Active Semi-Supervised Defect Categorization

Ferdian Thung, Xuan-Bach D. Le, and David Lo
School of Information Systems
Singapore Management University, Singapore
{ferdiant.2013,dxb.le.2013,davidlo}@smu.edu.sg

***Abstract*—Defects are inseparable part of software development and evolution. To better comprehend problems affecting a software system, developers often store historical defects and these defects can be categorized into families. IBM proposes Orthogonal Defect Categorization (ODC) which include various classifications of defects based on a number of orthogonal dimensions (e.g., symptoms and semantics of defects, root causes of defects, etc.). To help developers categorize defects, several approaches that employ machine learning have been proposed in the literature. Unfortunately, these approaches often require developers to manually label a large number of defect examples. In practice, manually labelling a large number of examples is both time-consuming and labor-intensive. Thus, reducing the onerous burden of manual labelling while still being able to achieve good performance is crucial towards the adoption of such approaches. To deal with this challenge, in this work, we propose an active semi-supervised defect prediction approach. It is performed by actively selecting a small subset of diverse and informative defect examples to label (i.e., active learning), and by making use of both labeled and unlabeled defect examples in the prediction model learning process (i.e., semi-supervised learning). Using this principle, our approach is able to learn a good model while minimizing the manual labeling effort.**

To evaluate the effectiveness of our approach, we make use of a benchmark dataset that contains 500 defects from three software systems that have been manually labelled into several families based on ODC. We investigate our approach's ability in achieving good classification performance, measured in terms of weighted precision, recall, F-measure, and AUC, when only a small number of manually labelled defect examples are available. Our experiment results show that our active semi-supervised defect categorization approach is able to achieve a weighted precision, recall, F-measure, and AUC of 0.651, 0.669, 0.623, and 0.710, respectively, when only 50 defects are manually labelled. Furthermore, it outperforms an existing active multi-class classification algorithm, proposed in the machine learning community, by a substantial margin.

I. Introduction

Defects are prevalent in software systems. Managing and understanding defects are thus important to not only better maintain but also improve the reliability of software systems. To better manage and understand defects, one would first need to categorize the types of defects that appear in a system. For example, Orthogonal Defect Classification (ODC) [9], [10] is a defect classification scheme from IBM that has been widely used to manage defects in various software projects [24], [37]. By understanding the frequency and severity of each defect type appearing in a system, one can then plan the best course of action to minimize the future impact of defects. Some possible courses of actions are deploying automated bug finding tools, performing additional testing, and training developers especially for the more frequent and severe types of bugs that happened in the past.

Although categorizing defects presents obvious benefits for software understanding and maintenance, manually labelling the defects is a tedious and cumbersome task. Thus, several studies have been proposed a number of approaches which automate the defects categorization process [14], [33]. These approaches typically rely on supervised learning in which a portion of the defects are manually labelled, and these are input to a machine learning technique to learn a discriminative model, which is then used to automatically classify other unlabelled defects. Still, training a good discriminative model in this manner typically requires a large number of training data. It may not be practical to ask a developer to manually label a large number of defects due to time and cost constraints. Given this situation, an effective active learning and semi-supervised technique would be of tremendous value.

Active learning attempts to train a good classification model with minimal labelled data. Given a budget specifying how many defects we are willing to manually label, it then selects a subset of defects that are the most informative. Developers can then label this smaller subset of defects and the defects can be used to train a model to label other unlabelled defects with reasonable accuracy. Semi-supervised learning attempts to use not only labelled data but also unlabelled data in the model construction process. Active and semi-supervised learning can then be combined together to reduce the amount of training defect examples that need to be manually labelled while maximizing accuracy.

In this paper, we extend the defect categorization work by Thung et al. [33]. In that work, defects are categorized into three families: control and data flow, structural, and non-code[1]. These three families are based on IBM's Orthogonal Defect Classification (ODC). In their experiments, Thung et al.'s work requires 90% of the data to be available for training a model, and the model is then used to label the remaining 10% of the data. However, in practice, it is not practical to ask developers to manually label 90% of the defects, while only the labelling of 10% of the defects are automated. Our primary goal is to automatically label all unlabeled defects by leveraging active learning to select a small set of defect examples to be labelled. These labeled examples are then mixed with the remaining unlabelled defect examples and the resultant set is used to train a classification model using a semi-supervised learning

[1]Previously, this category was referred to as non-functional [33]

978-1-4673-8159-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPC.2015.15

strategy. The classification model is then used to classify each unlabelled defect into one of the three defects families.

Our proposed approach works in three steps. In the first step, we use a clustering algorithm to group defect examples. We then pick one representative defect example for each cluster. Our goal is to get a diverse yet small set of examples to manually label. These initial examples are then used to learn an initial classification model. In the second step, the initial classification model is used to pick additional examples that the model is most doubtful of (i.e., the model is uncertain on the defect labels (families) of the examples). We propose three different strategies for actively selecting examples to label one-at-a-time. In the final step, we learn a discriminative model that is able to differentiate defects of different families by making use of labelled and unlabelled defect examples. We follow a semi-supervised learning method, namely self-training [7], to use the unlabelled defect examples. Following self-training method, we learn another classification model using all labelled training data. We then take the top unlabelled defect examples that the model is most certain of their labels. These unlabelled examples are then treated as labelled data and used along with existing labelled data to train a final prediction model.

We have evaluated our solution on 500 defects collected from the JIRA repositories of three software systems. These 500 defects are manually labeled by Thung et al. [33]. Our experiment shows that the effectiveness of our proposed approach is promising. With just 50 labeled defect examples, our semi-supervised and active-learning based approach can achieve a weighted precision, recall, F-measure, and AUC of 0.651, 0.669, 0.623, and 0.710 respectively. Our proposed approach is an active and semi-supervised learning method for multi-class classification. To show the need for our approach(rather than using an off-the-shelf classifier), we have also compared our approach against a state-of-the-art multi-class classification algorithm that employs active learning and comes with an off-the-shelf implementation [15]. The experiment results show that our approach can outperform this baseline approach by a substantial margin.

