Active Refinement of Clone Anomaly Reports

Lucia, David Lo, Lingxiao Jiang, and Aditya Budi
School of Information Systems
Singapore Management University
{*lucia.2009,davidlo,lxjiang,adityabudi*}*@smu.edu.sg*

***Abstract*—Software clones have been widely studied in the recent literature and shown useful for finding bugs because inconsistent changes among clones in a clone group may indicate potential bugs. However, many inconsistent clone groups are not real bugs (true positives). The excessive number of false positives could easily impede broad adoption of clone-based bug detection approaches.**

In this work, we aim to improve the usability of clone-based bug detection tools by increasing the rate of true positives found when a developer analyzes anomaly reports. Our idea is to control the number of anomaly reports a user can see at a time and actively incorporate incremental user feedback to continually refine the anomaly reports. Our system first presents top few anomaly reports from the list of reports generated by a tool in its default ordering. Users then either accept or reject each of the reports. Based on the feedback, our system automatically and iteratively refines a classification model for anomalies and re-sorts the rest of the reports. Our goal is to present the true positives to the users earlier than the default ordering. The rationale of the idea is based on our observation that false positives among the inconsistent clone groups could share common features (in terms of code structure, programming patterns, etc.), and these features can be learned from the incremental user feedback.

We evaluate our refinement process on three sets of clone-based anomaly reports from three large real programs: the Linux Kernel (C), Eclipse, and ArgoUML (Java), extracted by a clone-based anomaly detection tool. The results show that compared to the original ordering of bug reports, we can improve the rate of true positives found (i.e., true positives are found faster) by 11%, 87%, and 86% for Linux kernel, Eclipse, and ArgoUML, respectively.

I. Introduction

Code clones, or pieces of similar code, commonly occur in large software systems [1], [2] due to various reasons, which range from improper code reuse via the prevalent copy-and-paste practice, to the introduction of redundant code to improve runtime efficiency and/or reliability of systems. They have attracted many research interest and various studies on detecting code clones [2]–[5], tracking and managing code clones [6]–[8], and examining the harmfulness or usefulness of code clones [9]–[11].

One important use of code clones is their applicability in detecting bugs [11]–[16]. These *clone-based anomaly detection* tools look for *inconsistencies* among code clones in every clone group (*i.e.*, a group of code fragments similar to each other) and report them as anomalies (*i.e.*, potential bugs). For example, Li *et al.* [14] look for different identifier names among clones and check whether all names are changed consistently; Jiang *et al.* [12] look at syntactical structures of the code surrounding every clone, in addition to the identifier names in clones, and report differences as anomalies. Tens of true positives of diverse characteristics from large systems, such as the Linux kernel and Eclipse, have been found by these tools. Figure 1 shows a true positive from the Linux kernel: there is a missing null-check on the `tmp` variable in the code fragment 2. Such a detection is possible because most parts of the two code fragments (after the `if` condition) are detected as clones, and the code surrounding the clones (which are the variable declaration and the `if` condition) shows some structural differences (no `if` in code fragment 2).

However, the set of reported anomalies can be huge, containing hundreds or even thousands of reports. Among these anomalies, only a small proportion are true positives; others are *benign variations* among clones in a clone group, which are intended changes rather than mistakes. The process of verifying whether these anomalies are true or not can be painstaking and time-consuming. Developers tend to give up if many of the first set of anomaly reports that they check are false positives. For example, Jiang *et al.* [12] reported that among more than 800 reports generated by their tool for the Linux kernel, only 57 are true bugs or bad programming styles. Gabel *et al.* [15] applied more advanced filtering techniques based on textual similarity and sequence alignment on inconsistent clones detected from a large commercial code base. They reported that among 500 manually checked anomaly reports (out of 8103 in total), 149 may be true bugs and 109 may be code smells, while the rest is unsure. Hence, reducing the manual effort in locating true positives in clone-based anomaly reports remains an important task for wide adoption of such tools.

In this paper, we propose an active-learning and user-feedback directed approach to help alleviate the problem of false positives. The task is challenging as there are only a few true positives embedded in a mass of false positives. Our idea is to actively, iteratively incorporate user feedbacks to refine anomaly reports. Users are presented anomaly reports one by one; as a user labels the report as a false positive or true positive, our system *actively updates* the remaining set of anomaly reports. In so doing, we aim to make true positives appear earlier in the list

```
Code Fragment 1

File: linux-2.6.19/fs/sysfs/inode.c

219: struct dentry * dentry = sd->s_dentry;
220:
221: if (dentry) {
      /* the following parts are detected as clones */
222:        spin_lock(&dcache_lock);
223:        spin_lock(&dentry->d_lock);
224:        if (!(d_unhashed(dentry) && dentry->d_inode)) {
225:                    dget_locked(dentry);
226:                    __d_drop(dentry);
227:                    spin_unlock(&dentry->d_lock);
228:                    spin_unlock(&dcache_lock);
229:                    ......
```

```
Code Fragment 2

File: linux-2.6.19/drivers/infiniband/hw/ipath/ipath_fs.c

456: struct dentry *tmp;
457:
458: tmp = lookup_one_len(name, parent, strlen(name));
459:
460: spin_lock(&dcache_lock);
461: spin_lock(&tmp->d_lock);
462: if (!(d_unhashed(tmp) && tmp->d_inode)) {
463:         dget_locked(tmp);
464:         __d_drop(tmp);
465:         spin_unlock(&tmp->d_lock);
467:         spin_unlock(&dcache_lock);
468:         ......
```

Figure 1. A sample bug (missing null-check) revealed by contextual inconsistency among clones in a clone group from the Linux kernel – compare lines 221 & 224 in code fragment 1 with lines 459 & 462 in code fragment 2.

