word and a label, the association is between two words.

B. Ranking

After we apply Chi-Square and bigram to compute the scores for each unigram and bigram, we rank these scores from high to low to generate a ranked list. The higher the score, the more important the unigram/bigram is for distinguishing a label. We select unigrams and bigrams whose feature selection scores are in the top k of the ranked list and remove the others. In this way, we reduce the number of features in the model building phase and also in the prediction phase. Research has showed that the k value ranging from 30 to 200 is useful for prediction [1]. As such, by default we used 100 for the value of k. The selected features consist of $m\%$ unigrams and $n\%$ bigrams of the total k features. By default, m is 80 and n is 20. We further show how performance varies across different numbers of selected features and across different ratios of unigrams to bigrams in Section VI.

C. Machine Learning

After selecting the features in the labeled bug reports, CoLUA uses machine learning to build classifiers and uses them to help identify a new bug report as CBR or NCBR. In the prediction phase, the classifier is then used to predict whether a bug report with unknown label is a configuration bug or not.

D. Extracting Configuration Options

Configuration option extraction takes as inputs both CBRs and a configuration database of an application.

1) Splitting Configuration Options: CoLUA extracts configuration options from the user manuals of the studied system to form a configuration database. Once we have the configuration options, we split them into sets of words. The configuration options can use camel case or have dots or underlines separating the words. In processing configurations, we split the words by camel case, period (.), underscore (_), and dash(-), such as `Browser.chrome.image_icons.max_size` and `Browser.urlbar.filter.javascript`. The words are restored to their root forms with lemmatization.

2) Matching and Ranking: Configuration bug reports are first processed as described in Section III-A, except that Chi-Square is not employed. Next, the splitting method described above is applied to the configuration database. Once we have parsed both the configuration database and the bug reports, the next step is to suggest configuration options that are most relevant to each CBR. We compute similarity between a bug report and a configuration option. To do this, we use the *term frequency-inverse document frequency (tf-idf)* algorithm [29] based on the term frequency matrix described in Section III-A. Tf-idf has been widely used in information retrieval [40] to measure similarity between queries and documents, so we use it to link a bug report to a configuration option.

Here, each configuration option is considered a document d that contains a bag of words using the splitting method in Section III-D1. A configuration database that consists of N configuration options forms a document corpus. As mentioned in Section III-A, *term frequency* ($tf_{t,d}$) is defined as the number of occurrences of a term t in the document d. If t is not in d, the value of $tf_{t,d}$ is zero. *Document frequency* (df_t) is defined as the number of documents in the corpus that contain the term t. If t does not exist in any documents in the

TABLE I
RANKING TERMS IN THE EXAMPLE CONFIGURATION OPTION

term	*tf*	*df*	*idf*	*tf-idf*
browser	1	28	0.16	0.16
url	1	31	0.18	0.18
multiple	0	-	-	-
filter	1	18	0.15	0.15
javascript	1	22	0.14	0.14
...	...	...	...	...

corpus, df_t is equal to zero. The *inverse document frequency* (idf_t) is used to reduce the effect of terms that appear in many documents; it is defined as:

$$idf_t = log\frac{N}{df_t}$$

So a large value for df_t makes idf_t small. This indicates that if a term exists in many documents, it carries less discriminating power. The weight $tf\text{-}idf$ for a term t in d is defined as:

$$tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t$$

As we can see, a term in d would have a heavier weight if it occurs many times in just a few documents (both $tf_{t,d}$ and idf_t are large). Finally, we calculate the similarity score by adding the weights of all terms that occur in both the bug report and configuration options, which is defined as:

$$similarity(b, d) = \sum_{t \in b} tf - idf_{t,d}$$

Consider the example in Section II where the bag of words, after parsing, there are a list of terms for the bug report b: {browser, url, multiple, filter, javascript, ...}. The {browser, url, bar, filter, javascript} is the corresponding document d (i,e., the configuration option `Browser.urlbar.filter.javascript`). Table I shows the statistics of each term in b. The overall similarity score is the sum of the weights of all the terms (0.16 + 0.18 + 0 + 0.15 + 0.14 = 0.64). The term *multiple* fails to match any word in d and contributes zero weight. After assigning each configuration option a similarity score for a given bug report, all configuration options are ranked in decreasing order with respect to the score. The top n preferences are sent to the users. In our study, n is 10.