The contributions of this work are as follows:

1) We propose an automated defect categorization solution that only requires a small set of defect examples to manually label. To build a good classification model from limited training examples, we propose an active and semi-supervised approach that is able to pick a diverse and informative set of defect examples to label and also make good use of unlabelled defect examples.
2) We have performed an evaluation of our active and semi-supervised approach. The results based on 500 labeled defects show that our model can achieve a weighted precision, recall, F-measure, and AUC of 0.651, 0.669, 0.623, and 0.710 respectively. We have compared our approach against a state-of-the-art classification algorithm that supports active learning and multi-class classification [15] and show that our approach outperforms it by a substantial margin.

The structure of this paper is as follows. In Section II, we describe some preliminary materials. We present our proposed approach in Section III. We present the results of our empirical evaluation in Section IV. We discuss related work in Section V. Section VI concludes with future work.

II. Preliminaries

In this section, we first describe preliminary materials on defect classification and automatic defect categorization. We then provide details of the k-means clustering algorithm which we use as part of our approach. Finally, we briefly state our problem definition.

A. Defect Classification

In this work, we focus on the defect families defined by Thung et al. [33]. There are three defect families: *control and data flow*, *structural*, and *non-code*. This classification is derived from IBM's Orthogonal Defect Classification (ODC) [1] which is widely used in the industry [9], [10], [14], [24], [37]. ODC provides a number of orthogonal ways to classify defects and we particularly focus on a categorization of defects based on their *defect type*.

Table I maps each of the three defect families to the corresponding ODC's defect types. First, *control and data flow* defect family includes defects related to Assignment/Initialization, Checking, Algorithm/Method, and Timing/Serialization. Second, *structural* defect family includes defects related to Function/Class/Object, Interface/O-O Messages, and Relationship. A more detailed explanation on each ODC defect is provided in IBM's website[2] and technical report [1]. Last, *non-code* defect type is outside the scope of ODC defect types, however Thung et al. encounter many of such defects when they label hundreds of bug reports from several projects, and thus they consider it too. This family includes all defects that do not appear in the source code, but instead in other parts of a software system, e.g., in configuration and documentation files.

Thung et al. consider the 3 defect families rather than 7 original ODC defect types since building a machine learning solution that accurately classifies defect families into 7 types are much harder than one that can accurately classifies defects into 3 families [33]. A multi-class classification problem gets much harder as the number of classes (in our case, defect types) increases. If the accuracy of a machine learning solution is too low, it is no longer useful. Thus, similar to the original work by Thung et al. we focus on these 3 defect families and aim to get a reasonable classification accuracy.

B. Automatic Defect Categorization

Our work builds upon Thung et al.'s work [33]. Thung et al.'s approach accepts as input a textual description of defect report and the code for fixing the defect. Since their goal is to automate *post-mortem* bug analysis, they assume that developers have fixed the defects and the bug fixing code is available. *Post-mortem* bug analysis is useful to find out

[2]http://researcher.watson.ibm.com/researcher/view_group.php?id=480

TABLE I
Defect Families with Their Types and Descriptions

Family	ODC Defect Type	Description
Control and Data Flow	Algorithm/Method	"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 ..." [1]
	Assignment/Initialization	"Value(s) assigned incorrectly or not assigned at all ..." [1]
	Checking	"Errors caused by missing or incorrect validation of parameters or data in conditional statements ..." [1]
	Timing/Serialization	"Necessary serialization of shared resource was missing, the wrong resource was serialized, or the wrong serialization technique was employed ..." [1]
Structural	Function/Class/Object	"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) ..." [1]
	Interface/O-O Messages	"Communication problems between modules, components, device drivers, objects, or functions ..." [1]
	Relationship	"Problems related to associations among procedures, data structures and objects. Such associations may be conditional ..." [1]
Non-Code	Code-unrelated defects	Wrong settings in a configuration file or wrong descriptions in documentation.

the kinds of defects that plague a software system. Such information is valuable in preventing and dealing with future bugs. Bug reports and bug fixing code are preprocessed before a classification algorithm can be employed to learn a discriminative model. There are two different pre-processing steps in their approach: text pre-processing and code pre-processing.

In the text pre-processing step, they perform the following steps:

1) *Tokenization.* This process breaks a document into word tokens. Before doing tokenization, HTML tags, numbers and punctuation marks that appear in defect reports are removed. The remaining word tokens are then extracted.
2) *Stop-Word Removal.* Stop words are words that appear very frequently in English text. Since they appear very frequently, they are of little use in discriminating a document from another and thus are often removed. A set of stop words from Ranks NL[3] is used and these words are removed from the extracted tokens.
3) *Stemming.* This process converts a word to its root form. For example, "reads" and "reading" are all converted to "read". The well-known Porter stemmer[4] is employed.

In the code pre-processing step, bug fixing code is extracted from the version control system. The code version before the fix is also extracted. The changed lines between these two versions are identified using standard *diff*[5]. Next, the abstract syntax trees (ASTs) of both versions are generated. The nodes that correspond to the changed lines marked by *diff* are identified. All nodes that are not related to the changed lines are then removed.

After code and text pre-processing steps are completed, the approach extracts some code and text features for learning the classification model. These code and text features are summarized in Table II.

[3]http://www.ranks.nl/resources/stopwords.html
[4]http://tartarus.org/~martin/PorterStemmer/
[5]http://www.gnu.org/software/diffutils/

TABLE II
Code and Text Features Used for Classification

ID	Description
$Code_1$	#Assignment Added
$Code_2$	#Assignment Deleted
$Code_3$	$\|Code_1 - Code_2\|$
$Code_4 - Code_{42}$	Similar to $Code_1$ to $Code_3$ for #BlockComment, #CharacterLiteral, #EnhancedForStatement, #ExpressionStatement, #ForStatement, #IfStatement, #JavaDoc, #LineComment, #MethodInvocation, #ParenthesizedExpression, #ThrowStatement, #WhileStatement, and #Line
$Code_{43}$	DeletedAndAddedCodeSimilarity
$Code_{44}$	hasJavaFile
$Code_{45}$	hasXML/HTMLFile
$Code_{46}$	hasXML/HTMLButNoJavaFile
$Text^T_1 - Text^T_i$	Each feature is a unique word token from the *title* of a defect report. The value is the number of word token occurrences. i is the number of unique work tokens in the *title* of bug report.
$Text^D_1 - Text^D_j$	Similar like $Text^T$ for the *description* of a defect report. j is the number of unique work tokens in the *description* of bug report.
$Text^A_1 - Text^A_k$	Similar like $Text^T$ for *both* the title and description of a defect report. k is the number of unique work tokens in both the *title* and *description* of bug report.