G# 1

```
Code Clone 1

File: linux-2.6.19/fs/nfsd/nfs3xdr.c

423: if (!(p = decode_fh(p, &args->fh))
424:  || !(p = decode_filename(p, &args->name, &args->len))
425:  || !(p = decode_sattr3(p, &args->attrs)))
426:         return 0;
```

```
Code Clone 2

File: linux-2.6.19/fs/nfsd/nfsxdr.c

344: if (!(p = decode_fh(p, &args->ffh))
345:  || !(p = decode_fh(p, &args->tfh))
346:  || !(p = decode_filename(p, &args->tname, &args->tlen)))
347:         return 0;
```

G# 2

```
Code Clone 1

File: linux-2.6.19/drivers/hwmon/lm87.c

688: if ((err = device_create_file(&new_client->dev,
689:                         &dev_attr_in6_input))
690:  || (err = device_create_file(&new_client->dev,
691:                         &dev_attr_in6_min))
692:  || (err = device_create_file(&new_client->dev,
693:                         &dev_attr_in6_max)))
694:         goto exit_remove;
```

```
Code Clone 2

File: linux-2.6.19/drivers/hwmon/gl520sm.c

615: if ((err = device_create_file(&new_client->dev,
616:                         &dev_attr_in4_input))
617:  || (err = device_create_file(&new_client->dev,
618:                         &dev_attr_in4_min))
619:  || (err = device_create_file(&new_client->dev,
620:                         &dev_attr_in4_max)))
621:         goto exit_remove_files;
```

Figure 2. False positive clone groups in Linux Kernel. Each row is a pair of inconsistent clones which do not correspond to bugs. Each pair of clones involve the same numbers of `if` statements, `||` operators, function calls, and assignments.

Table I
INFORMAL ILLUSTRATION: REFINEMENT PROCESS

Case 1: Without refinement
Jack is presented with 500 bug reports. He investigates the first 100, and can find five true positives. If the bugs are mission critical, it's worth the effort.
Case 2: With refinement
Jack is presented with 500 bug reports. As he navigates through the bug reports and labels each of them as true bugs or false positives, the system automatically reorders the remaining unlabeled bug reports. He can now find ten true positives after investigating 100 reports. Jack's productivity in finding bugs is doubled.

of all anomaly reports. Thus, we provide a feedback loop between bug detection tools and developers, and help to improve the quality of anomaly reports and reduce the effort of manual investigation. As an informal illustration consider two scenarios in Table I.

Now we describe how this active refinement of anomaly reports could be performed. Conceptually, we divide the space of possible clone groups into four quadrants as shown in Figure 3. The columns separate clone groups that *have* inconsistencies from those that do not; The rows separate clone groups that *allow* variations (*i.e.*, *flexible*) from those that do not (*i.e.*, *rigid*). A rigid clone group is a set of clones where variations among clones are harmful; a flexible clone group is a set of clones where variations are benign[1]. Current clone-based anomaly detection tools would separate clone groups in the two quadrants on the left from those in the two quadrants on the right. However, clone groups in the bottom left quadrant would be all false positives since the inconsistent changes in those clones are allowed or intentional and should not be reported as anomalies. The goal of our approach is to learn a *discriminative model* to provide the likelihood of a clone group belonging to the top left quadrant (*i.e.*, rigid but inconsistent) versus belonging to the bottom left quadrant (*i.e.*, flexible and inconsistent). Then, this model can be used to re-sort the list of anomaly reports and make true positives appear earlier in the list.

	Inconsistent	Consistent
Rigid	True Positive	✓
Flexible	✓	✓

Figure 3. Clone Group's Four Quadrants

We observe that false positives could be similar to one another in certain ways. For example, consider the code snippets in Figure 2 containing two clone groups with two clones each. All of code snippets involve the same number of `if` statements, `||` operators, function calls, and assignments, but they are quite different from the true positive shown in Figure 1. Thus, the intuition of our approach is that false positives may have similar characteristics among themselves, but they have different characteristics from true positives, and differences between false and true positives could be leveraged to build a discriminative model.

In order to characterize the similarities and differences among clones, we convert each clone group into a set of features. These features are built from the various syntactical patterns in the clones of each group. A discriminative model is a composition of features that collectively capture the differences between false and true positives. Often such models are built from a fixed *static* training dataset, e.g., [17], [18]. However, in our bug refinement process, the training

[1] Both notions allow gapped clones, and are orthogonal to the concept of gapped clones.

dataset is incrementally updated as a new anomaly report is inspected and marked by a developer as either a false or true positive, and we would like to build our discriminative model based on such a *dynamic* training dataset.

