

Automatic Defect Categorization

Ferdian Thung, David Lo, and Lingxiao Jiang

School of Information Systems

Singapore Management University, Singapore

{ferdianthung,davidlo,lxjiang}@smu.edu.sg

Abstract—Defects are prevalent in software systems. In order to understand defects better, industry practitioners often categorize bugs into various types. One common kind of categorization is the IBM’s Orthogonal Defect Classification (ODC). ODC proposes various orthogonal classification of defects based on much information about the defects, such as the symptoms and semantics of the defects, the root cause analysis of the defects, and many more. With these category labels, developers can better perform post-mortem analysis to find out what the common characteristics of the defects that plague a particular software project are. Albeit the benefits of having these categories, for many software systems, these category labels are often missing. To address this problem, we propose a text mining solution that can categorize defects into various types by analyzing both texts from bug reports and code features from bug fixes.

To this end, we have manually analyzed the data about 500 defects from three software systems, and classified them according to ODC. In addition, we propose a classification-based approach that can automatically classify defects into three super-categories that are comprised of ODC categories: *control and data flow*, *structural*, and *non-functional*. Our empirical evaluation shows that the automatic classification approach is able to label defects with an average accuracy of 77.8% by using the SVM multiclass classification algorithm.

I. INTRODUCTION

Defects are common in software systems. There are various kinds of defects. Some kinds correspond to wrong control flows or data flows in a software system; others are related to structural defects in the linkages of various classes, attributes, and methods; yet others are non-functional defects. To ensure reliability of software systems, the management of defects is important. Understanding defect types and managing recurring defects can help to suggest corresponding mitigation actions, such as the deployment of an automated bug finding tool, additional testing, developer retraining, etc., to prevent such defects from recurring in the future. For example, IBM has proposed a defect classification scheme referred to as Orthogonal Defect Classification (ODC) [11], [12] which has been applied to various software systems in various industries [28], [47]. ODC contains many orthogonal classifications based on defect types, impact, and many more.

Despite the benefit of categorizing defects into types and performing post-mortem analysis on the defects, many defects are often *not* grouped into categories as such categorization potentially involves much manual effort and the hectic schedule of development teams may not have time budgeted for bug categorization. Thus, there is a need for an automated approach that could help developers in assigning category labels to defects during post-mortem analysis.

In this paper, we propose a classification-based approach that categorizes defects into three families: control and data flow, structural, and non-functional. Our goal is to automatically classify a defect into one of the three families according to the content of the bug report and the associated code changes made to fix the bug. Given a large set of data about known defects and their fixes, various textual features corresponding to stemmed non-stop words are extracted from the bug descriptions. Also, various code features corresponding to the counts of various program elements are extracted from the bug-fixing code. Based on these pieces of information, we train a discriminative model that can classify a defect into one of the three families. Since there are three classes, we use a multi-class classification algorithm to establish the discriminative model. We can then use the resultant discriminative model to realize our goal of automatically labeling defect types.

We have evaluated our solution on 500 defects collected from the JIRA repositories of three software systems. We manually assign defect types to these 500 defects and use them as ground truths. Our results show that our classification model is promising. Measured in terms of average accuracy, which is often used in other studies involving multi-class classification [16], [46], our model could achieve an average accuracy of 77.8%.

Our work enriches the existing studies that classify bug reports and change requests. Antoniol et al. classify change requests into defect reports or feature requests [2]. Menzies and Marcus, and Lamkanfi et al. classify the severity of a bug report [24], [31]. Huang et al. classify bug reports based on their impact [22]. Orthogonal to these studies, our work assigns defects into one of the three defect categories, namely control and data flow, structural, and non-functional.

The contributions of this work are as follows:

- 1) We propose a solution that automatically classifies defects into categories. These categories are used to tag defects and could be used for post-mortem analysis. We leverage both textual and code features, and multi-class classification to infer these categories.
- 2) We have performed an empirical evaluation of our automated defect classification approach. The results based on 500 manually labeled defects show that our model can achieve an average accuracy of 77.8%.

The structure of this paper is as follows. In Section II, we describe some preliminary materials on defect classification. We present our overall framework in Section III. We elaborate our pre-processing strategies in Section IV. We present the

TABLE I
DEFECT TYPES AND THEIR DESCRIPTIONS BASED ON ODC [1]

Defect Type	Family	Description
Algorithm/Method	Control and Data Flow	"Efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change ..."
Assignment/Initialization	Control and Data Flow	"Value(s) assigned incorrectly or not assigned at all ..."
Checking	Control and Data Flow	"Errors caused by missing or incorrect validation of parameters or data in conditional statements ..."
Timing/Serialization	Control and Data Flow	"Necessary serialization of shared resource was missing, the wrong resource was serialized, or the wrong serialization technique was employed ..."
Function/Class/Object	Structural	"The error should require a formal design change, as it affects significant capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure(s) ..."
Interface/O-O Messages	Structural	"Communication problems between modules, components, device drivers, objects, or functions ..."
Relationship	Structural	"Problems related to associations among procedures, data structures and objects. Such associations may be conditional ..."

list of textual and code features that we use to characterize a bug in Section V. We describe our model learning and label prediction steps in Section VI. We present the results of our empirical evaluation in Section VII. We discuss related work in Section VIII. Section IX concludes with future work.

II. DEFECT CLASSIFICATION

We describe preliminary materials on Orthogonal Defect Classification (ODC) and state our problem definition.

A. Orthogonal Defect Classification

Orthogonal defect classification (ODC) is a classification of defects proposed by Chillarege et al. at IBM and widely used in the industry [11], [12], [22], [28], [47]. Within ODC, there are various categories; each category groups defects based on a particular criterion. Various criteria are proposed, including defect types, impact, defect removal activities, triggers, targets, qualifiers, ages, sources, etc.

In this work, we focus on the criterion with defect types. ODC defines 7 defect types, including Assignment/Initialization, Checking, Algorithm/Method, Function/Class/Object, Timing/Serialization, Interface/O-O Messages, and Relationship. These defect types and their meaning, extracted from IBM's ODC website¹, are shown in Table I. Further, we group these defect types into two families: *control and data flow* and *structural*. Defect types Assignment/Initialization, Checking, Algorithm/Method, and Timing/Serialization belong to the control and data flow family, while defect types Function/Class/Object, Interface/O-O Messages, and Relationship belong to the structural family.