These features are extracted from a labelled training data (i.e., a set of defects whose family/class labels are known). It is then fed to the classification algorithm to create the discriminative model. The SVM^{multiclass} [11] implementation from http://svmlight.joachims.org/svm_multiclass.html is used as the classification algorithm. The same set of features can then be extracted from unlabelled data (i.e., a set of defects whose family/class labels are unknown) and the constructed model can then be applied for assigning defect category labels to the unlabelled data.

C. *k-means Clustering*

k-means clustering is an unsupervised learning technique that has been widely used and studied among clustering methods that are based on minimizing a formal objective function. k-means algorithm attempts to group a given data set containing a number of data points to a certain number of clusters (i.e., k clusters) so as to minimize the mean squared distance from each data point to its nearest center [16]. To achieve this goal, the initial step is to define k centroids, one for each cluster, randomly. Next, every data point belonging to the given data set is associated to the nearest centroid. Afterwards, the centroids are re-calibrated to form new k centroids and the whole process is repeated until the centroids get stabilized.

D. Problem Definition

Given a number of unlabelled defects and a training budget (e.g., a fixed number of defect examples), we incrementally select the best set of training data to manually label (into one of the three defect families) until the budget is reached to build a classification model. This model is then used to label the remaining unlabelled data. Our goal is two-fold: practicality and accuracy. In other words, we want to minimize the cost of labelling defects while still being able to build a model with as much discriminative capability as possible based on the given budget.

III. Proposed Approach

In this section, we present our active semi-supervised defect categorization approach. We name it *LeDEx* which stands for *Learning with Diverse and Extreme Examples*. Its framework is illustrated in Figure 1. It is divided into three phases: the initial sample selection phase, the active selection phase, and the semi-supervised learning phase. In the initial sample selection phase, our approach captures a *diverse* set of examples to manually label. In the latter two phases, our approach learn using *extreme* examples to generate a more effective prediction model.

In the initial sample selection phase, *Sample Selector* would select a subset of *Unlabelled Defect Set* and we refer to this subset as *Unlabelled Defect Sample*. Note that this selection is without replacement and thus it would reduce the total number of unlabelled defects in the *Unlabelled Defect Set*. The *Unlabelled Defect Sample* would then be output to an *Oracle*. The *Oracle* will label the defects to their correct categories and outputs the *Labelled Defect Sample*. In practice, this *Oracle* would be a developer that manually categorizes the small number of defects in the sample. This *Labelled Defect Sample* is then put into the *Labelled Defect Set*.

In the active selection phase, using the initial *Labelled Defect Set* from the initial sample selection phase, a *Classifier* learns a classification model. Then, *Defect Selector* takes as input the *Classifier* and the *Unlabelled Defect Set* and analyze them to decide which defect from the *Unlabelled Defect Set* should be labeled next – we refer to this defect as the *Unlabelled Selected Defect*. Note that this selection is also without replacement and thus would reduce the total number of unlabelled defects in the *Unlabelled Defect Set*. We can then send the *Unlabelled Selected Defect* to the *Oracle* for labelling. *Labelled Selected Defect* would then be output. This labelled defect would be added to the *Labelled Defect Set*. The steps

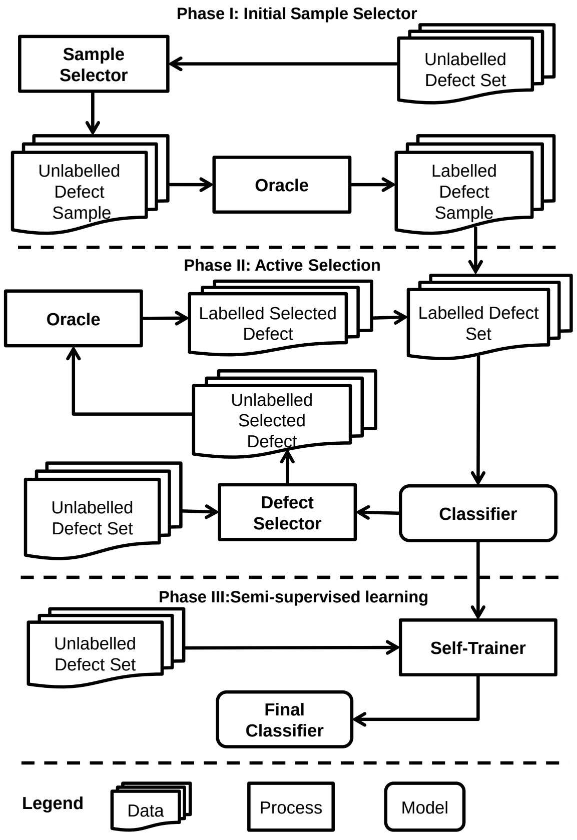

Fig. 1. Our Active Semi-Supervised Learning Framework

in the active selection phase would then be repeated until a training budget is reached (i.e., a maximum number of defects have been labelled).

In the semi-supervised learning phase, we can no longer ask the *Oracle* to label any defect example. However, the unlabelled defects can still provide valuable information that can help improve the classification model. In this phase, the resultant *Classifier* which is trained using the latest training data and the remaining *Unlabelled Defect Set* are input to *Self-Trainer* for further analysis. *Self-Trainer* would then refine the model by making use of the unlabelled defects. In a way, the machine is learning by itself (i.e., without the help of the *Oracle*). At the end, *Self-Trainer* outputs the *Final Classifier*.

We explain the details of the steps in the three phases of *LeDEx* in the following sections. *LeDEx* makes use of an underlying multi-class classification algorithm. In this work, we make use of $SVM^{multiclass}$ [11] as the underlying classification algorithm. We use of the implementation of $SVM^{multiclass}$ made publicly available from http://svmlight.joachims.org/svmmulticlass.html.