We propose a framework consisting of a refinement engine that leverages user feedbacks and is iteratively invoked. People need to take action on anomaly reports, to either get bugs fixed or discard them. Our feedbacks are from such actions and no extra effort is needed. The refinement engine is composed of a feature extractor, a pre-processor, and a classifier arranged in a pipeline. It takes in the feedbacks given so far to build and refine discriminative models. The resultant discriminative model in each iteration is used to refine the remaining uninvestigated anomaly reports.

We evaluate our framework on three sets of clone anomaly reports for three large programs: the Linux kernel (C), Eclipse, and ArgoUML (Java) [12] extracted by a clone-based anomaly detection tool. Our evaluation shows that compared to the original ordering of bug reports, we can improve the *average percentage of true positives found*, an evaluation metric adopted from the test case prioritization community [19], by 11%, 87%, and 86% for the Linux kernel, Eclipse, and ArgoUML respectively.

The main contributions of this work are as follows:

1. We present the topic of refining clone anomaly reports.
2. We propose an active learning approach to incrementally refine anomaly reports with user feedbacks.
3. We present an engine that learns discriminative models that can assign the likelihood of each anomaly report being a false positive.
4. We evaluate our proposed approach on three large systems—the Linux Kernel, Eclipse, and ArgoUML—with promising results.

This paper is organized as follows. We describe related work in Section II. In Section III, we review the concept of clone-based anomaly detection. In Section IV, we describe our overall active refinement framework which iteratively invokes a refinement engine. We elaborate this engine in Section V. Our evaluation metric is described in Section VI. In Section VII, we describe our evaluation results on three large software systems. We conclude and mention potential future work in Section VIII.

II. Related Work

There are many studies in software engineering that are related to our work. We summarize them in the following.

1) Code Clone Analysis and Clone-Based Bug Detection: Code clones have been widely studied in the literature. Some studies focus on detection of code clones, based on similarities among strings, tokens, syntax trees, dependency graphs, and even functionalities [2]–[4], [14], [20]–[22]. Clones are traditionally thought as harmful, and techniques have been proposed to reduce clones [23], [24]. On the other hand, some studies show that clones can be useful and necessary [9], [10]. Then, instead of reducing clones, some studies investigate techniques to track and manage code clones [7], [8], [25].

One important use of code clones is to detect bugs. A number of studies detect bugs by detecting inconsistencies among clones [11], [12], [14]–[16]. Such inconsistence-based detection of clone-related bugs often produces many false positives, and uses various filtering rules to reduce false positives. However, even with the most recent filtering techniques, such as ones based on textual similarity and sequence alignment [15], false positive rates remain high. Compared with these studies that use filtering-based approaches to remove reports which may cause false negatives, our approach actively and incrementally refines and reranks anomaly reports based on user feedbacks without removing any report. The code features used in our reranking are also different from those papers. Our work is not an alternative, but rather a complement of others. Filtering-based approaches (which still leave many false positives behind) may be applied first, then our work refines the filtered reports as users take actions, e.g., to fix an anomaly if it is a true positive.

2) Bug Prediction and Triage: Many studies aim to predict whether certain code changes or files may contain faults. Kim *et al.* [26], [27] use bug history to predict faults. Ruthruff *et al.* [28] use logistic regression models from historical data to predict whether a warning generated by FindBugs is actionable. Zimmermann *et al.* [29] have studied the accuracies of bug prediction models that may be used across various projects in various domains.

Other studies aim to reduce similar bug reports or prioritize bug reports. Podgurski *et al.* [30] group software failures with similar symptoms together. Kremenek and Engler [31] propose *z-ranking* to order bug reports produced by a static program checking analysis tool. Heckman and William [32] propose FAULTBENCH, a benchmark for evaluating alert prioritization and classification techniques. These ranking models only perform reordering of bug reports once.

Our approach is different from the above studies in several aspects. We focus on anomaly reports generated by a *clone-based anomaly* detection *tool*, instead of reports from *users*. We *reorder* anomaly reports, while most other studies *filter* reports. Filtering anomaly reports carries a risk of removing true positives. Filtering and reordering are complementary as we could first filter and then re-order anomaly reports. Some studies leverage *historical* data to prioritize anomaly reports, while we leverage *immediate user feedbacks* to iteratively prioritize clone-based anomaly reports.

III. Clone-Based Anomaly Detection

Clone-based bug detection techniques [12], [14] are based on code clone detection and the concept of *contextual consistency*. The intuition behind the technique is that code clones should be inherently similar to each other, and inconsistent changes to the clones themselves or their surrounding code (which are called *contexts*) may indicate unintentional changes, bad programming styles, and bugs.

The technique in [12] is summarized as follows:

1) It uses a code clone detection tool, DECKARD [5], to detect code clones in programs. The output of this step is a set of clone groups, where each clone group is a set of code pieces that are syntactically similar to each other (*a.k.a.* clones);
2) Then, it locates the locations of every clone in the source code and generates parse (sub)trees for them;
3) Next, it detects inconsistencies among the parse trees of the clones and their contexts, e.g., whether the clones contain different numbers of unique identifiers, and how the language constructs of the contexts are different. The inconsistencies are then ranked heuristically based on their potential relationship with bugs. Inconsistent clones unlike to be buggy are also filtered out.
4) Finally, it outputs a list of anomaly reports, each of which indicates the location of a potential bug in the source code, for developers to inspect.