E. Implementation

CoLUA begins by extracting useful information from a webpage (given the URL) that contains bug reports, because the webpage can contain extraneous information (e.g., version, reporters name, priority) that needs to be excluded. A text file is thus built that contains only the relevant information from the webpage; we generally include the title and the comment text from the webpage. We use Python's NLTK package to extract features from the text files. In the configuration option extracting component, we use Sklearn to calculate similarity scores. Specificially, the API `TfidfVectorizer` converts documents to a matrix of *tf-idf*. We also include both unigrams and bigrams as the API to help increase the chance of finding the right similarities between bug reports and configurations.

TABLE II
SUBJECTS AND THEIR CHARACTERISTICS

Apps	*versions*	*# Options*	*CBR*		*NCBR*	
			labeled	unlabeled	labeled	unlabeled
Apache	v1.3–v2.4	1,145	100	50	100	50
MySQL	v5.0–v5.7	1,429	100	50	100	50
Mozilla	v18–v47	1,650	100	50	100	50

We use the classes `BigramAssocMeasures`, which contains an implementation of Chi-Square and `BigramCollocationFinder` in the NLTK modules metrics and collocations to identify the highly informative words and commonly occurring bigrams. The selected features are arranged in the form of a dictionary with the words as the key and the assigned values as the values of the keys. The format we use is {`word: true`}, where word is the word selected and the value is `true`. We uniformly use `True` as the value for a word (the key) to indicate that a word appears in that report.

IV. EMPIRICAL STUDY

We perform a case study aimed at evaluating CoLUA that addresses three research questions. Supporting data on the queries used and the associated results can be found on our website[1].

RQ1: How effective is CoLUA at classifying bug reports into CBRs and NCBRs using different classification models?

RQ2: Can the learned model be used to predict new configuration bug reports?

RQ3: How effective is CoLUA at finding the correct configuration options?

RQ1 lets us assess the effectiveness of CoLUA for classifying CBRs across different classification techniques. We also use ZeroR as a baseline approach [17]. RQ2 lets us investigate whether a predictor trained from the labeled bug reports can be used to predict new bug reports. RQ3 lets us evaluate the effectiveness of CoLUA's configuration option extraction component.

A. Objects of Analysis

As objects of analysis we chose three large, mature, and popular open-source software projects: Apache, MySQL, and Mozilla. With millions of lines of publicly accessible code and well maintained bug repositories, these subjects have been widely used by existing bug characteristic studies [24], [47], [48]. The selected programs and their versions are listed in Columns 1-2 of Table II. The subject programs cover various application spectrums - the world's most used HTTP server, the world's most popular database engine, and a leading web browser. All three projects started in the early 2000's and each has over ten years of bug reports.

To obtain training and testing data, we selected CBRs and NCBRs from the bug tracking systems of the three projects. In the Mozilla project, some bug reports had been labeled

[1]http://cs.uky.edu/~tyu/research/CoLUA

as CBRs, so we randomly selected 100 of them. However, NCBRs were not labeled in Mozilla, and neither CBRs nor NCBRs were labeled in the other two projects. In such cases, we manually inspected all confirmed bug reports until 150 bug reports were selected for each of the CBR and NCBR categories across all three projects. Since usually there were more NCBRs than CBRs, we improved the process of searching for CBRs by using a set of configuration-related keywords ("configuration," "option," "preference," "setting," etc.). Finally, we randomly picked 100 CBRs and 100 NCBRs from the selected bug reports for each subject as a dataset for that subject.

During the manual inspection, we read the reports with sufficient details in the bug descriptions and examined the discussions posted by commentators to decide if the inspected bug was a CBR or not. To ensure the correctness of our results, the manual inspections were performed independently by three graduate students, each of whom has at least two years of software development experience. Any time there was dissension, the authors and the inspectors discussed to reach a consensus. Note that keyword search alone is inadequate, as we found that 67% of bug reports were false positives when using the keyword search .

RQ2 requires new datasets to validate the learned classifiers. We repeated the above manual process and selected 50 CBRs and 50 NCBRs. While larger number of bug reports may yield better evaluation, the cost of the manual process is quite high: the understanding and preparation of the object used in the study and the conduct of the study required between 400 and 600 hours of researcher time.

To answer RQ3, we also need to know the configuration database for each subject. We collected this information by studying all artifacts that are publicly available to users, including documents (e.g., user manuals and on-line help pages), configuration files, and source code. In Firefox, we also utilized the APIs that have been provided to programmatically manipulate internal data structures that hold configuration information as well as studied the `about:config` page (a utility for modifying configurations). This process yielded the total number of configuration options for the three subjects (Column 5 of Table II).

B. Variables and Measures

This section describes the independent variable and the dependent variables.

1) Independent Variable: Our independent variable involves the techniques used in our study. In CoLUA, we consider three classification techniques: NaiveBayes and Decision Tree, and Logistic Regression from Weka [18], We chose them because they are popular and have been shown to be effective at classifying bug reports [37] as well as predicting software defects in a recent study [16].