A. Phase I: Initial Sample Selection

In this phase, we pick some samples from *Unlabelled Defect Set* to be labeled by the *Oracle*. In this selection process, we want the sample defects to be as diverse as possible. In other words, we want the defects inside the sample to be as dissimilar as possible to one another. Intuitively, this should allow the classification algorithm to extract more information from the defects as compared to learning from defects that are similar to one another. Although learning to classify similar defects may be beneficial if those defects are of different labels, the model would have a very restricted knowledge about other defects outside those particular set of defects. There is also no guarantee that such benefit exists. In the worst case, those similar defects may actually belong to a one particular label and thus the classification algorithm would have no idea about the characteristics of the other labels. Thus, we would have a better chance to learn a model from a diverse set of defects.

In this work, we make use of a clustering algorithm to select a diverse set of defects. Clustering works by grouping data points that are similar to one another (i.e., according to a particular similarity measure) in the same cluster. Hence, data points in one cluster will naturally be dissimilar with data points in the other clusters. In our case, we use a clustering algorithm named k-Means (c.f., Section II) to cluster the unlabelled defects into k clusters. The parameter k is the number of training sample defects that would be used to construct the base model.

To use k-Means, we need to first define how distance between two defects is computed. To compute this distance, we first represent each defect by its feature vector which contains values of features described in Table II. Distance between two defects is then mapped to computing distance between their representative feature vectors. We compute the *Euclidean distance* between the two vectors and use it as the distance between a pair of defects. The Euclidean distance between two vectors $A = \langle a_1, \ldots, a_n \rangle$ and $B = \langle b_1, \ldots, b_n \rangle$ can be computed by the following formula: $\sqrt{\Sigma_{i=1}^{n} (a_i - b_i)^2}$. This Euclidean distance will be used by k-means to group the defects into k clusters.

After we have created the k clusters, we would pick only one defect from each cluster. In doing so, we intend to maximize the diversity of defects that we choose as the sample. For each cluster, our approach would pick a defect that is closest to the centroid. The centroid of a cluster of defects is a vector whose elements are the arithmetic means of the elements in the corresponding positions of the feature vectors of the defects in the cluster. A defect closest to the centroid would be able to represent the cluster better than any other defect inside the cluster. We measure the distance between a defect to the centroid by computing the Euclidean distance between the feature vector representation of the defect with the centroid. Mathematically, the centroid of a cluster can be defined as follows. Let us denote the i^{th} element of a feature vector F as $F[i]$. Consider a cluster of m defects represented by the following feature vectors $F_1, \ldots, F_m$, each of size n. The centroid of the cluster is a vector $\langle c_1, \ldots, c_n \rangle$, where c_i is defined as $\frac{\Sigma_{j=1}^{m} F_j[i]}{m}$.

B. Phase II: Active Selection

In this phase, some additional defects would be selected for manual labelling based on the information acquired from the *Unlabelled Defect Set* and the *Classifier* constructed from the initial sample.

First, we would apply the *Classifier* to the *Unlabelled Defect Set*. For each of the defect in *Unlabelled Defect Set*, *Classifier* would output a confidence score for each defect category. The defect category with the highest confidence score is the most probable defect category according to the *Classifier*. Next, we then adopt uncertainty based active selection method [21] adapted for the multi class setting. Intuitively, we want to pick defect that the current *Classifier* is the most uncertain about. We investigate the following three measures of uncertainty that we can use as a strategy for active selection:

1) *Margin Sampling [29].* This strategy selects a defect for which the *Classifier* has the smallest difference in confidence for two most probable defect categories. The defect is selected following this equation:

$$CD = \operatorname*{arg\,min}_{DF} P(\hat{DC_1}|DF) - P(\hat{DC_2}|DF)$$

where CD is the chosen defect, $\hat{DC_1}$ and $\hat{DC_2}$ are the first and second most probable defect category, DF represents features used to represent a defect, $P(\hat{DC_1}|DF)$ represents the probability that the defect with feature DF is a member of defect category $\hat{DC_1}$, and $P(\hat{DC_2}|DF)$ represents the probability that the defect with feature DF is a member of defect category $\hat{DC_2}$. This measure is designed based on the intuition that a large probability margin/difference between two most probable defect categories means that it is easy to differentiate between different defect categories. Therefore, we will pick a defect that is hard for the model to differentiate (i.e., defect with the smallest probability margin between the two most probable categories).

2) *Least Confidence [30].* This strategy selects a defect that the *Classifier* is least confident about. The defect is selected following this equation:

$$CD = \operatorname*{arg\,max}_{DF} 1 - P(\hat{DC}|DF)$$

where CD is the chosen defect, $\hat{DC}$ represents the most probable defect category, DF represents features used to represent the defect, and $P(\hat{DC}|DF)$ represents the probability that the defect with feature DF is the member of defect category DC.

3) *Shannon Entropy [31].* This strategy selects a defect with the highest entropy. Entropy is an information theoretic measure that quantifies the unpredictability of information. Thus, it can be used to measure uncertainty. The defect is selected following this equation:

$$CD = \operatorname*{arg\,max}_{DF} - \sum_{i=1}^{\#DC} P(DC_i|DF) \times log(P(DC_i|DF))$$

where CD is the chosen defect, DF represents features used to represent the defect, $\#DC$ is the number of defect categories and $P(DC_i|DF)$ is the probability that the defect with feature DF belongs to defect category DC_i.

Using one of the above uncertainty based active selection strategies, *Defect Selector* would select the most uncertain defect as *Unlabelled Selected Defect*. We choose to select just the most uncertain defect instead of the top m defects to maximize the benefit acquired from labelling, which is limited by the training budget. Consider the case where the most uncertain defect is labelled and then the classification model is rebuilt. The previously second most probable defect might be able to be labelled with a much higher certainty. As such, it would be better to label the most uncertain defect after the model is rebuild. *Defect Selector* would repeat this process until the training budget is reached.