It has been reported that this technique has high false positive rates, even though it can find true bugs of diverse characteristics that are difficult to detect by other techniques. For example, among more than 800 reported bugs for the Linux Kernel, only 41 are true bugs and another 17 are bad programming styles; among more than 400 reported bugs for the Eclipse, only 21 are true bugs and 17 are issues with bad programming styles [12].

IV. OVERALL REFINEMENT FRAMEWORK

A typical clone-based anomaly detection system performs a single batch analysis where a *static* set of anomaly or bug reports (ordered or unordered) are produced. It requires no or little user intervention (e.g., setting some parameters),but may produce many false positives. To alleviate this problem, we propose an active learning approach that can *dynamically* and continually refine anomaly reports based on incremental user feedbacks; each feedback is immediately incorporated by our approach into the ordering of anomaly reports to move possible true positive reports up in the list while moving likely false positives towards the end of the list.

Our proposed active refinement process supporting user feedbacks is shown in Figure 4. It is composed of five parts corresponding to the boxes in the figure.[2] Let us refer to them as Block 1 to 5 (counter-clockwise from left to right). Block 1 represents a typical batch-mode clone-based anomaly detection system. Given a program, the system identifies parts of the program that are different from the norm, where the norm corresponds to the common characteristics in a clone group. Then, the set of anomalies or bugs (*i.e.*, Block 2) is presented for manual user inspection.

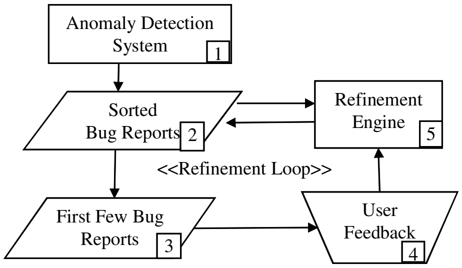


Figure 4. Active Refinement Process

We extend such typical clone-based anomaly detection systems by incorporating incremental user feedbacks through the feedback and refinement loop starting at Block 2 followed by Blocks 3, 4, and 5, and back to Block 2. At Blocks 3 and 4, a user is presented with a few bug reports and is asked to provide feedbacks on whether the reports he or she sees are false or true positives. These feedbacks are then fed into our refinement engine (*i.e.*, Block 5) to update the original or intermediate lists of bug reports.

With user feedbacks, the refinement engine analyzes the characteristics of both false positives and true positives labeled by users so far and hypothesizes about other false positives and true positives in the list based on various classification and machine learning techniques. This hypothesis is then used to rearrange the remaining bug reports. It is possible that a true positive, that is originally ranked low, is moved up the list; a false positive, that is originally ranked high, is "downgraded" or pushed down the list.

The active refinement process repeats and users are asked for more feedbacks. With more iterations, more feedbacks are received, and a better hypothesis can be made for the remaining unlabeled reports.

The ultimate goal of our refinement process is to produce a *better ordering* of bug reports so that true positive reports are listed first ahead of false positives, which we refer to as the *bug report ordering problem*. With better ordering, true positives can be identified earlier without the need to investigate the entire report list. With less false positives earlier in the list, a debugger can be encouraged to continue investigating the rest of the reports and find more bugs in a fixed period of time. If all (or most) of the true positives can appear early, a debugger may stop analyzing the anomaly reports once he or she finds many false positives.

V. REFINEMENT ENGINE

This section elaborates our refinement engine further. Our refinement engine takes in a list of anomaly reports and refines it by reordering the reports. Each anomaly report is a set of code clones (*i.e.*, a clone group) which contain inconsistencies among the clones. Given a list of anomaly reports, ordered either arbitrarily or with some ad-hoc criteria, and user-provided labels (*i.e.*, true positives or false

[2] A square, a trapeze, and a parallelogram represent a process, a manual operation, and data respectively.

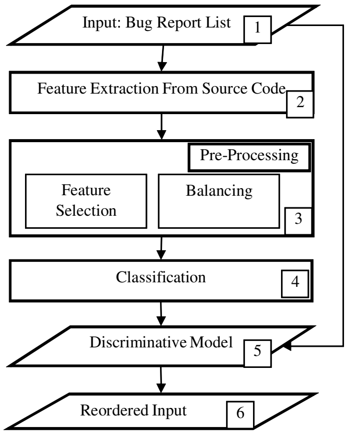


Figure 5. Refinement Engine

positives) for *some* of the reports, our refinement process reorders *unlabeled* anomaly reports based on the predicted likelihood of each of them being true positives. Figure 5 shows our refinement engine that is composed of mainly three blocks: feature extraction, pre-processing (including feature selection and data balancing), and classification.

The feature extraction is meant to transform each clone group into a set of *features*, where each feature is simply a quantitative value that represents a certain property of the code. The set of features is also referred to as a *data point*. In our case, we apply feature extraction to each inconsistent clone group reported by the clone anomaly detection tool and collect a set of data points (*a.k.a. a dataset*) for all clone groups in the reported list.