In addition to ODC defect types, we find that there are many entries in bug tracking systems *explicitly* marked as defects that do not affect system code. For example, many systems involve configuration files and defects could occur in these configuration files. We find that a sizable proportion of defects tracked in bug tracking systems belong to this kind. Furthermore, during software evolution, often developers also fix inconsistencies in program documentations, e.g., Javadocs, comments, etc. These fixes are also often tracked in bug

tracking systems in a way similar to fixes for code defects. Thus, we introduce an additional defect type family: *non-functional* defects. This defect family captures errors that do not affect the correct functioning of a software system or are even not in the code.

B. Problem Definition

Given the three defect family labels (*control and data flow*, *structural*, and *non-functional*), we aim to establish a discriminative model that can automatically assign a defect family label to a defect based on its bug reports containing texts and its fixes containing code changes.

In this study, we predict one of the 3 defect families rather than one of the 7 original ODC defect types. A multi-class classification problem gets more difficult as the number of classes (i.e., in our setting, possible defect labels) increases. Thus, in our first study, we start with 3 defect labels, with the goal of achieving reasonable average accuracy.

III. OVERALL FRAMEWORK

Our proposed approach is based on classification algorithms and illustrated in Figure 1. The classification framework consists of two phases: training and deployment.

In the training phase, we take a set of defects with known defect family labels as input. Each defect is also associated with its textual bug reports and bug fixing code. The output of the training phase is a discriminative model that is able to differentiate the 3 defect families: *data and control flow*, *structural*, and *non-functional*.

In the testing phase, given a defect associated with its textual reports and bug fixing changes but with no defect family labels, we apply the discriminative model generated from the training phase to automatically infer the defect family to which the defect belongs.

A. Training Phase

The training phase has 4 components: text preprocessor, code preprocessor, feature extractor, and model learner. These components would process defects with known defect family labels.

¹<http://www.research.ibm.com/softeng/ODC/DETODC.HTM>

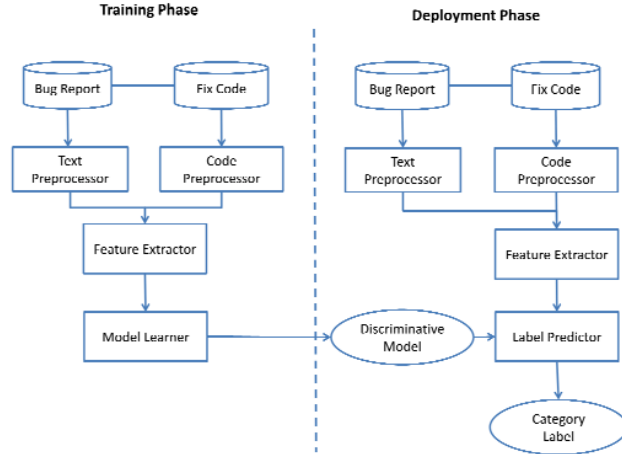


Fig. 1. Our Classification Framework

Each defect is represented by two pieces of data: a textual bug report and its bug fixing code. We need to convert these pieces of data into a representation suitable to be used as input to a machine learning algorithm. To that end, we perform two pre-processing steps: text pre-processing and code pre-processing, which correspond to the text preprocessor and code preprocessor components in Figure 1 and are explained further in Section IV.

Next step is feature extraction performed by the feature extractor component in Figure 1. A feature is an individual measurable attribute of a defect. The goal of the feature extraction step is to reduce the defects to some important, quantitative facets. In our setting, we can extract features from both texts and code. Textual features include the word tokens appearing in the pre-processed bug reports. Code features include the counts of various program elements in the pre-processed bug fixing code. Good feature engineering is essential in building an accurate classifier suitable for a particular domain. Section V explains in more details about our feature engineering.

After extracted, features are fed to the model learner component, which is a machine learning algorithm that can infer a discriminative model to differentiate each of the three families of defects. Part of Section VI explains the model learner used in our study.

When the discriminative model is established, it is passed to the deployment phase.

B. Deployment Phase

Inputs to the deployment phase are defects whose defect family labels are to be inferred by the discriminative model learned in the training phase. Similar to the training phase, each defect consists of a textual bug report and its bug fixing code. These inputs are processed by 4 components: text preprocessor, code preprocessor, feature extractor, and label predictor.

The first three components are the same as those performed in the training phase. The resultant features characterizing an input defect and the discriminative model learned in the training phase are taken as input to the label predictor component to infer the likely label of the defect. Part of Section VI also describes our label predictor in more details.

IV. TEXT & CODE PRE-PROCESSING

In this section, we present the text preprocessor and the code preprocessor in our framework.

A. Text Pre-Processing Strategies

We preprocess bug reports that are textual documents in three steps: tokenization, stop-word removal, and stemming.

1) *Tokenization*: The main objective of tokenization is to break a document into its constituent word tokens. Bug reports may contain contents that do not appear in average natural language texts, such as HTML tags, etc. We remove these HTML tags², and remove all numbers and punctuation marks appearing in the reports as they often have weak correlation with the meaning of the bug reports. We then extract the remaining word tokens.

2) *Stop-Word Removal*: Stop words are words that are used often and carry little meaning. We take a set of standard stop words from Ranks NL³, a search engine optimization company. These stop words are also removed from the extracted word tokens.

3) *Stemming*: This is to reduce a word to its root form. For example, the words “reading” and “reads” would be both reduced to “read”. We employ the well-known Porter stemmer⁴ to reduce a word to its representative root form.