C. Phase III: Semi-Supervised Learning

In this phase, we would make use of the remaining defects in the *Unlabelled Defect Set* to further refine our classification model. We use a semi-supervised learning method to combine information from *Labelled Defect Set* and *Unlabelled Defect Set*. In particular, we employ the self-training method [7]. Following this method, our approach first builds a *Classifier* from the labelled defects that are output by the previous phase. This *Classifier* is then used to increase the size of the training data by automatically labelling some of the unlabelled data, which it is most certain of, by itself. This newly labelled data is then used to retrain the *Classifier*.

Algorithm 1 shows the detail of the third phase of our approach. It accepts as input the set of labelled defects L, the set of unlabelled defects U, and the desired number of defects to be self labelled $size$. Initially, the algorithm would initialize $counter$ to 0 (Line 1). This $counter$ keeps track of the current number of self-labelled defects that have been added to the labelled defect set L. It also initializes a classifier C that is built from the set of labelled defects L (Line 2). It then perform the self-labelling process until the desired number of defects to be self-labelled (i.e., $size$) is reached (Lines 3- 20).

Inside the loop, the algorithm first makes sure that the number of defects that the classifier C will label would never exceed $size$ (Lines 4- 5). Next, for each unlabelled defect u in U and each label l (i.e., each defect family) in the set of labels $LABEL$, the classifier C would output the probability that u belong to the label l. This probability can be considered as classifier C's confidence that u belongs to the label l.

The algorithm then takes the maximum of these confidence scores (Line 7-10). Defects in U are then sorted based on their confidence scores $conf$ (Line 11). The top-k defects in U with the highest $conf$ is then removed from the unlabelled defect sets U (Lines 12-13). We refer to this subset as su, and for each member u in su, the classifier C assigns labels to each of them (Lines 14-16). The self-labelled defects in su are then added to the labeled defect set L (Line 17). Our approach then trains the classifier C using the updated labelled defect set L (Line 18). $counter$ is then increased according to the size of su (Line 19). The whole process repeats until $counter$ is equal to $size$. Finally, the algorithm returns the updated classifier C (Line 21).

In our work, we set the total number of defects to be self-labelled (i.e., $size$) to be half of the size of the remaining unlabeled defects. We pick this choice in order to maximize the use of the unlabelled defects and at the same time do not overestimate the capability of the unlabelled defects in helping us to refine the model.[6] We also set k equals to 1 to further minimize the risk of misclassification.

Algorithm 1: Self-Training Using Unlabelled Defects

```
Input : $L$ = set of labelled defects
        $U$ = set of unlabelled defects
        $k$ = number of defects to be self labelled
              in each iteration
        $size$ = total number of defects to be self labelled
Output: refined classifier $C$
1  Let $counter$ = 0;
2  Let $C$ = the classifier built from $L$;
3  while $counter < size$ do
4      if $size - counter < k$ then
5          $k = size - counter$;
6      end
7      $conf = [\,]$;
8      for $u \in U$ do
9          $conf[u] =$
           $max_{l \in LABELS}(C.labelsProbability(u, l))$;
10     end
11     Sort $U$ by $conf$;
12     Let $su$ = top-$k$ member of $U$;
13     Remove $su$ from $U$;
14     for $u \in su$ do
15         $su.label = C.classify(su)$;
16     end
17     Add $su$ to $L$;
18     Train $C$ using $L$;
19     $counter$ = $counter$ + $su.size$;
20 end
21 return $C$;
```

IV. Empirical Evaluation

In this section, we first describe our datasets and experiment settings. Next, we describe our research questions, followed by answers to these questions. Finally, we describe some threats to validity.

A. Datasets & Experiment Settings

We analyze defects from three software systems: Mahout [4], Lucene [3], and OpenNLP [5]. Mahout is a machine learning library that has a capability to analyze large data and performs task parallelization. It implements many popular learning algorithms, such as clustering, classification, frequent pattern mining, and collaborative filtering. As of version 0.6, it contains 1,251 Java files and 175,295 lines of code. Lucene is a search engine library that provides many options for retrieving documents that are relevant to a query. As of version 3.6, it contains 2,564 Java files and 554,036 lines of code. OpenNLP

[6]Note that some of the unlabelled defects may be wrongly self-labelled and they may reduce the effectiveness of the retrained model/classifier.

is a library for natural language processing with capabilities to support tasks such as tokenization, segmentation, chunking, etc. It contains 697 Java files and 78,224 lines of code.

We use the dataset that was used by Thung et al. [33]. It consists of 200 bug reports from Mahout, 200 bug reports from Lucene, and 100 bug reports from OpenNLP. In total, there are 500 randomly selected defects. We show the distribution of the 500 defects across the three defect families in Table III. About half of the defects belong to the *control & data flow* family, about 25% of them belong to the *structural* family and the remaining defects belong to the *non-code* family.

TABLE III
DEFECT STATISTICS FROM MAHOUT, LUCENE, AND OPENNLP

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

To measure the effectiveness of our proposed approach, we use 4 standard effectiveness measures for multi-class classification, namely precision, recall, F-measure, and AUC. The first three measures are defined based on the number of true positive (TP), false positive (FP), true negative (TN), and false negative (FN). Consider a given defect label l and a defect D with its actual defect category label c. If a classifier outputs defect category label o for D, we can consider 5 possible cases:

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

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

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

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

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

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 and it is a summary measure that combines precision and recall. It is often used to judge whether an increase in precision (recall) outweighs a decrease in recall (precision).

AUC is the area under the Receiver Operating Characteristics (ROC) curve [12], [22]. A ROC curve for a label l plots the true positive rates for l (i.e., the number of true positives out of the predictions for label l) versus the false positive rates (i.e., the number of true positives out of the predictions for label l). This curve could be created by considering the likelihood of each data point (i.e., defect whose label is to be predicted) to be assigned label l. A defect likelihood is then considered as the classification threshold; allowing the rates of true positives and false positives to be computed for each defect. Thus, each defect contributes a point in ROC curve. AUC is the area under this ROC curve.