This feature set is then fed to the preprocessor, which analyzes the data points, and may remove some features or data points from the dataset. Its goal is to smooth over data "noise" as much as possible before classification.

The classifier then takes a preprocessed dataset to mine a classification model that can discriminate features belonging to one class from the other. In our setting, the two classes are false positive class and true positive class. We use *class labels* (`False` and `True`) to indicate whether a clone group is a false or true positive. This mined model in turn is used to predict the class labels of the reported clone groups that have received no user feedback. We also make our classification engine to provide the degree of *likelihood* for a clone group to be in each of the two classes, which is used as a key to rank and sort unlabeled clone groups.

A. Feature Extraction

Our feature extraction block analyzes parse trees which are commonly used to represent programs written in various languages. As a benefit, it is easier to adopt our refinement engine to code written in different programming languages.

A parse tree node is labeled with different information to represent various program constructs e.g., for, switch, etc. Each clone is reported as a sub-tree rooted in a particular node in a parse tree. The feature extraction would construct parse trees for every reported inconsistent clone group and traverse the trees to collect features. More specifically, it performs the following two steps:

1) Tree Construction: For each clone in the anomaly reports, we invoke a parser on the source file containing the clone to construct a parse tree for the file; each node in the tree contains a label indicating its type (*e.g.*, `for`, `if`, `assignment`, etc.). Then, we locate the root node of the subtree that corresponds to the clone. We refer to this subtree as a *clone tree*. We also locate the first ancestor node of this subtree that corresponds to the containing scope of the clone in the source file, and refer to the subtree rooted at this ancestor node as a *clone ancestor tree*.

Clone ancestor trees correspond to more code than clone trees. They may contain more information that could help decide whether an anomaly report is false or true positive. Thus, we extract features from clone ancestor trees.

2) Representing Clone Ancestor Trees as Features: We define five sets of features that could be extracted from a clone ancestor tree, namely: basic, pair, proportional-basic, proportional-pair, and rich. Consider a clone group CG containing a set of clones corresponding to a set of clone ancestor trees, the five sets of features are defined in Definitions 5.1, 5.2, 5.3, 5.4, and 5.5. Rich features belong to the most comprehensive feature set that is a superset of the other four feature sets. Our engine would convert the clone ancestor trees into rich features.

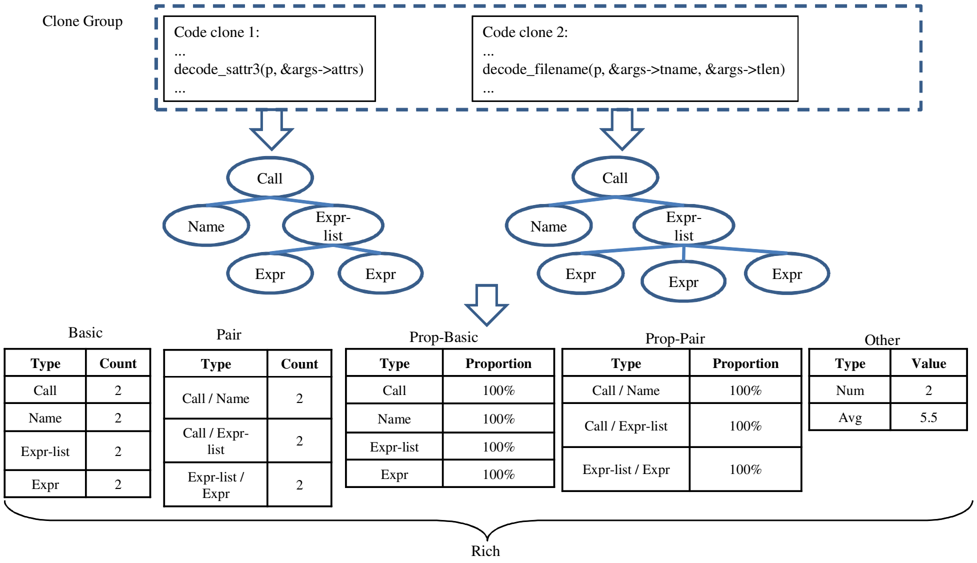
Definition 5.1 (Basic Features): The basic feature set (*Basic*) of a clone group CG is the set of *type-count pairs* in which each pair contains a node type and the number of parse trees in CG having that particular type. For example, considering the trees in Figure 6, the basic feature set contains the following pairs: $\langle Call, 2\rangle$, $\langle Name, 2\rangle$, $\langle Expr\text{-}list, 2\rangle$, and $\langle Expr, 2\rangle$. Mathematically, $Basic(CG) =$

$$\left| \begin{array}{l} (t, |CS|), where \\ CS = \{c \in CG \mid c\ has\ a\ node\ of\ type\ t\} \wedge |CS| > 0 \end{array} \right|$$