In addition to the three classification techniques, we use ZeroR as the baseline for comparison with our approach. During training, ZeroR ignores the features and relies only on the labels for predicting. Although it does not have much predicting capability, it establishes the lowest possible predictability that a classifier should have. It works by constructing a frequency table for the labels in the training data and selects the most frequent values of the testing data in predicting.

2) Dependent Variables: As dependent variables, we chose metrics allowing us to answer each of our three research questions.

Prediction performance: We used the F-measure to evaluate performance of CoLUA across the three classification models and the baseline model ZeroR. The F-measure usually represents the harmonic mean of precision and recall. The computed F-measure values are between 0 and 1 and a larger F-Measure value indicates a higher classification quality. The following outcomes are used to define precision, recall, and F-measure: (1) A CBR is correctly classified as a CBR ($a \rightarrow a$); (2) A CBR is incorrectly classified as a NCBR ($a \rightarrow b$); and (3) A NCBR is incorrectly classified as a CBR ($b \rightarrow a$). We use the above outcomes to evaluate the prediction accuracy of our models with the following measures:

- **Precision:** the number of bug reports correctly classified as CBRs ($N_{a\rightarrow a}$) over the number of all bug reports classified as CBRs.
$$PrecisionP(a) = \frac{N_{a\rightarrow a}}{N_{a\rightarrow a}+N_{b\rightarrow a}}$$
- **Recall:** the number of bug reports correctly classified as CBRs ($N_{a\rightarrow a}$) over the total number of CBRs.
$$RecallR(a) = \frac{N_{a\rightarrow a}}{N_{a\rightarrow a}+N_{a\rightarrow b}}$$
- **F-measure:** a composite measure of precision $P(b)$ and recall $R(b)$ for CBRs.
$$F-measureF(a) = \frac{2*P(a)*R(a)}{P(a)+R(a)}$$

The F-measure for NCBRs can be calculated similarly. We consider F-measures over 0.6 to be good [19].

To evaluate our prediction models and answer RQ1, we used 10-fold cross validation, which has been widely used to evaluate prediction models [26], [30]. Of these 10 folds, 9 folds are used to train a classification model while the 10^{th} fold is used to evaluate the performance of the model. The whole process is iterated 10 times, and the average performance across the 10 iterations is recorded. Since 10-fold cross validation randomly samples instances and puts them in ten folds [3], we repeated this process 100 times and calculated their average for each prediction model to avoid sampling bias [26].

Statistical Tests: We report statistical measures when applicable. For example, in RQ1 we assess whether prediction performance of different classification techniques were statistically significant. To do this, we applied the t-test to the sets of F-measures, comparing each pair of two techniques. We checked if the mean of F-measure values of technique A was greater than the mean of F-measures of technique B at the 95% confidence level ($p-value < 0.05$).

Ranking effectiveness: To evaluate how accurately CoLUA extracts configuration options, for each subject we ran the new sample of the 50 CBRs and evaluated whether the correct answer (ground truth) was found and at which rank (for effectiveness). This ranking strategy has been widely adopted by existing fault localization techniques [11], [25].

For Mozilla, the configuration bug reports were already labeled, so we use that information as ground truth to test the accuracy of ranking. For bug reports from the other two projects, the association of a bug report with one or more configuration options was manually identified using the process described in Section IV-A.

C. Study Operation

We used NLTK to generate the term-by-document frequency matrix from all the terms in the bug reports and selected k terms as features based on their Chi-Square scores. Among the k terms, $m\%$ of them were unigrams and $n\%$ of them were bigrams. By default, k = 100, m = 80, and n = 20. In Section VI, we further investigate how varying these values can affect the performance of CoLUA.

We applied the four classification techniques on each of the three training sets to build prediction models. We then applied the trained models to test data sets and calculated their F-measures. In the second step, we ran the configuration identification component on all CBRs in the test datasets.

D. Threats to Validity

The primary threat to external validity for this study involves the representativeness of our subjects and bug reports. Other subjects may exhibit different behaviors. Data recorded in bug tracking systems and code version histories can have a systematic bias relative to the full population of bug reports [9] and can be incomplete or incorrect [4]. However, we do reduce this threat to some extent by using several varieties of well studied open source code subjects and bug sources for our study. We also used two hundred bug reports from each system. We cannot claim that our results can be generalized to all systems of all domains though.