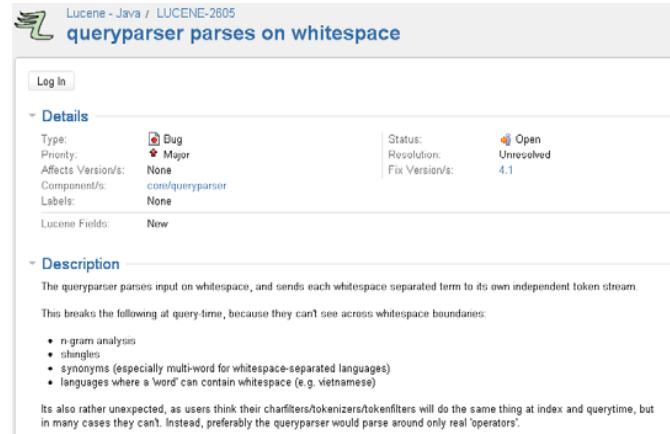


Fig. 2. Sample JIRA Bug Report For Lucene

²There are cases when HTML tags are meaningful for the reported bugs, which we leave for future improvements. Other non-English contents (e.g., code snippets, stack traces, etc.) in bug reports are currently treated as English texts too. In the future, we may use other tools, e.g., InfoZilla [9], to separate such contents from normal texts to improve the features used for classification.

³<http://www.ranks.nl/resources/stopwords.html>

⁴<http://tartarus.org/~martin/PorterStemmer/>

Various bug tracking systems may have different ways to store and present bug reports. Figure 2 shows a sample bug report from JIRA—a commercial bug/issue tracking system widely used for Apache projects⁵. There are various fields in a bug report (e.g., “Title” and “Description”) to which our text preprocessing strategies may be applied.

B. Code Pre-Processing Strategies

We also preprocess the code that fixes a bug. We collect these pieces of code from both bug tracking systems and commit logs. In the cases where Bugzilla is used as the bug tracking system, it may need special techniques to infer from the commit logs which pieces of code fix a bug (e.g., [45]). In the cases where JIRA is used as the bug tracking system, the bug fixing code has most likely been tagged in the system together with the bug reports since a unique keyword is used in both commit logs and bug reports to refer to a particular bug. As stated in our empirical evaluation (cf. Section VII-A), we focus on bugs tracked in JIRA.

A snapshot of JIRA interface showing a bug report linked to its fixing code is shown in Figure 3. We can see that, aside from the usual bug report information (title, description, severity, etc.), we have other information regarding the revision related to the bug (e.g., “Repository”, “Revision”, “Date”, etc., and the list of affected files). Using such information, we can extract the bug fixing code.

After bug fixing code is extracted, we convert it into abstract syntax trees (ASTs). There are two versions of the code, one before the bug fix is made, and the other after. Firstly, we identify the changed lines in the two versions of the code by performing a standard *diff*⁶ which would provide the code changes as a set of lines of code added and the set of lines of code deleted. Secondly, we parse both versions of the code before and after the fix to form two abstract syntax trees (ASTs), and then identify the nodes in both ASTs that correspond to the lines of code added and deleted, and prune the trees so that nodes not directly related to the added or deleted lines of code are removed.

V. FEATURE EXTRACTION

We extract two kinds of features for a defect: textual features from the bug reports, and code features from the code that fixes the bug.

A. Textual Features

As shown in the sample bug report in Figure 2, we can extract textual features from two fields: title and description. The title is a condensed representation of a bug report, and description provides additional information.

We first perform text preprocessing strategies as described in Section IV-A. Then, we extract three kinds of textual features from these two bug report fields:

$Text^T$	We take pre-processed word tokens from the <i>title</i> of a bug report as features. The value of each feature is the number of times the word tokens appear.
$Text^D$	We take pre-processed word tokens from the <i>description</i> of a bug report as features. For each feature, we also use its count as its feature value.
$Text^A$	We take pre-processed word tokens from <i>both</i> the title and description of a bug report as features. Again, for each feature, we also use its count as its feature value.

By default, we only use $Text^A$ in our `model learner` and `label predictor`, and we compare the effects of using $Text^T$ and $Text^D$ in Section VII-F.

B. Code Features

For each defect, we count various statistics from the AST trees as constructed according to the code preprocessing strategies (cf. Section IV-B), and use these statistics to form many of the code features. We list the complete set of code features that we use in Table II. The features capture the counts of various program elements in the added and deleted code. We also take the difference in the counts of program elements in the added and deleted code. The program elements that we consider include assignments, comments, character literals, looping structure, method invocations, throw statement, etc. We also count the numbers of lines that are added, deleted, and the difference of these two. The first 39 code features correspond to these AST node statistics. Note that we use Eclipse Java Development Tools⁷ to construct the ASTs, and comments are attached to the ASTs as special AST nodes although they are not considered in the main structure of ASTs. We also count the number of lines of code that are added and deleted, and the difference of the two ($F_{40} - F_{42}$).


TABLE II
CODE FEATURES USED FOR CLASSIFICATION

ID	Description
F_1	#Assignment Added
F_2	#Assignment Deleted
F_3	$ F_1 - F_2 $
$F_4 - F_6$	Similar to F_1 to F_3 for #BlockComment
$F_7 - F_9$	Similar to F_1 to F_3 for #CharacterLiteral
$F_{10} - F_{12}$	Similar to F_1 to F_3 for #EnhancedForStatement
$F_{13} - F_{15}$	Similar to F_1 to F_3 for #ExpressionStatement
$F_{16} - F_{18}$	Similar to F_1 to F_3 for #ForStatement
$F_{19} - F_{21}$	Similar to F_1 to F_3 for #IfStatement
$F_{22} - F_{24}$	Similar to F_1 to F_3 for #JavaDoc
$F_{25} - F_{27}$	Similar to F_1 to F_3 for #LineComment
$F_{28} - F_{30}$	Similar to F_1 to F_3 for #MethodInvocation
$F_{31} - F_{33}$	Similar to F_1 to F_3 for #ParenthesizedExpression
$F_{34} - F_{36}$	Similar to F_1 to F_3 for #ThrowStatement
$F_{37} - F_{39}$	Similar to F_1 to F_3 for #WhileStatement
$F_{40} - F_{42}$	Similar to F_1 to F_3 for #Line
F_{43}	DeletedAndAddedCodeSimilarity
F_{44}	hasJavaFile
F_{45}	hasXML/HTMLFile
F_{46}	hasXML/HTMLButNoJavaFile

⁵<http://wiki.apache.org/general/ApacheJira>

⁶<http://www.gnu.org/software/diffutils/>

⁷<http://www.eclipse.org/jdt/>


Lucene - Java / LUCENE-3913

HTMLStripCharFilter produces invalid final offset

Log In

Type: Bug

Priority: Major

Affects Version/s: None

Component/s: None

Labels: None

Status: Resolved

Resolution: Fixed

Fix Version/s: 3.6, 4.0

Lucene Fields: New, Patch Available

Description

Nightly build found this... I boiled it down to a small test case that doesn't require the big line file docs.