Definition 5.2 (Pair Features): The pair feature set (*Pair*) of a clone group CG is the set of *type-count pairs* in which each pair contains a *pair of node types* and the number of parse trees in CG having that particular pair. For example, considering the trees in Figure 6, the pair feature set contains the following pairs: $\langle Call/Name, 2\rangle$, $\langle Call/Expr\text{-}list, 2\rangle$, and $\langle Expr\text{-}list/Expr, 2\rangle$. Mathematically, $Pair(CG) =$

$$\left| \begin{array}{l} ((t_1, t_2), |CS|), where \\ CS = \{c \in CG \mid \exists_{n_1, n_2 \in c}.n_1\ \&\ n_2\ are\ connected\ \wedge \\ n_1\ is\ of\ type\ t_1\ \wedge\ n_2\ is\ of\ type\ t_2\} \wedge |CS| > 0 \end{array} \right|$$

Definition 5.3 (Proportional Features—Basic): The proportional-basic feature set (*Prop-Basic*) of a clone group CG is the set of *type-count pairs* in which each pair contains a node type and the *proportion* of parse trees in CG having that particular type. For example, considering the trees in Figure 6, the *Prop-Basic* feature set



```
Code clone 1:
...
decode_sattr3(p, &args->attrs)
...
```

```
Code clone 2:
...
decode_filename(p, &args->tname, &args->tlen)
...
```

Basic

Type	Count
Call	2
Name	2
Expr-list	2
Expr	2

Pair

Type	Count
Call / Name	2
Call / Expr-list	2
Expr-list / Expr	2

Prop-Basic

Type	Proportion
Call	100%
Name	100%
Expr-list	100%
Expr	100%

Prop-Pair

Type	Proportion
Call / Name	100%
Call / Expr-list	100%
Expr-list / Expr	100%

Other

Type	Value
Num	2
Avg	5.5

Figure 6. Feature Extraction

contains the following pairs: $\langle Call, 100\% \rangle$, $\langle Name, 100\% \rangle$, $\langle Expr\text{-}list, 100\% \rangle$, and $\langle Expr, 100\% \rangle$. Mathematically, $PrBasic(CG)$ =

$$\left| \begin{array}{l} (t, \frac{|CS|}{|CG|}), where \\ CS = \{c \in CG \mid c\ has\ a\ node\ of\ type\ t\} \wedge |CS| > 0 \end{array} \right|$$

Definition 5.4 (Proportional Features—Pair): The proportional-pair feature set (*Prop-Pair*) of a clone group CG is the set of *type-count pairs* in which each pair contains a *pair of node types* and the *proportion* of parse trees of CG having that particular pair. For example, considering the trees in Figure 6, the *Prop-Pair* feature set contains the following pairs: $\langle Call/Name, 100\% \rangle$, $\langle Call/Expr\text{-}list, 100\% \rangle$, and $\langle Expr\text{-}List/Expr, 100\% \rangle$. Mathematically, $PrPair(CG)$ =

$$\left| \begin{array}{l} ((t_1, t_2), \frac{|CS|}{|CG|}), where \\ CS = \{c \in CG\ |\exists_{n_1, n_2 \in c}.n_1\ \&\ n_2\ are\ connected\ \wedge \\ n_1\ is\ of\ type\ t_1 \wedge n_2\ is\ of\ type\ t_2\}\ \ \wedge |CS| > 0 \end{array} \right|$$

Definition 5.5 (Rich Features): The rich feature set (*Rich*) of a clone group CG is the union of the Basic, Pair, Prop-Basic, Prop-Pair feature sets, plus two additional features: the number of clones in CG (Num), and the average size of the clones in CG (Avg). For example, considering the trees in Figure 6, the *Rich* feature set is the union of other features sets plus two additional features: $\langle Num, 2 \rangle$, and $\langle Avg, 5.5 \rangle$. Mathematically, $Rich(CG)$ =

$$\begin{array}{c} Basic(CG) \cup Pair(CG) \cup PrBasic(CG)\ \cup \\ PrPair(CG) \cup \{(Num, |CG|), (Avg, \frac{\Sigma_{c \in CG}.|c|}{|CG|})\} \end{array}$$

B. Preprocessing

We consider two pre-processing options: feature selection and dataset re-balancing. Feature selection is to reduce the number of features by removing unimportant ones. Unimportant features are noise and are good to be removed. Also, as our data contains much more false positives than true positives, we need to re-balance the dataset; otherwise the discriminative model would be biased to always label unknown reports as false positives.

1) Feature Selection: Various approaches have been proposed to select important features. Information gain has been widely used to evaluate the usefulness of a feature, e.g., [33]. If we use c to denote the class labels (true positive [$+ve$ class] vs. false positive [$-ve$ class]), and use f to represent a feature, then information gain of f is defined as in Eq.(1).

$$IG(c|f) = H(c) - H(c|f) \qquad (1)$$

where $H(c) = -\sum_{c_i \in \{\pm ve\}} P(c_i) \log P(c_i)$ is the entropy and $H(c|f) = -\sum P(f) \sum_{c_i \in \{\pm ve\}} P(c_i|f) \log P(c_i|f)$ is the conditional entropy given the feature f.

We select important features based information gain and the Weka toolkit [34] with its default configuration.