The primary threat to internal validity involves the use of keyword search and manual inspection to identify the CBRs and NCBRs. To minimize the risk of incorrect results given by manual inspection, bugs were labeled as configuration bugs independently by three people. The risk of not analyzing all configuration bugs cannot be fully eliminated. However, combining keyword search and manual inspection is an effective technique to identify bugs of a specific type from a large pool of generic bugs and has been used successfully in prior studies [24], [31], [47].

The primary threat to construct validity involves the dataset and metrics used in the study. To mitigate this threat, we used bug reports from the bug tracking systems of the three subjects which are publicly available and generally well understood. We also used the well known, accepted, and validated measures of recall, precision, and F-measure. We minimized threats to conclusion validity by performing statistical analysis of our results.

V. Results and Analysis

Results for each research question are presented below.

TABLE III
RQ1: Classification Results

Program	*Classes*	*ZeroR*	*NB*	*LR*	*DT*
Mozilla	Conf.	0.667	0.787	0.775	0.832
	Non-config.	0.0	0.811	0.719	0.782
Apache	Conf.	0.667	0.889	0.574	0.940
	Non-config.	0.0	0.891	0.623	0.782
MySQL	Conf.	0.667	0.607	0.701	0.679
	Non-config.	0.0	0.385	0.718	0.615
Average	Conf.	0.667	0.761	0.683	0.817
	Non-config.	0.0	0.696	0.687	0.726

TABLE IV
RQ2: Predicting New Bug Reports

Program	*Classes*	*NB*	*LR*	*DT*
Mozilla	Conf.	0.775	0.779	0.815
	Non-config.	0.811	0.719	0.782
Apache	Conf.	0.891	0.609	0.922
	Non-config.	0.891	0.623	0.782
MySQL	Conf.	0.621	0.698	0.671
	Non-config.	0.393	0.723	0.634

A. RQ1: Effectiveness of Prediction

We use the F-measure to evaluate prediction performance for the four classification algorithms, as described in Section IV.

Table III summarizes the mean of F-measures from 100 ten-fold cross validations for four classification techniques computed for the CBRs and NCBRs across all three subjects. As we can see from the table, ZeroR performed the worst. It achieved F-measure of 0.667 for CBRs but F-measure of 0 for NCBRs in all three open source projects. In fact, each of the other three techniques was *statistically* better than Zero-R across all subjects. Specifically, the improvement for individual techniques over Zero-R with averaged F-measures ranged from 107.6% to 132.9%. The F-measure values across the three techniques were greater than 0.6 in 16 out of 18 models. These results imply that *CoLUA is effective at predicting configuration bug reports*.

Inspection of all three classifiers in all subjects suggests that, for each technique, there exists at least one case that outperformed the other two. For example, in Apache, Naive-Bayes was the best, whereas in MySQL, Logistic Regression did better than the other two techniques. This difference could be related to the styles of the bug reports in the three subjects and the diversity of terms in the bug reports (this difference is discussed further in Section VI). In fact, there is no statistical difference between the effectiveness of the three techniques. This result indicates that *it is always a good idea to include all these classifiers in the preliminary work and identify one or a few that are better for the specific data source*.

B. RQ2: New Bug Reports Prediction

RQ2 investigates whether a classifier learned from the labeled datasets can be used for predicting new bug reports. For each subject, we applied classification models built from the labeled data set (100 CBRs and 100 NCBRs) to its unlabeled dataset (50 CBRs and 50 NCBRs). We then checked the accuracy of the prediction by assessing the performance

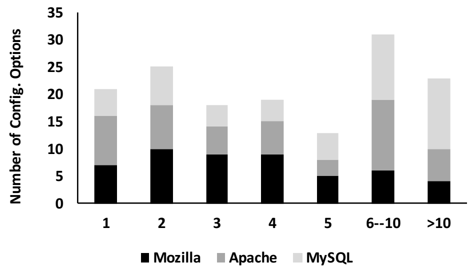


Fig. 3. **Ranking distribution for Mozilla, Apache, and MySQL.**

TABLE V

RANKING RESULTS FOR MOZILLA, APACHE, AND MYSQL CBRS

Program	*# CBRs*	*Top-10 Rate*	*Rank Range*
Mozilla	50	92%	1-92
Apache	50	88%	1-113
MySQL	50	74%	1-108

of each model. The prediction results are shown in Table IV. For the three classification techniques, the F-measure values were greater than 0.6 for 17 of the 18 models, indicating that the classifiers are effective. Only the BayesNet model learned from MySQL had low prediction effectiveness (F-measure = 0.359). Overall, these results suggest that *CoLUA is effective at predicting configuration bug reports that exist in unseen new bug reports*.