Repository	Revision	Date	User	Message
ASF	#1304904	Sat Mar 24 20:10:42 UTC 2012	sarowe	LUCENE-3913: Fix HTMLStripCharFilter invalid final offset for input containing
 Files Changed MODIFY /lucene/dev/branches/branch_3x/lucene/test-framework/src/java/org/apache/lucene/analysis/BaseTokenStreamTestCase.java MODIFY /lucene/dev/branches/branch_3x/lucene/test-framework/src/java/org/apache/lucene/util/TestUtil.java MODIFY /lucene/dev/branches/branch_3x/lucene/contrib/analyzers/common/src/test/org/apache/lucene/analysis/charfilter/HTMLStripCharFilterTest.java MODIFY /lucene/dev/branches/branch_3x/lucene/contrib/analyzers/common/src/java/org/apache/lucene/analysis/charfilter/HTMLStripCharFilter.java MODIFY /lucene/dev/branches/branch_3x/lucene/contrib/analyzers/common/src/java/org/apache/lucene/analysis/charfilter/HTMLStripCharFilter.flex

Fig. 3. Sample JIRA Bug Report with Links to Subversion Commits (including commit message and files changed).

We also include a derived feature *DeletedAndAddedCodeSimilarity* – F_{43} . This feature measures how similar the added code is from the deleted code. This is to address the limitation of diff which could only flag an entire line even if only one character is changed in the line. Also, diff sometimes flags more lines than it should. This feature also helps to differentiate between the situations when many real changes are done in the fixed code and the situations when most of the changed lines are only slightly modified from the original ones. To compute this feature, we tokenize the added and deleted lines by treating non alphabets and numbers as delimiters. We then use a bag of word tokens as the characterization of the added and deleted code. We represent these bags as term vectors in Vector Space Model (VSM) and compute the cosine similarity of these term vectors [29]. The resultant similarity score is used as the value of this derived feature *DeletedAndAddedCodeSimilarity*.

Besides extracting features from changed lines, we also extract features from added, modified, or deleted files. We investigate bug fixing changes in the repository for the existence of Java files that are added, modified, or deleted (F_{44}), and the existence of HTML and XML files that are added, modified, or deleted (F_{45}). We also extract a derived feature *hasXML/HTMLButNoJavaFile*; this is a boolean feature that indicates if there are XML and HTML files but no Java files that get added, deleted, or modified to fix the bug. We use this feature as we observe that many of the *non-functional* defects only involve changes to XML configuration files or the HTML files corresponding to manuals and JavaDocs.

As the values of these various features vary a lot, we normalize them such that each feature only takes a value between zero and one. To normalize these features we simply find the maximum (*maximum*) and minimum (*minimum*) values for

each feature and perform the following normalization for each feature value (*original*):

$$normalized = \frac{original - minimum}{maximum - minimum}$$

VI. MODEL-LEARNING & LABEL PREDICTION

This section describes the model learner and label predictor. The former is part of the training phase, while the latter is part of the deployment phase.

A. Model Learner

The goal of the Model learner is to learn a discriminative model that differentiates defects belonging to the three families: *control and data flow*, *structural*, and *non-functional*. It takes as input a multi-set of feature vectors from a training dataset. A training dataset contains a set of defects with a known defect family label, and each defect is represented as a feature vector. A feature vector is a set of features and their associated values. We use the set of features defined in Section V.

Each of the three defect families are represented in the training set. The model learner then learns some characteristics of each of the families based on the feature values of the defects belonging to the defect family in the training set. Our particular problem is known as a multi-class classification problem as there are more than two class labels: *control and data flow*, *structural*, and *non-functional*.

There are many machine learning algorithms that could perform multi-class classification. They include classification algorithms such as Support Vector Machine, decision tree, logistic, Naive Bayes, and many more. In this study, we mainly use Support Vector Machine (SVM) as it has been shown to be effective in many past studies in mining software

repository [15], [41]. In particular, we use the SVM^{multiclass} [13] implementation available at http://svmlight.joachims.org/svm_multiclass.html. In addition, we compare the performance of SVM with those of other classification algorithms in Section VII-E.

B. Label Predictor

The `Label predictor` takes as input the discriminative model learned by the `model learner` and a defect whose label is to be predicted. The defect is also represented by its feature vector, and the discriminative model would assign likelihoods of the defect to belong to each of the three defect families. The family with the highest likelihood would be outputted as the predicted label for the defect.

This step is a natural extension of `model learner`. Again, we use SVM^{multiclass} for our purpose by default.

VII. EMPIRICAL EVALUATION

In this section, we describe our datasets, research questions, answers to these questions, and threats to validity.

A. Datasets & Experiment Settings

We analyze defects from three software systems: Mahout [6], Lucene [5], and OpenNLP [7]. Mahout is a data mining library that provides algorithms to analyze large data by employing parallelization. It includes the implementations of various clustering, classification, frequent pattern mining, and collaborative filtering algorithms. It consists of 1,251 Java files and 175,295 lines of code (Version 0.6, May 2012). Lucene is an information retrieval library that supports various ways of retrieving relevant documents when given a query. Lucene has indexing and search capabilities. It consists of 2,564 Java files and 554,036 lines of code (Version 3.6, May 2012). OpenNLP is a natural language processing (NLP) library providing supports for various NLP tasks such as tokenization, segmentation, chunking, etc. It consists of 697 Java files and 78,224 lines of code.

We collect random defects from the JIRA repositories of the respective software systems: 200 from Mahout JIRA repository, 200 from Lucene JIRA repository, and 100 from OpenNLP JIRA repository. Thus, in total we have 500 randomly selected defects. These 500 entries in JIRA are *explicitly* tagged as defects rather than feature requests. The distribution of the 500 defects across the three defect families is shown in Table III. We note that about half of the defects belong to the *control & data flow* family. About 25% of the defects belong to the *structural* family and the others belong to the *non-functional* family.