2) Dataset Re-balancing: To re-balance the dataset, we reduce the number of data points in the larger class. We retain all data points in the smaller class. For each data point in the smaller class, we find the *most similar* data points in the other class and retain it—cosine similarity [35] between two feature sets corresponding to the two data points is used as the similarity measure. Other data points in the larger class are dropped. This is motivated by the nearest neighbor approach by Renieris and Reiss that localizes bugs by comparing the nearest faulty and correct executions [36]. In their setting, they also have the issue of imbalanced dataset: correct executions are many more than faulty ones.

C. Classification

The classification block takes preprocessed datasets and learns a discriminative model that discriminates true positives from false ones. We refer to the true and false positives as class labels. The purpose of a discriminative model is to take an unlabeled datapoint (*i.e.*, a datapoint or an anomaly report that is not known to be a true positive or a false positive) and assign a class label to it. To produce a discriminative model, the classifier learns from a given labeled training data points. In our case, the training data points are the clone reports that have been investigated by developers to be true positives or false positives.

In this paper, we use a variant of nearest neighbor classification scheme, namely nearest neighbor with non-nested generalization (NNGe) [37]. Nearest neighbor classification has been proved successful for various tasks, e.g., [38]. Also, this technique matches our intuition: an instance similar to known false positives is likely to be a false positive. Our initial study showed that NNGe performs no worse than other common classification approaches. We describe the technique in the following.

1) Nearest Neighbor with Non-Nested Generalization: As its name suggests, in nearest neighbor classification, unknown data would be assigned with the same label as its nearest neighbor. The time needed to build a model would be little as it only involves index building and distance calculation [39].

The nearest neighbor approach can not generalize or group several data points together, which potentially reduces its classification accuracy. Thus, it has been extended with generalization [40]. Rather than loading all data points into the memory, the training phase constructs multi-dimensional rectangles (*i.e.*, hyper-rectangles) that generalize a few data points in a multi-dimensional space. This approach has poor performance on some settings due to nested generalization (*i.e.*, hyper-rectangles are contained inside other hyper-rectangles or overlap with one another) [41]. Martin addresses this issue by proposing nearest neighbor with non-nested generalization (NNGe) [37] which we use in this work. In particular, we use the implementation available in Weka [34] with its default distance function.

In this work, we extend NNGe to output the likelihood for a data point dp to belong to each of the two classes (*i.e.*, true positives (T) and false positives (F)). Let's refer to the set of exemplars, the set of exemplars with label T, and the set of exemplars with label F as D, D_T, and D_F, respectively. Also, considering an exemplar d, let $sim(dp, d) = 1 - dist(dp, d)$, where $dist(dp, d)$ is the distance between dp to the exemplar d which ranges from 0 to 1. Our likelihood measure to re-sort the bug reports is given by the following formula:

$$LH(dp) = 0.5 + \frac{RS(dp)}{2}$$

$$RS(dp) = \frac{|\sum_{d_T \in D_T} sim(dp, d_T)|}{|D_T|} - \frac{|\sum_{d_F \in D_F} sim(dp, d_F)|}{|D_F|}$$

The LH measure corresponds to the normalized relative similarity of a datapoint dp to the datapoints in D_T as compared to those in D_F. Bug reports with higher LH are more likely to be true positives and would be listed first.

D. Concrete Refinement Process

Algorithm 1 is the pseudo-code of our refinement process. It takes in several parameters: the list of bug reports (BR) from a bug detection tool, the initial number of bug reports to be labeled (k), and the feedback pool size (p). The process would be bootstrapped by manually labeling the first k bug reports which are used to train an initial model (Lines 1–5). The classification model is then used to re-sort the unlabeled bug reports (Line 6). The next top p reports are presented for user feedback (Lines 7–8). We only repeat the refinement process after p new feedback are obtained. Then, the feedback are incorporated by learning a new discriminative model and applying it to the remaining unlabeled bug reports in the refinement loop (Lines 3–14). When the false positive rate goes too high, a user can choose to stop the refinement process (Lines 10–11).

With accumulated user feedback (Lines 8 and 13), the refinement process can incrementally improve the classification and ranking accuracy of the discriminative models so that true positives can be ranked higher.

Algorithm 1 Clone Report Refinement Process

```
Input:  BR: Bug Reports
        $k$: Initial set of bug reports to be labeled
        $p$: Feedback pool size
Output: Re-ordered Bug Reports

1:  Let BK = Select the first $k$ bug reports
2:  Label all bug reports in BK (manual)
3:  Let FK = Features extracted from BK
4:  Perform pre-processing on FK
5:  Let M = Classification model created from FK
6:  Refine BR using M
7:  Let BP = Select the new top $p$ unlabeled bug reports
8:  Ask for user feedback on bug reports in BP
9:  Let FPRate = Compute false positive rate
10: If FPRate is too high (based on user feedback)
11:     Stop
12: Else
13:     Set BK = BK $\cup$ BP
14:     Goto 3
```

VI. Evaluation Criteria

In this section we define a suitable metric to measure the quality of the re-sorted bug reports to evaluate the effectiveness of our active refinement process.