In order to measure the overall performance of our approach, we compute the weighted average of each of the above evaluation metrics across the category labels.

$$WPrecision = \frac{1}{N} \times \sum_{label=1}^{\#label} n_{label} \times Precision_{label}$$

$$WRecall = \frac{1}{N} \times \sum_{label=1}^{\#label} n_{label} \times Recall_{label}$$

$$WF\text{-}measure = \frac{1}{N} \times \sum_{label=1}^{\#label} n_{label} \times F\text{-}measure_{label}$$

$$WAUC = \frac{1}{N} \times \sum_{label=1}^{\#label} n_{label} \times AUC_{label}$$

In the above equations, $WPrecision$, $WRecall$, $WF\text{-}measure$, and $WAUC$ are the weighted averages of precision, recall, F-measure, and AUC, respectively. N is the total number of defect reports in the test data, and n_{label} is the number of defect reports whose category is $label$.

For the experimental setting, we consider all 500 defects to be initially unlabelled. We then run our algorithm based on a training budget (i.e., the number of defects to be manually labeled). The remaining unlabelled data is used as the test data to measure the effectiveness of our approach. Due to some randomness in our approach (e.g., random seed for the k-means algorithm), we repeat our experiment ten times and report the average of each evaluation metric. By default, we use margin sampling as the active selection technique, set the training budget to 50, and set the initial sample size to be half of the training budget.

B. Research Questions

We are interested in answering these research questions:

Research Question 1. How effective is our proposed approach in selecting defects to be labelled?

In this research question, we want to measure the effectiveness of our approach in selecting the best defect examples given a training budget. We also want to investigate how effective our approach is when compared against another active learning approach. We browsed the machine learning literature and searched for a multi-class active learning classification approach that has been published in top machine learning conference/journal and comes with an off-the-shelf implementation. We finally find a multi class active learning approach proposed by Jain and Kapoor [15] that matches our criteria. Their algorithm is based on probabilistic k-Nearest Neighbor (pkNN) and has been tested and shown to be more effective than SVM for image data. Compared to ours, their algorithm does not have a clustering and a semi-supervised step. We use the original tool which is available at http://research.microsoft.com/en-us/downloads/16764958-77f9-4d36-8fdb-5131fb150f69/. For this RQ, we set the training budget to 50 examples.

Research Question 2. What is the contribution of each step of our approach?

There are three main steps in our approach. First, we pick an initial sample for building a base classification model by making use of a clustering algorithm. Second, we use active learning to select defect examples until the training budget is reached. Third, we perform semi-supervised learning to further refine the model by making use of the remaining unlabelled data. To measure the contribution of each step, we create two baselines: CLUSTER and CLUSTER+ACTIVE. For CLUSTER, we use the first step of our approach to select a sample defect examples to be labelled. These labelled examples are then used to build a classification model which is then used to assign labels to the remaining unlabelled data. For CLUSTER+ACTIVE, we use the first two steps of our approach to select examples to label. These labelled examples are then used to build a model which is used to assign labels to the remaining unlabelled data.

Research Question 3. What is the effect of training size to the effectiveness of our approach?

By default, we set the training budget to 50. In this research question, we want to investigate whether increasing training budget would increase the effectiveness of our approach. If it does, we also want to analyze the performance gain when we increase the training budget by a certain amount. Practically, it can give us some idea on how to trade off between performance and cost (in this case, the cost of labelling additional defect examples). If increasing size does not increase performance, we would also investigate why it is the case. Either way, we would have an idea on what to expect when changing training size and would be able to make better decision in choosing a reasonable training size depending on the situation.

Research Question 4. What is the performance of various active selection strategies?

By default, we use *Margin Sampling* as the active selection strategy. In our approach, there are two other possible strategies, namely *Least Confident* and *Shannon Entropy* active selection strategies. In this research question, we run our approach by using those two other active selection strategies. We report which of these three strategies leads to the best performance.

Research Question 5. What is the effect of using different number of self-labelled examples to the effectiveness of our approach?

By default, the number of self-labeled examples is set to be half of the number of remaining unlabelled defects which seems to be a reasonable compromise between performance gain and misclassification risk (see Section III-C). In this research question, we experiment with the following proportions of self-labelled defects: 20%, 40%, 60%, 80%, and 100% of the remaining unlabelled defects. We empirically investigate the effect of increasing the number of self-labelled examples.

C. RQ1: Overall Effectiveness

Table IV shows the effectiveness of our approach. We find that our approach can achieve a reasonable weighted precision, recall, F-measure, and AUC scores. The AUC of our approach is more than 0.7, and in many past studies, an AUC score above 0.7 is often considered reasonable [20], [27]. Note that we can achieve these reasonable scores by just using 50 labelled examples. Table IV also compares the effectiveness of our approach against that of Jain and Kapoor's approach. We note that although the performance of Jain and Kapoor's approach is better than random (AUC = 0.5), it is substantially worse than our approach. Our approach outperforms Jain and Kapoor's approach by 88.70%, 25.99%, 19.35%, and 36.34% in terms of weighted precision, recall, F-measure, and AUC respectively. The reason that our approach is better is likely due to our clustering and semi-supervised step. Also, although Jain and Kapoor's algorithm has been show to be performing well in image data (with a large number of classes), it had not been tested on defect data prior to this work. The algorithm might be adapted to the unique characteristics of image data that are absent from defect data.

TABLE IV
EFFECTIVENESS OF OUR APPROACH OVER JAIN AND KAPOOR'S APPROACH

Approach	**WPrec.**	**WRec.**	**WF-measure**	**WAUC**
Jain and Kapoor's	0.345	0.531	0.522	0.520
Ours	0.651	0.669	0.623	0.709
Improv.	88.70%	25.99%	19.35%	36.34%

D. RQ2: Contributions of Each Step of Our Approach

Table V compares the performance of our approach with the performance of two baselines: CLUSTERING and CLUSTERING+ACTIVE. CLUSTERING only uses the first step of our approach. CLUSTERING+ACTIVE uses the first and second steps of our approach. We use the same training budget setting for these two baselines and our entire approach (i.e., ALL STEPS). From the table, we can note that in terms of all evaluation metrics, the performance of our approach is better than CLUSTERING and CLUSTERING+ACTIVE. This shows that each step of our proposed approach contributes to its final effectiveness.