TABLE III
DEFECT STATISTICS FROM MAHOUT, LUCENE, AND OPENNLP

Software	Defect Families			Total
	Control & Data	Structural	Non-Functional	
Mahout	120	46	34	200
Lucene	120	32	48	200
OpenNLP	46	32	22	100

To measure the effectiveness of our proposed approach, we make use of common effectiveness measures for multi-class classification. They are precision, recall, F-measure, accuracy, and AUC. The first four measures are defined based on the concepts of true positive (TP), false positive (FP), true negative (TN), and false negative (FN). Consider a given label l and a bug B with its ground truth label c . If a classifier outputs label o for B , there are 5 cases:

- 1) If $l = c$ and $c = o$, then B is a true positive for label l .
- 2) If $l = c$ and $c \neq o$, then B is a false negative for l .
- 3) If $l \neq c$ and $c = o$, then B is a true negative for l .
- 4) If $l \neq c$, $c \neq o$, and $o = l$, then B is a false positive for l .
- 5) If $l \neq c$, $c \neq o$, and $o \neq l$, then B is a true negative for l .

The above are the definitions of TP, FP, TN, and FN in a multi-class setting. Then, we can define precision, recall, F-measure, and accuracy for each label as follows:

$$precision = \frac{\#TP}{\#TP + \#FP}$$

$$recall = \frac{\#TP}{\#TP + \#FN}$$

$$F\text{-measure} = \frac{2 \times precision \times recall}{precision + recall}$$

$$accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN}$$

In the above equations, #TP, #TN, #FP, and #FN denote the numbers of true positives, true negatives, false positives, and false negatives respectively. F-measure is the harmonic mean of precision and recall.

AUC is the area under the Receiver Operating Characteristics (ROC) curve [19], [27]. A ROC curve for a label l plots the true positive rates for l (i.e., the fractions of true positives out of those predicted of label l) versus the false positive rates (i.e., the fractions of false positives out of those not predicted of label l). This curve could be created by first sorting the data points (i.e., defects whose labels are to be predicted) in descending order based on their likelihood of being assigned label l . The data points are then considered one by one and then the rates of true positives and false positives could be computed. These form the points in the ROC curve. AUC is the area under this ROC curve.

We compute all the above commonly used measures. In past studies, F-measure, accuracy, and AUC scores of 0.7 or above are often considered reasonable (e.g., [2], [26], [36]).

We use a ten-fold cross validation strategy [18]. We divide the dataset into 10 parts and perform ten iterations. At each iteration, we take 9 parts for training and 1 part for testing (deployment). Note that each data point (i.e., defect) would be used for testing once and only once. We report the averaged overall performance after ten iterations.

B. Research Questions

We are interested in answering these research questions:

- RQ1 How effective is our proposed approach in inferring the defect type labels?
- RQ2 What are the most effective features for defect type classification?
- RQ3 What are the performance of various classification algorithms?
- RQ4 What are the impact of the various text pre-processing strategies on the defect classification performance?
- RQ5 How is the performance of our proposed approach affected with varying amount of training data?

C. RQ1: Overall Effectiveness

The precision, recall, F-measure, accuracy, and AUC of our proposed approach are shown in Table IV. We also compute the weighted average across the three defect families. We take weighted average rather than mean as the proportion of defects that each defect family has is different. We note that the result for *control & data flow* and *non-functional* are reasonably good—their F-measures are 0.782 and 0.699 (out of 1) respectively. The result for *structural* is not that good though; its F-measure is only 0.436. However, the accuracy and AUC of *structural* are still reasonably good (0.788 and 0.699 respectively). Taking the weighted average, the result is reasonably good with F-measure at 0.692, accuracy at 0.778, and AUC at 0.779.

TABLE IV
EFFECTIVENESS OF OUR APPROACH

Class	Precision	Recall	F-Measure	Accuracy	AUC
Control & Data Flow	0.741	0.829	0.782	0.736	0.77
Non Functional	0.75	0.655	0.699	0.876	0.878
Structural	0.488	0.394	0.436	0.788	0.699
Weighted Average	0.69	0.7	0.692	0.778	0.779

We also show the confusion matrix in Table V. The confusion matrix shows how many times a defect of a particular family is classified as belonging to another family. We note that defects of type *control and data flow* are often misclassified as *structural*. Defects of type *non-functional* and *structural* are also often misclassified as *control and data flow*, which indicate that the features we use may not be discriminative enough and may be targets of improvements to our algorithm in the future.

TABLE V
CONFUSION MATRIX FOR OUR CLASSIFICATION RESULTS

Class	Classified As		
	Control & Data Flow	Non Functional	Structural
Control & Data Flow	237	15	34
Non Functional	29	72	9
Structural	54	9	41

D. RQ2: Most Discriminative Features

Considering the textual and code features, we have in total 4341 features. Fischer score is often used in the data mining community to infer the most discriminative features. We compute the Fisher score of every feature as follows:

$$F(j) = \frac{\sum_{class=1}^{\#class} (\bar{x}_j^{(class)} - \bar{x}_j)^2}{\sum_{class=1}^{\#class} (\frac{1}{n_{class}-1} \sum_{i=1}^{n_{class}} (x_{i,j}^{(class)} - \bar{x}_j^{(class)})^2)}$$

In the above equation, $F(j)$ is the Fisher score for the j^{th} feature. n_{class} is the numbers of data points with the *class* label. \bar{x}_j and $\bar{x}_j^{(class)}$ are the averages of the j^{th} feature of all and *class*-labeled data points respectively. Lastly, $x_{i,j}^{(class)}$ is the j^{th} feature of the i^{th} *class*-labeled data point.

Fisher score ranges from 0 to 1; features with Fisher score equals to 1 are the most discriminative ones while those with score 0 are the least discriminative ones. We show the top-10 most effective features in Table VI. We notice that the first three features are code features and they are quite discriminative. The rest are textual features and they are marginally discriminative. In the future, we may employ feature selection to reduce less relevant features in the classification models to avoid potential overfitting and improve performance.

TABLE VI
TOP 10 MOST EFFECTIVE FEATURES

Feature	Fisher Score
hasJavaFile	0.89
hasXML/HTMLButNoJavaFile	0.709
hasXML/HTMLFile	0.38
stem word "releas"	0.06
stem word "pom"	0.059
stem word "link"	0.051
stem word "except"	0.049
stem word "distribut"	0.044
stem word "packag"	0.042
stem word "thread"	0.041

E. RQ3: Varying Classification Algorithms

Our framework allows the use of various classification algorithms, other than SVM, for the model learner and label predictor. We have evaluated multiple classification algorithms, including C4.5 [33], Logistic [25], Naive Bayes [34], Naive Bayes Multinomial [35], and RBF Network [32] with their Weka [44] implementations.