C. RQ3: Effectiveness of Configuration Option Extraction

The effectiveness of extracting configuration options is measured by the success rate. To compute success rate, we examine the rank score for each ranked list of configuration options. Specifically, for each CBR, if the correct configuration option (i.e., ground truth) is ranked first (in the first position of the ranked list), the rank is 1. If the ground truth is at the 10th position in the list, it is 10, etc. When there are multiple configuration options specified as the ground truth, we only consider the first one that CoLUA can find. For example, if 20 out of 50 are found in the top 5 of the ranked list, the top-5 success rate is 40%. The data is shown in Table V. In this table, we see the number of CBRs, the success rate in the top-10 range, and the range of the ranks where the answer (i.e., configuration options) is returned. The average success rate is 84.7%, i.e., 84.7% configuration options are ranked in the top-10 among thousands of configuration options. These results indicate that *CoLUA is effective at extracting root cause configuration options*.

To investigate the ranking distribution of all results, data is presented in Figure 3 and Table VI. In both the table and the graph we break out our results, showing those in the 1st position, 2nd position, ..., those found in position 5-10, and those found beyond the top-10. As can be seen, 57.3% of the CBRs that return correct results appear in the top 5 positions of the returned list. We believe that using additional lexical information may help to improve these results.

VI. DISCUSSION

We now explore additional observations relevant to our study.

TABLE VI

RANKING DISTRIBUTION FOR MOZILLA, APACHE, AND MYSQL

Program	**1**	**2**	**3**	**4**	**5**	**6-10**	**> 10**
Mozilla	7	10	9	9	5	6	4
Apache	9	8	5	6	3	13	6
MySQL	5	7	4	4	5	12	13

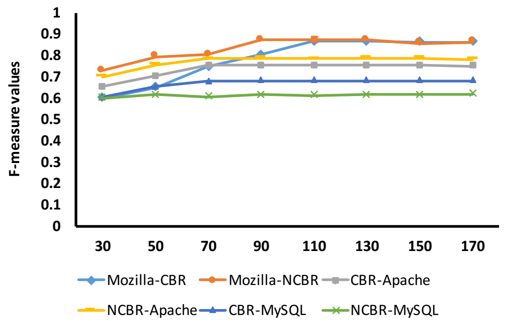


Fig. 4. **Number of features varying from 30 to 190 with the ratio of unigrams to bigrams 8:2.**

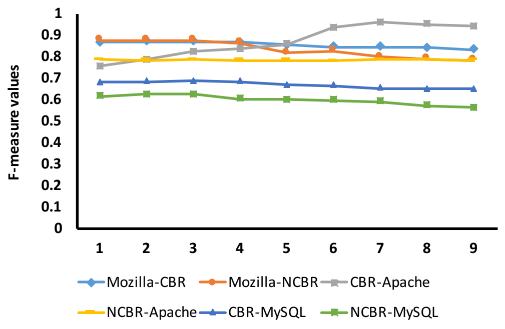


Fig. 5. **Ratio of unigrams to bigrams varying from 1:9 to 9:1.**

A. Effectiveness of Feature Selection

Since we use information gain and bigram to select features for training classifiers, we would like to investigate whether the use of the two techniques improves the performance of CoLUA. We use AW to denote the technique that uses all unigrams (i.e., feature selection is not applied), UNG to denote that Chi-Squaren on only unigrams is applied, and BIG to denote that Chi-Square on bigrams is applied.

Table VII summarizes the F-measure values computed for AW, UNG, and BIG across all three subjects for both CBRs and NCBRs. Comparing UNG to AW, UNG *statistically* outperformed AW in terms of F-measure values for all three subjects, with average improvement of 17.3% for CBRs and 23.4% for NCBRs. However, the improvement of BIG over AW varies across subjects and is not overall significant, indicating that including bigrams can increase performance somewhat sometimes, but is not as effective as informative unigrams. We next examine the three techniques in individual subjects.

In Mozilla, there is a significant performance improvement from AW to UNG (40% for CBRs and 47% for NCBRs). This is because using all unigrams to classify the bug reports leads to non-critical words having undue importance and thus lowers the accuracy. On the other hand, when using bigrams