E. RQ3: Effect of Different Training Size

Table VI shows the weighted precision, recall, F-measure, and AUC of our approach using various training budget (i.e., number of labelled training data). From the table we can note

TABLE V
CONTRIBUTION OF EACH STEP IN OUR APPROACH

Step	WPrec.	WRec.	WF-measure	WAUC
CLUSTERING	0.622	0.606	0.583	0.681
CLUSTERING+ACTIVE	0.630	0.638	0.613	0.700
ALL STEPS	0.651	0.669	0.623	0.710

that in general the weighted F-measure and AUC increase as we increase the amount of labelled training data that our algorithm selects. However, even when we double the amount of training data (from 50 to 100), the average F-measure is only increased by 4.17% and the average AUC is only increased by 3.94%. This indicates that the first 50 samples that our approach selects are of good quality and they can be used to train a good model. Adding additional labelled samples improves this model but not by a large margin.

TABLE VI
VARYING TRAINING SIZE

Size	WPrec.	WRec.	WF-measure	WAUC
50	0.651	0.669	0.623	0.710
60	0.653	0.677	0.628	0.714
70	0.647	0.677	0.628	0.717
80	0.661	0.684	0.647	0.730
90	0.659	0.683	0.642	0.733
100	0.669	0.692	0.649	0.738

F. RQ4: Effect of Different Active Selection Strategy

Table VII shows the effectiveness of our approach using various active selection strategies. In terms of weighted F-measure and AUC, among the three strategies, we find Margin Sampling to be performing the best, followed by Least Confident, and Shannon Entropy. In terms of weighted F-measure, Margin Sampling outperforms Least Confident and Shannon Entropy by 1.63% and 2.98% respectively. In terms of weighted AUC, Margin Sampling outperforms Least Confident and Shannon Entropy by 1.43% and 2.01% respectively.

TABLE VII
VARYING ACTIVE SELECTION STRATEGY

Active Selection	WPrec.	WRec.	WF-measure	WAUC
Least Confident	0.631	0.666	0.613	0.700
Margin Sampling	0.651	0.669	0.623	0.710
Shannon Entropy	0.629	0.660	0.605	0.696

G. RQ5: Effect of Different Number of Self-Labelled Examples

Table VIII shows the effectiveness of our approach when using different proportion of self-labelled examples. Increasing the proportion of self-labelled examples from 20%-60% generally has a positive impact towards the effectiveness of our approach. Increasing the proportion from 60%-100% however generally has a negative impact towards the effectiveness of our approach. As suspected, using too little self-labelled examples will prevent us from maximizing performance gain. Moreover, adding too much would increase misclassification risk and therefore decrease the classification performance. Based on this finding, we believe that choosing the number of self-labelled examples to be half of the remaining unlabelled defects is indeed a reasonable choice.

TABLE VIII
VARYING PROPORTION OF SELF-LABELLED EXAMPLES

Proportion	WPrec.	WRec.	WF-measure	WAUC
20%	0.638	0.647	0.619	0.704
40%	0.649	0.661	0.622	0.708
60%	0.665	0.673	0.621	0.712
80%	0.667	0.673	0.607	0.707
100%	0.663	0.674	0.606	0.695

H. Threats to Validity

Threats to Construct Validity. These threats refer to the appropriateness of our evaluation measures. We make use of four commonly used evaluation measures for multi class classification: precision, recall, F-measure, and AUC. These measures have been used before in many past studies. Hence, we believe there are minimal threats to construct validity.

Threats to Internal Validity. These threats refer to experimenter biases. We use 500 bug reports from Thung et al.'s work [33]. They manually label the defects and double check the labels. If there are any discrepancies, they were resolved through discussion. Also, since our approach involves randomness, we perform our experiment multiple times and report the average performance over the multiple runs. Thus, we believe there are minimal threats to internal validity.

Threats to External Validity. These threats refer to the generalizability of our experimental results. We have only investigated 500 random defects from three software systems. These defects might not be representative enough. In the future, we plan to reduce this threat by experimenting on more defects from more software systems.

V. RELATED WORK

A. Classification of Issue Reports

Antoniol et al. propose a technique to automatically classify issue reports as either bug or enchancement [2]. Ko and Myers investigate the linguistic characteristics of summaries and descriptions in issue reports to differentiate between bug reports and feature requests [17]. Kochhar et al. consider the problem of mislabelled issue reports (i.e., an issue report labelled as bug report might not be a bug report at all) and propose to automatically reclassify them [18]. Our work complements the above studies. After bug reports are identified among issues that are submitted to a bug tracking system, our approach can categorize these bug reports into defect families.

Menzies and Marcus propose a prediction model which enables the severity of bug reports to be automatically inferred with F-measures of 0.14 to 0.86 for multiple severity labels [26]. Lamkanfi et al. propose a prediction model to classify severity of bug reports in open source systems, but consider only two severity labels: severe or not [19]. Tian et al. propose DRONE, a multi factor analysis technique to classify the priority of bug reports [34]. Their approach outperforms

Menzies and Marcus's approach substantially in terms of F-measure. While the above studies classify a bug report based on its importance, our work classifies a bug report based on its defect type. Both the importance of a bug and its defect type are important pieces of information that developers can potentially use for personnel training, test planning, and defect mitigation purposes.

A number of studies propose ways to classify if a bug report is a duplicate bug report or not [28], [32], [36]. Leveraging natural language text in bug reports, Runeson et al. retrieve textually similar bug reports by using $cosine$, $dice$ and $jaccard$ metrics to measure the similarity of reports [28]. To further improve the retrieval performance, Wang et al. exploit richer information available in bug reports, which include not only text but also execution traces [36]. Sun et al. extend BM25F, an effective similarity formula in the information retrieval community, for accurate retrieval of duplicate reports [32]. Our work is orthogonal to the above studies as we aim to classify types of defects instead of whether a bug report is a duplicate one or not.

By analyzing textual features obtained from bug reports, Huang et al. classify defects based on their impact [14]. Based on their approach, 403 defects from a software system can be classified into different category labels, including reliability, capability, integrity, usability, and requirement. For each of these classes, their prediction model achieves F-measures of 0.222, 0.885, 0.700, 0.629, and 0.393 respectively. More recently, Thung et al. automatically categorize bug reports based on their defect types into three families: *control and data flow*, *structural*, and *non-code* [33]. They employ code and text features and show that their approach can achieve an average F-measure and AUC of 0.692 and 0.779 respectively. Our work extends the prior work by Thung et al.. In the prior work, 90% of the defects are used to train an accurate discriminative model to label the remaining 10% of the defects. The cost of manually labelling 90% of the defects is often too high in practice. In this work, we build upon the prior work by proposing an active semi-supervised defect categorization solution that is able to work well with much less manual labelling effort.