The results are shown in Table VII. We notice that in terms of the weighted averages, SVM is better than all other classification algorithms. On the other hand, the other algorithms are able to achieve more than 0.6 score for F-measure (aka. F1), accuracy, and AUC as well.

F. RQ4: Varying Textual Pre-processing Strategies and Textual Features

As presented in Section IV-A, we employ both stop word removal and stemming. We investigate if these pre-processing strategies improve performance. We disable each of them and show the results for precision, recall, F-measure, accuracy, and AUC in Table VIII.

TABLE VII
VARYING CLASSIFICATION ALGORITHMS IN OUR FRAMEWORK

Algorithm	Class	Prec	Rec	F1	Acc	AUC
C4.5	Control & Data Flow	0.643	0.636	0.64	0.59	0.597
	Non Functional	0.663	0.627	0.645	0.848	0.797
	Structural	0.23	0.25	0.24	0.67	0.545
	Weighted Average	0.562	0.554	0.558	0.663	0.63
Logistic	Control & Data Flow	0.722	0.79	0.755	0.706	0.725
	Non Functional	0.657	0.591	0.622	0.842	0.782
	Structural	0.409	0.346	0.375	0.76	0.651
	Weighted Average	0.643	0.654	0.646	0.747	0.722
Naive Bayes	Control & Data Flow	0.753	0.598	0.667	0.658	0.716
	Non Functional	0.487	0.673	0.565	0.772	0.793
	Structural	0.38	0.442	0.409	0.734	0.645
	Weighted Average	0.617	0.582	0.591	0.699	0.718
Naive Bayes Multinomial	Control & Data Flow	0.77	0.678	0.721	0.7	0.757
	Non Functional	0.57	0.736	0.643	0.82	0.854
	Structural	0.406	0.413	0.41	0.752	0.676
	Weighted Average	0.65	0.636	0.639	0.737	0.761
RBF Network	Control & Data Flow	0.655	0.717	0.684	0.622	0.594
	Non Functional	0.461	0.373	0.412	0.766	0.662
	Structural	0.245	0.231	0.238	0.692	0.529
	Weighted Average	0.527	0.54	0.532	0.668	0.595
SVM Multiclass	Control & Data Flow	0.741	0.829	0.782	0.736	0.77
	Non Functional	0.75	0.655	0.699	0.876	0.878
	Structural	0.488	0.394	0.436	0.788	0.699
	Weighted Average	0.69	0.7	0.692	0.778	0.779

TABLE VIII
VARYING PREPROCESSING STRATEGIES IN OUR FRAMEWORK

Preprocessing	Class	Prec	Rec	F1	Acc	AUC
Stop Word Removal	Control & Data Flow	0.702	0.790	0.743	0.688	0.763
	Non Functional	0.733	0.6	0.66	0.864	0.87
	Structural	0.409	0.346	0.375	0.76	0.678
	Weighted Average	0.648	0.656	0.648	0.742	0.769
Stemming	Control & Data Flow	0.729	0.79	0.758	0.712	0.751
	Non Functional	0.718	0.673	0.695	0.87	0.872
	Structural	0.471	0.394	0.429	0.782	0.689
	Weighted Average	0.673	0.682	0.676	0.761	0.765
Both	Control & Data Flow	0.741	0.829	0.782	0.736	0.77
	Non Functional	0.75	0.655	0.699	0.876	0.878
	Structural	0.488	0.394	0.436	0.788	0.699
	Weighted Average	0.69	0.7	0.692	0.778	0.779

We notice that stop-word removal and stemming are important since disabling them would reduce the performance measures in F-measure, accuracy, and AUC.

We also investigate the effectiveness of various textual features. In Table IX, we notice that $Text^A$ gives an overall better performance compared to $Text^T$ and $Text^D$.

G. RQ5: Varying Training Data

We also vary the amount of training data. In k -fold cross validation, we use $\frac{k-1}{k} \times 100\%$ of the data for training, and $\frac{1}{k} \times 100\%$ for testing. This is a standard approach. We try to reduce the number of folds to reduce the proportion of data used for training. We show the result in Table X. We notice that reducing the amount of training data by reducing the number of folds does not affect the result much. In terms of F-measure, accuracy, and AUC, the largest reduction is the reduction in F-measure from 0.692 to 0.657 when we reduce the number of

TABLE IX
VARYING TEXT FEATURES USED FOR CLASSIFICATION

Feature	Class	Prec	Rec	F1	Acc	AUC
$Text^T$	Control & Data Flow	0.73	0.776	0.753	0.708	0.776
	Non Functional	0.655	0.738	0.898	0.868	
	Structural	0.351	0.375	0.363	0.726	0.68
	Weighted Average	0.677	0.666	0.668	0.754	0.776
$Text^D$	Control & Data Flow	0.738	0.776	0.756	0.714	0.746
	Non Functional	0.76	0.664	0.709	0.88	0.884
	Structural	0.418	0.414	0.416	0.758	0.672
	Weighted Average	0.676	0.676	0.675	0.76	0.761
$Text^A$	Control & Data Flow	0.741	0.829	0.782	0.736	0.77
	Non Functional	0.75	0.655	0.699	0.876	0.878
	Structural	0.488	0.394	0.436	0.788	0.699
	Weighted Average	0.69	0.7	0.692	0.778	0.779