TABLE VII
RESULTS OF FEATURE SELECTION

Program	*Classes*	*NaiveBayes*			*Logistic Regression*			*Decision Tree*		
		AW	UNG	BIG	AW	UNG	BIG	AW	UNG	BIG
Mozilla	Conf.	0.515	0.801	0.787	0.598	0.704	0.775	0.769	0.706	0.832
	Non-config.	0.511	0.932	0.811	0.686	0.792	0.719	0.711	0.698	0.782
Apache	Conf.	0.606	0.842	0.889	0.574	0.741	0.574	0.766	0.858	0.940
	Non-config.	0.592	0.877	0.891	0.523	0.851	0.623	0.702	0.779	0.782
MySQL	Conf.	0.530	0.667	0.607	0.654	0.740	0.701	0.708	0.649	0.679
	Non-config.	0.535	0.609	0.385	0.674	0.625	0.718	0.621	0.562	0.615
Average	Conf.	0.550	0.770	0.761	0.608	0.728	0.683	0.748	0.738	0.817
	Non-config.	0.546	0.803	0.696	0.628	0.756	0.687	0.678	0.680	0.726

as features (BIG), the performance improvement over AW is not as high as when using high information unigrams only (38.3% for CBRs and 15% for NCBRs).

In Apache, we also see performance improvement over AW when using UNG (19.7% for CBRs and 20.3% for NCBRs). However, the improvement is not as high as for Mozilla. This is due to the different styles of bug reports in Mozilla and MySQL. The terms used to describe bugs in Mozilla vary greatly from report to report; that is not the case in MySQL.

In MySQL, UNG did not perform better than AW. One possible reason is that there is too much extraneous information in these bug reports, such as the execution error results. While these execution error results can assist a developer with debugging, they contain many repetitive terms that detract from accurately identifying the bug report type. They become high information words to the classifier due to their frequency. To address this problem, we could extract frequently occurring terms and check if they are "noise" terms (i.e., error results in this case) and exclude those counts. As part of future work, we will leverage the work in [43] to eliminate noise terms.

The other likely reason that UNG did not outperform AW is that the high information terms do not occur predominantly in just one class (configuration OR non-configuration). For example, the words "set" and "value" appear in the first 100 high information words produced by Chi-Square. They are not used by the classifiers as key features for identifying configuration bug reports though (or are so far down the list of features that they are not used). These two words make it into the 100 high information words because they are frequent in some reports, making their occurrence count high. It is possible to address this in the future as was mentioned above; basically identifying additional stopwords or "noise" words by using prior work [43].

From this, we gained a number of insights. First, we learned that using informative unigrams plus bigrams generally increases performance compared to informative words alone, but not by much. In the case of the Apache subject using NaiveBayes, the performance is not increased as much. By balancing performance gain and required processing, we can choose to use UNG and BIG or can use UNG in isolation. Second, it is clear that bug reports may contain extraneous information and need to be "cleaned" or "condensed" to their essence. For future work, it is possible that prior work by Murphy et al. [37] can be used to distill bug reports to their essential terms. Third, it may be necessary to increase the number of bug reports in the training set in order to ensure that they are representative of real world occurrences and ratios of CBRs and NCBRs.

B. Numbers of Features

By default, CoLUA used the top 100 features in terms of their information gain scores, with the default ratio of unigrams to bigrams of 8:2. To further investigate whether using different numbers of selected features affect CoLUA's performance, we vary the number of selected features in the range [30, 50, 70, 90, 110, 130, 150, 170, 190]. We used the Decision Tree classifier because it achieved the best average performance among all three classifiers. Figure 4 plots the F-measure values computed for CBRs and NCBRs in all three subjects across different numbers of features. The results show that while the optimum number of features varies across different subjects, when we select more than 110 features, the performance of CoLUA is stable.

C. Ratios of Unigrams to Bigrams

We next investigate how varying the ratio of unigrams to bigrams in the selected 100 features can affect the performance of CoLUA. Again, we used the Decision Tree classifier. Figure 5 plots the F-measure values computed for CBRs and NCBRs in all three subjects across different portions of unigrams (1-9). As the figure shows, the performance does not make much difference for different ratios. The only exception occurred for Apache, where 7:3 is the optimum ratio.

D. Best Features

In addition to generating a model that can identify configuration bug reports, we are also interested in finding discriminative features that could help in distinguishing CBRs and NCBRs. The top 10 features are displayed in Table VIII, sorted by their information gain scores.

As the table shows, when the information gain score is high, it has more power to discriminate CBRs from NCBRs. For example, the keyword "Configuration" has been used by existing work [5] to identify configuration bug reports; it appears in the top 10 most discriminative features.