B. Active Learning in Software Engineering

Several studies have proposed the usage of active learning for automating software engineering tasks. Bowring et al. propose an automatic approach for classifying program behaviour (i.e., execution traces) by leveraging Markov model and active learning through bootstrapping [6]. Their approach classifies execution traces as normal execution traces or failures. Lucia et al. proposes an approach to actively incorporate user feedback in ranking clone anomaly reports [23]. They show that their approach can improve the ordering of clone anomaly reports. Wang et al. propose a technique that can refine the results of Portfolio, a code search engine, by actively incorporating incremental user feedback [35]. They show that their proposed refinement approach can improve Portfolio's performance (i.e., in terms of NDCG) by up to 11.12%. Similar with our work, the above studies also use active learning. However, they work on binary classification and ranking problem while we work on multi-class classification problem. The tasks that they focus on are also different from ours.

C. Text Mining in Software Engineering

Various software engineering problems have been addressed by employing text mining. Marcus and Maletic use Latent Semantic Indexing, an information retrieval method, to recover the documentation to source code traceability links [25]. Chen et al. combine multiple techniques including regular expression matching and clustering to recover traceability links [8]. Hou and Mo use Naive Bayes to classify contents in Java Swing Forum into 8 to 17 categories [13]. In this work, we also analyze textual contents, however, we focus on a different problem, namely the categorization of bug reports into families.

VI. Conclusion and Future Work

In this study, we proposed a hybrid machine learning approach, that combines clustering, active learning and semi-supervised learning algorithms, to automatically categorize defects. Our primary goal is to minimize the number of training data that needs to be labelled while maximizing the accuracy of the trained classification model. Our approach is built on three main steps, including picking initial sample, actively selecting defect examples that are most informative for training classification model, and incrementally refining the trained model. Each step is done by employing each of the three learning algorithms accordingly. We evaluated our approach on 500 defects collected from JIRA repositories of three software systems. The evaluation results show that our proposed approach is promising. With just 50 labeled defect examples, our approach can achieve a weighted precision, recall, F-measure and AUC of 0.651, 0.669, 0.623, and 0.710 respectively. Furthermore, our approach outperforms a baseline approach by Jain and Kapoor [15], which is a state-of-the-art active-learning multi-class classification algorithm with a publicly available implementation, by a substantial margin.

As future work, we plan to further improve the precision, recall, F-measure and AUC of our approach. To do so, we plan to investigate the effectiveness of various clustering, active learning, and semi-supervised learning techniques when they are used as building blocks of our approach. We also plan to extend the features that we use to train the classification model, to help increase the discriminative capability of the model and experiment with more data to further evaluate the generalizability of our result. We also plan to perform an in-depth analysis to get an insight on the reason why our approach is less effective for some cases. Based on this analysis, we plan to design an appropriate extension to our approach.

Acknowledgement

We would like to thank Prateek Jain and Ashish Kapoor (Microsoft Research) for sharing the implementation of their active-learning multi-class classification algorithm [15].

References

[1] http://researcher.watson.ibm.com/researcher/files/us-pasanth/ODC-5-2.pdf.

[2] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. ACM, 2008, p. 23.

[3] Apache Software Foundation, "Apache Lucene," http://lucene.apache.org/core/.

[4] ——, "Apache Mahout: Scalable Machine Learning and Data Mining," http://mahout.apache.org/.

[5] ——, "Apache OpenNLP," http://opennlp.apache.org/.

[6] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 195–205.

[7] O. Chapelle, B. Schölkopf, and A. Zien, "Semi-supervised learning," 2006.

[8] X. Chen and J. C. Grundy, "Improving automated documentation to code traceability by combining retrieval techniques," in *ASE*, 2011, pp. 223–232.

[9] 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.

[10] R. Chillarege, W.-L. Kao, and R. G. Condit, "Defect type and its impact on the growth curve," in *ICSE*, 1991, pp. 246–255.

[11] 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.

[12] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.*, 2nd ed. Springer, 2009.

[13] D. Hou and L. Mo, "Content categorization of api discussions," in *29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 60–69.

[14] 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.

[15] P. Jain and A. Kapoor, "Active learning for large multi-class problems," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 762–769.

[16] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, no. 7, pp. 881–892, 2002.

[17] 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.

[18] P. S. Kochhar, F. Thung, and D. Lo, "Automatic fine-grained issue report reclassification," in *2014 19th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2014, pp. 126–135.

[19] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2010, pp. 1–10.

[20] 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.

[21] D. D. Lewis and J. Catlett, "Heterogeneous uncertainty sampling for supervised learning," in *Proceedings of the eleventh international conference on machine learning*, 1994, pp. 148–156.

[22] 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.

[23] Lucia, D. Lo, L. Jiang, and A. Budi, "Active refinement of clone anomaly reports," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 397–407.

[24] 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.

[25] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *ICSE*, 2003, pp. 125–137.

[26] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *ICSM*, 2008, pp. 346–355.

[27] D. Romano and M. Pinzger, "Using source code metrics to predict change-prone java interfaces," in *ICSM*, 2011, pp. 303–312.

[28] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE*, 2007, pp. 499–510.

[29] T. Scheffer, C. Decomain, and S. Wrobel, "Active hidden markov models for information extraction," in *Advances in Intelligent Data Analysis*. Springer, 2001, pp. 309–318.

[30] B. Settles and M. Craven, "An analysis of active learning strategies for sequence labeling tasks," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2008, pp. 1070–1079.

[31] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[32] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *ASE*, 2011.

[33] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *2012 19th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2012, pp. 205–214.

[34] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *2013 29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 200–209.

[35] S. Wang, D. Lo, and L. Jiang, "Active code search: incorporating user feedback to improve code search relevance," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 677–682.

[36] 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.

[37] 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.