TABLE X
VARYING k FOR k -FOLD CROSS VALIDATION

k	Class	Prec	Rec	F1	Acc	AUC
2	Control & Data Flow	0.730	0.815	0.77	0.722	0.76
	Non Functional	0.705	0.609	0.654	0.858	0.862
	Structural	0.384	0.317	0.347	0.752	0.674
	Weighted Average	0.653	0.666	0.657	0.758	0.764
3	Control & Data Flow	0.73	0.815	0.77	0.722	0.771
	Non Functional	0.777	0.664	0.716	0.884	0.889
	Structural	0.425	0.356	0.387	0.766	0.693
	Weighted Average	0.677	0.686	0.679	0.767	0.782
4	Control & Data Flow	0.741	0.832	0.784	0.738	0.773
	Non Functional	0.707	0.636	0.67	0.862	0.841
	Structural	0.488	0.375	0.424	0.788	0.711
	Weighted Average	0.681	0.694	0.684	0.776	0.775
5	Control & Data Flow	0.716	0.811	0.761	0.708	0.757
	Non Functional	0.774	0.591	0.67	0.872	0.858
	Structural	0.446	0.394	0.418	0.772	0.692
	Weighted Average	0.672	0.676	0.67	0.757	0.766
6	Control & Data Flow	0.745	0.818	0.78	0.736	0.765
	Non Functional	0.742	0.655	0.696	0.874	0.878
	Structural	0.438	0.375	0.404	0.77	0.69
	Weighted Average	0.681	0.69	0.683	0.773	0.774
7	Control & Data Flow	0.714	0.822	0.764	0.71	0.77
	Non Functional	0.795	0.6	0.684	0.878	0.861
	Structural	0.432	0.365	0.396	0.768	0.69
	Weighted Average	0.673	0.678	0.67	0.759	0.774
8	Control & Data Flow	0.745	0.797	0.77	0.728	0.771
	Non Functional	0.737	0.636	0.683	0.87	0.854
	Structural	0.444	0.423	0.433	0.77	0.71
	Weighted Average	0.681	0.684	0.681	0.768	0.777
9	Control & Data Flow	0.734	0.801	0.766	0.72	0.777
	Non Functional	0.737	0.636	0.683	0.87	0.872
	Structural	0.462	0.413	0.437	0.778	0.712
	Weighted Average	0.678	0.684	0.679	0.765	0.784
10	Control & Data Flow	0.741	0.829	0.782	0.736	0.77
	Non Functional	0.75	0.655	0.699	0.876	0.878
	Structural	0.488	0.394	0.436	0.788	0.699
	Weighted Average	0.69	0.7	0.692	0.778	0.779

folds from 10 to 2. The results could also be because the bug report population used in this paper is homogeneous. In the future, we plan to investigate more defects and related reports to enhance the general applicability of our algorithm.

H. Threats to Validity

This subsection considers threats to construct validity, threats to internal validity, and threats to external validity.

Threats to construct validity refer to the appropriateness of our evaluation measures. We make use of five commonly used evaluation measures: precision, recall, F-measure, accuracy, and AUC, used before in past studies. Thus, we believe threats to construct validity are minimal.

Threats to internal validity refer to experimenter biases. We manually label the 500 bug reports. The labeling process is a subjective one and there might be errors. We have involved a PhD student who is not an author of this paper to double check the labels, and any discrepancy is resolved by a discussion.

Threats to external validity correspond to the generalizability of our results. We have only investigated 500 randomly selected defects from three software systems, which may not be representative of even just the bugs in the three systems. In the future, we plan to reduce this threat further by analyzing more defects from more software systems.

VIII. RELATED WORK

In this section, we discuss some related studies on the classification of bug reports, empirical studies of bug reports, and text mining in software engineering. The survey here is by no means complete.

A. Classification of Bugs & Changes

One line of research related to ours is retrieval of duplicate reports. When a new report comes, it returns a list of reports existing in the repository which are potentially similar to the new report. The intrinsic problem is how to measure the similarity between two reports. Runeson *et al.* take natural language text of bug reports and use *cosine*, *dice* and *jaccard* to measure the similarity of reports [37]. Wang *et al.* use not only texts also execution information to improve the retrieve performance [43]. Sun *et al.* propose a machine learning approach and extend BM25F to accurately retrieve duplicate reports [40], [41]. Our work is orthogonal to the above as we aim to classify the type of a defect.

Studies in [3], [14], [42], propose techniques to assign bug reports to the right developers. Menzies and Marcus propose a prediction model to automatically infer the severity of bug reports which achieves F-measures of 0.14 to 0.86 for various severity labels [31]. Hindle *et al.* propose a technique that automatically classifies large changes into several categories and achieve an accuracy of 13-70% for various classification strategies [20]. Ko and Myers investigate the differences between defect reports and feature requests based on the linguistic characteristics of summaries and descriptions in bug reports [23]. Huang *et al.* classify defects based on their impact by analyzing textual features obtained from defect reports [22]. Their approach classifies 403 defects from a software system into reliability, capability, integrity, usability, and requirements category labels achieving F-measures of 0.222, 0.885, 0.700, 0.629, and 0.393 respectively. Our work uses a classification algorithm to classify bugs into *control and*

data flow, *structural*, and *non-functional* based on ODC [11], [12] and both textual and code features of a bug.

B. Empirical Studies of Bug Reports

Researchers have also done empirical studies on bug repositories. Sandusky *et al.* investigate the nature, extent, and impact of bug report networks in one large F/OSS development community [38]. Anvik *et al.* empirically study the characteristics of bug repositories and show findings on the number of reports that a person submits and the proportion of different resolutions [4]. Hooimeijer and Weimer develop a descriptive model based on a statistical analysis of surface features of over 27,000 bug reports in open source projects, to predict bug report quality [21]. Bettenburg *et al.* survey developers of Eclipse, Mozilla, and Apache to study what makes a good bug report. A good bug report provides enough information to developers for debugging [8].

C. Text Mining for Software Engineering

Text mining has been widely employed to solve various software engineering problems. It includes work that supports program comprehension and recovers traceability links between software artifacts.

Haiduc *et al.* produce succinct and informative text to characterize software code [17]. Sridhara *et al.* detect source code corresponding to high level abstractions and describe them succinctly [39]. Marcus and Maletic use Latent Semantic Indexing to recover the traceability links [30]. Chen *et al.* combine multiple techniques including regular expression matching and clustering to recover traceability links [10].

IX. CONCLUSION AND FUTURE WORK

To better understand defects, defect classification schemes have been proposed in the industry. These defect classifications can give us insight on the frequency of different kinds of defects appearing in a software system. Appropriate mitigation action could then be taken based on the findings. Despite the benefit of defect classification, the defect classification process is a manual one and involves much manual labor. Due to this, many defects stored in various bug tracking systems are not assigned any category label.