In contrast to our finding on the usefulness of high information gain score, not all important terms occur in the list. In NLTK, once a word is selected as a feature, it is treated

TABLE VIII
THE SUBSET OF THE MOST INFORMATIVE WORDS FOR MOZILLA, APACHE, AND MYSQL BUG REPORTS THAT ARE USED BY CLASSIFIERS

Mozilla		*Apache*		*MySQL*	
word	*score*	*word*	*score*	*word*	*score*
crash	8375.5	Configuration	542.4	option	253.4
build	1675.2	module	200.3	global	212.7
talkback	736.3	conf	196.4	configuration	208.1
reproducible	676.0	directive	175.8	cnf	171.2
identifier	553.7	enable	170.2	usr	136.2
option	442.9	src	112.3	ref	93.4
agent	376.5	xindice	99.0	connector	81.9
preference	357.3	ssl	95.0	socket	80.0
code	272.8	configure	45.8	sock	68.2

equally with other features, regardless of how often it appears in reports. It is up to the classifier to decide which of the selected features are important for determining if a bug report is configuration-related or not. For example, though they have high information gain scores, in the training bug reports the words “set” and “value” do not appear enough in bug reports to merit being treated as important features, so they do not appear at all in Table VIII. On the other hand, even though the word “preference” is not high on the list of information gain scores for Mozilla, it becomes high on the list of features selected by classifiers as shown in Table VIII. This is because “preference” appears in the majority of configuration bug reports. Thus, even though there are not many words that we consider as being very informative for Mozilla in Table VIII, the result for Mozilla is good.

VII. RELATED WORK

There has been a great deal of work on configuration-aware techniques [7], [36], [47], [49]. For example, Yin et al. [47] study a number of configuration bugs to understand the configuration errors in commercial and open source systems. Rabkin et al. [36] propose a static analysis technique to extract configuration options from Java code. There has been a large body of work in the testing community that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [35], [46]. Zhang et al. [49] propose a technique to diagnose crashing and non-crashing errors related to software misconfigurations. However, none of this work considers classifying configuration bug reports or extracting configurations from the bug reports. Our work is orthogonal to the above work though.

There has been some research on mining bug repositories to classify and predict specific fault types. For example, Padberg et al. [14] leverage statistical and machine learning techniques to label bug reports related to concurrency bugs. Gegick et al. [15] classify bug reports as either security- or non-security-related. However, these techniques neither classify configuration bug reports nor identify concrete bug sources. Xia et al. [44] use text mining to categorize configuration bug reports related to system settings and compatibilities, but their technique does not target configuration bugs due to misuse of configuration options. In contrast, CoLUA first predicts whether a configuration bug report is related to the incorrect settings of configuration options and then extracts concrete configuration options.

There has been considerable work on using natural language and information retrieval techniques to improve code documentation and understanding [13], [21], [22] and to create code traceability links [2], [12], [28], [33]. While our work applies some of these same basic techniques, such as tokenization, lemmatization, vector space model with term frequency-inverse document frequency weighting [8], the prior art has not applied these techniques to configuration bug reports and has not considered or extracted configuration options. Machine learning has been used recently to identify traceability links and to categorize or classify requirements [10], [20]. Weka and other machine learning frameworks have been leveraged for this work, just as with our work. In contrast, our work studies the application of such classifiers to configuration bug reports. Also, there is no prior art utilizing machine learning, natural language processing, and IR to discover the configuration options related to a predicted configuration bug report, to our knowledge. CoLUA is unique in that it works on the bug reports without access to other artifacts. The return result is the configuration options that are used within the configuration database.

Li et al. [27] collect security bug reports from Mozilla and Apache and use a natural language model to identify the root causes of the security bugs. The results provide guidance on what types of tools and techniques security engineers should use to address security bugs. Podgurski et al. [34] use a clustering approach for classifying bug reports to prioritize and identify the root causes of bugs. Their techniques, however, do not deal with configuration bug reports.

VIII. CONCLUSION

In this paper, we presented CoLUA, an automated approach for classifying configuration bug reports and extracting configuration options. CoLUA involves two steps. The first step trains classification models on the labeled bug reports to predict a given unlabeled bug report as being either a configuration or non-configuration bug report. The second step employs natural language processing and information retrieval to extract configuration options from the identified configuration bug reports. We applied CoLUA on 900 bug reports from three open source projects. The results show that CoLUA discriminates configuration bug reports from non-configuration bug reports with high accuracy and that it is effective at extracting configuration options. In the future, we will improve the performance of CoLUA by leveraging techniques to eliminate noise terms. In addition, we will perform more extensive experiments.

IX. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1464032, and CCF-1511117.

References

[1] *Getting Started With SAS 9.1 Text Miner*. SAS Institute, Incorporated, 2004.

[2] N. Ali, W. Wu, G. Antoniol, M. Di Penta, Y. G. Guhneuc, and J. H. Hayes. Moms: Multi-objective miniaturization of software. In *International Conference on Software Maintenance*, pages 153–162, 2011.

[3] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2004.

[4] Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *International Conference on Software Engineering*, pages 298–308, 2009.

[5] F. A. Arshad, R. J. Krause, and S. Bagchi. Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss. In *International Symposium on Software Reliability Engineering*, pages 198–207, 2013.

[6] B Ashok, Joseph Joy, Hongkang Liang, Sriram K Rajamani, Gopal Srinivasa, and Vipindeep Vangala. Debugadvisor: a recommender system for debugging. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 373–382, 2009.

[7] Mona Attariyan, MIchael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2012.

[8] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[9] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. pages 121–130, 2009.

[10] Jane Cleland-Huang, Adam Czauderna, Marek Gibiec, and John Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In *International Conference on Software Engineering - Volume 1*, pages 155–164, 2010.

[11] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, pages 342–351, 2005.

[12] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. Can better identifier splitting techniques help feature location? In *International Conference on Program Comprehension*, pages 11–20, 2011.

[13] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *International Working Conference on Mining Software Repositories*, pages 71–80, 2009.

[14] M. Blersch F. Padberg, P. Pfaffe. On mining concurrency defect-related reports from bug repositories. In *International Workshop on Mining Unstructured Data*, 10 2013.

[15] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *InternationalWorking Conference on Mining Software Repositories*, pages 11–20, 2010.

[16] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *International Conference on Software Engineering*, pages 789–800, 2015.

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, November 2009.

[18] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *Special Interest Group on Knowledge Discovery and Data Mining Explorations Newsletter*, 11(1):10–18, November 2009.

[19] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[20] Jane Huffman Hayes, Wenbin Li, and Mona Rahimi. Weka meets tracelab: Toward convenient classification: Machine learning for requirements engineering problems: A position paper. In *International Workshop on Artificial Intelligence for Requirements Engineering*, pages 9–12, 2014.

[21] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. Amap: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *International Working Conference on Mining Software Repositories*, pages 79–88, 2008.

[22] Matthew J. Howard, Samir Gupta, Lori Pollock, and K. Vijay-Shanker. Automatically mining software-based, semantically-similar words from comment-code mappings. In *International Working Conference on Mining Software Repositories*, pages 377–386, 2013.

[23] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *International Conference on Software Engineering (ICSE Companion)*, pages 215–224, 2014.

[24] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–88, 2012.

[25] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *International Conference on Automated Software Engineering*, pages 273–282, 2005.

[26] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the ACM SIGSOFT Symposium and the European Conference on Foundations of Software Engineering*, pages 311–321, 2011.

[27] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33, 2006.

[28] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), September 2007.

[29] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

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

[31] Adrian Nistor, Tian Jiang, and Lin Tan. Discovering, reporting, and fixing performance bugs. In *International Working Conference on Mining Software Repositories*, pages 237–246, 2013.

[32] Frank Padberg, Philip Pfaffe, and Martin Blersch. On mining concurrency defect-related reports from bug repositories.

[33] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. When and how using structural information to improve ir-based traceability recovery. In *European Conference on Software Maintenance and Reengineering*, pages 199–208, 2013.

[34] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *International Conference on Software Engineering*, pages 465–475, 2003.

[35] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis*, pages 75–86, 2008.

[36] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *International Conference on Software Engineering*, pages 131–140, 2011.

[37] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014.

[38] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computer Survey*, 34(1):1–47, 2002.

[39] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262, 2011.

[40] Sandeep Tata and Jignesh M Patel. Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM Sigmod Record*, 36(2):7–12.

[41] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, 2015.

[42] K. Wiklund, D. Sundmark, S. Eldh, and K. Lundvist. Impediments for automated testing – an empirical analysis of a user support discussion board. In *International Conference on Software Testing, Verification and Validation*, pages 113–122, 2014.

[43] Hans Friedrich Witschel. Estimation of global term weights for distributed and ubiquitous ir. In *The Workshop on Ubiquitous Knowledge Discovery for Users*, 2006.

[44] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou. Automated configuration bug report prediction using text mining. In *Computer Software and Applications Conference*, pages 107–116, 2014.

[45] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning*, pages 412–420, 1997.

[46] Cemal Yilmaz, Myra B. Cohen, and Adam Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, 29(4), July 2004.

[47] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *ACM symposium on Operating Systems Principles*, pages 159–172, 2011.

[48] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *International Working Conference on Mining Software Repositories*, pages 199–208, 2012.

[49] Sai Zhang and Michael D. Ernst. Automated diagnosis of software configuration errors. In *International Conference on Software Engineering*, pages 312–321, 2013.

[50] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. In *International Conference on Software Maintenance and Evolution*, pages 311–320, 2014.