To address this problem, in this paper, we propose an automated approach that categorizes defects into three families: *control and data flow*, *structural*, and *non-functional*. To realize this, we extract features from both bug reports and bug fixing code. Both simple and derived features are extracted to better discriminate one defect family from the others. The derived features are designed based on our domain knowledge on the characteristics of the three families of defects. These features are then used by a multi-class classification algorithm to train a discriminative model that predicts which family a defect belongs to. We have evaluated our approach on a dataset of 500 manually labeled defects from three software systems, Lucene, Mahout, and OpenNLP. Our results are promising; we could achieve a (weighted) average F-measure, accuracy, and AUC of 0.692, 0.778, and 0.779.

In the future, we plan to improve our algorithm to achieve better F-measure, accuracy, and AUC scores. For example, we may incorporate more features (e.g., code McCabe complexity metrics, finer-grained code features for both the code in the diff trunks and the code surrounding the diff in addition to the statement-level features in the diff, and comments in the code) to enhance the discriminative capability of the classification models. On the other hand, we may employ feature selection to reduce less relevant features to avoid overfitting classification models. We also want to develop better customized classification models that can help to classify defects into finer-grained categories and defect types in addition to the three super-categories used in this paper. In addition, we want to reduce the threats to external validity by evaluating our approach on more defects from more software systems.

Also, note that our current approach can only be used for post-mortem analysis since it requires code features from bug fixes. In the future, we also plan to investigate defect classification solely based on bug reports so that the approach may be used before bug fixes to help triage and prioritize bug fixing efforts, and even suggest appropriate bug fixing actions based on classified defect types.

ACKNOWLEDGEMENTS

We thank anonymous reviewers to provide many valuable comments to help improve the quality of the paper.

REFERENCES

- [1] <http://www.research.ibm.com/softeng/ODC/DETODC.HTM>.
- [2] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *CASCON*, 2008, pp. 23:304–23:318.
- [3] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *ICSE*, 2006.
- [4] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *ETX*, 2005, pp. 35–39.
- [5] Apache Software Foundation, "Apache Lucene," <http://lucene.apache.org/core/>.
- [6] —, "Apache Mahout: Scalable Machine Learning and Data Mining," <http://mahout.apache.org/>.
- [7] —, "Apache OpenNLP," <http://opennlp.apache.org/>.
- [8] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *FSE*, 2008, pp. 308–318.
- [9] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," in *MSR*, 2008, pp. 27–30.
- [10] X. Chen and J. C. Grundy, "Improving automated documentation to code traceability by combining retrieval techniques," in *ASE*, 2011, pp. 223–232.
- [11] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, "Orthogonal defect classification—a concept for in-process measurements," *IEEE TSE*, vol. 18, no. 11, pp. 943–956, nov 1992.
- [12] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve," in *ICSE*, 1991, pp. 246–255.
- [13] K. Crammer and Y. Singer, "On the algorithmic implementation of multiclass kernel-based vector machines," *Journal of Machine Learning Research*, vol. 2, pp. 265–292, 2001.
- [14] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," in *SEKE*, 2004, pp. 92–97.
- [15] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, May 2008.
- [16] M. Galar, A. Fernández, E. B. Tartas, H. B. Sola, and F. Herrera, "An overview of ensemble methods for binary classifiers in multi-class problems: Experimental study on one-vs-one and one-vs-all schemes," *Pattern Recognition*, vol. 44, no. 8, 2011.
- [17] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *ICSE* (2), 2010, pp. 223–226.
- [18] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. Morgan Kaufmann, 2011.
- [19] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.*, 2nd ed. Springer, 2009.
- [20] A. Hindle, D. M. Germán, M. W. Godfrey, and R. C. Holt, "Automatic classification of large changes into maintenance categories," in *ICPC*, 2009.
- [21] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *ASE*, 2007, pp. 34–43.
- [22] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "AutoODC: Automated generation of orthogonal defect classifications," in *ASE*, 2011, pp. 412–415.
- [23] A. Ko and B. Myers, "A linguistic analysis of how people describe software problems," in *IEEE Visual Languages and Human-Centric Computing*, 2006, pp. 127–134.
- [24] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, 2010, pp. 1–10.
- [25] S. le Cessie and J. C. van Houwelingen, "Ridge estimators in logistic regression," *Applied St.*, vol. 41, no. 1, pp. 191–201, 1992.
- [26] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Software Eng.*, vol. 34, no. 4, pp. 485–496, 2008.
- [27] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *KDD*, 2009, pp. 557–566.
- [28] R. R. Lutz and I. C. Mikulski, "Empirical analysis of safety-critical anomalies during operations," *IEEE TSE*, vol. 30, no. 3, pp. 172–180, 2004.
- [29] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.
- [30] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *ICSE*, 2003, pp. 125–137.
- [31] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008, pp. 346–355.
- [32] M. J. L. Orr, "Introduction to radial basis function networks," 1996.
- [33] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
- [34] J. D. Rennie, L. Shih, J. Teevan, and D. R. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *ICML*, 2003, pp. 616–623.
- [35] D. Romano and M. Pinzger, "A comparison of event models for naive bayes text classification," in *AAAI*, 1998, pp. 41–48.
- [36] —, "Using source code metrics to predict change-prone java interfaces," in *ICSM*, 2011, pp. 303–312.
- [37] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007, pp. 499–510.
- [38] R. J. Sandusky, L. Gasser, and G. Ripoché, "Bug report networks: Varieties, strategies, and impacts in a F/OSS development community," in *MSR*, 2004, pp. 80–84.
- [39] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *ICSE*, 2011, pp. 101–110.
- [40] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.
- [41] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010.
- [42] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging," in *ICSE*, 2011, pp. 884–887.
- [43] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008, pp. 461–470.
- [44] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd ed. Morgan Kaufmann, 2005.
- [45] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering links between bugs and changes," in *SIGSOFT FSE*, 2011, pp. 15–25.
- [46] L. Xie, Z.-H. Fu, W. Feng, and Y. Luo, "Pitch-density-based features and an svm binary tree approach for multi-class audio classification in broadcast news," *Multimedia Syst.*, vol. 17, no. 2, 2011.
- [47] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE TSE*, vol. 32, no. 4, pp. 240–253, 2006.