Our refinement process is effective if it could re-sort the reports such that all reports corresponding to true positives are listed first. As an illustration, consider a scenario where our refinement process starts with a set of k labeled bug reports and there are m true positive reports among the remaining unlabeled reports. The ideal situation happens when all m other true positives are listed in the $(k+1)^{th}$ to $(k+m)^{th}$ positions after the refinement process. The worst case happens when the true positives are listed as the last m reports after refinement.

To measure the quality of the refinement process, we adapt a measure proposed in test case prioritization area—average percentage faults detected (APFD) [19]. There are a number of similarities between test case prioritization and our problem. In test case prioritization, test cases need to be sorted (*i.e.*, prioritized) in the order of their likelihood to reveal program failures. Also, there is a need to measure and compare the quality of different test case orderings.

In [19], a graph capturing the cumulative proportion of faults captured as more test cases are run is plotted. APFD defined as the area under this curve measures the *rate of fault detection*. In our work, we use the same concept and plot a graph capturing the cumulative proportion of true positives found as more anomaly reports are inspected by users. We refer to this graph as the *cumulative true positives curve*.

In the *cumulative true positives curve*, a larger area under the curve indicates that more true positives are found by developers early, which means that the refinement process effectively re-sorts the anomaly reports. Consider the sample graphs in Figure 7 and assume that there are five true positives among 10 reports. Each of the five increments in each of the two cumulative curves corresponds to when each of the five true positives is found. The left curve shows that true positives are found at positions 1, 2, 3, 7, and 8. The right curve shows that true positives are found at positions 1, 2, 3, 4, and 5. Using the original list of reports (left), the developer could only find three true positives by investigating the first six reports. Using the refined list produced by the refinement process (right), five true positives are found in the first six reports. Hence, by performing the same number of inspections (which may correspond to the time budget a developer has in real-world situations), the developer could find more true positives using the refined report list as compared to the original one. In this case, the refined report list has a better true positive detection rate. In the graphs, this improvement is indicated by a larger area under the curve for the refined report list.

The idea of using APFD as the evaluation criteria for bug finding is also used by Kremenek and Engler [31]. Following the same idea, we define *average percentage true positives found* (APPF) as the area under the *cumulative true positives curve*. Our goal is to improve the APPF score which measures the rate of true positives found. We illustrate APPF improvement computation in Figure 7.

VII. Empirical Evaluation

In this section, we describe our experimental settings, our evaluation results, and the threats to validity.

A. Settings and Results

1) Settings: We evaluate our approach on clone-based anomaly reports for three real programs written in different programming languages: the Linux kernel (C), Eclipse and ArgoUML (Java). We analyze the reports generated by Jiang *et al.* [12]. We choose these reports as they contain a large number of false positives. There are more than 800 anomaly reports (*i.e.*, clone groups) for the Linux kernel, and only 57 of them are true positives or programming style issues. There are more than 400 anomaly reports for Eclipse, and only 38 of them are true positive. There are more than 50 anomaly reports for ArgoUML, and only 15 of them are true positive. Finding a few true positives on the large number of false positives is a challenging task that would stress test the usability of our approach. The authors from [12] have manually labeled all the reported inconsistent clone groups from the Linux kernel and Eclipse as either true positives or false positives. We manually label the reported inconsistent clone groups from ArgoUML. We use these clone groups and their labels to simulate initial and incremental user feedbacks as inputs to our refinement engine.

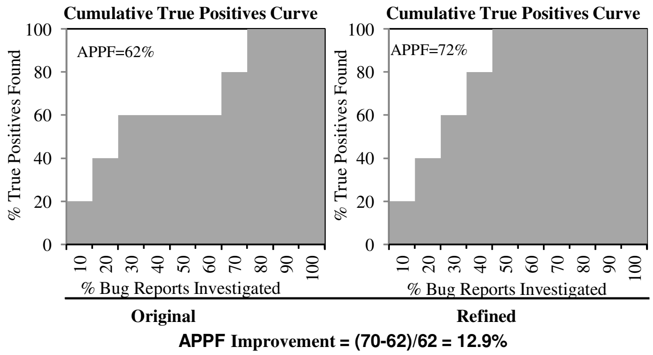


Figure 7. Computing and Comparing APPF

The tool in [12] returns the list of anomalies in a particular order. We take the ordering returned by the tool and refine it. Following the steps in Section V, we initially take the first k labeled clone groups. We set k to be 50 since there is only one true positive among the first 50 bug reports for Eclipse. We also use $k = 50$ for the Linux kernel. Since there are only 50 inconsistent clone groups reported for ArgoUML, we set k to be 10 for ArgoUML. We set the feedback pool size (i.e., p) to be 1. We thus iteratively refine the bug reports as each feedback is received. In this paper, we repeat the refinement process until all anomaly reports are inspected.

2) Evaluation Results: We improve APPF by 11%, 87%, and 86% for the Linux kernel, Eclipse, and ArgoUML bug reports respectively. These measures mean that within a limit period of time, a developer investigating the anomaly reports may find more true positives in the Linux kernel, Eclipse, and ArgoUML. The improvement for the Linux kernel is not as much as Eclipse and ArgoUML. The bugs in the Linux kernel often involves identifier changes (e.g., variations in variable names, function names, type names, etc.) which are not captured well by our feature sets which are mostly based on syntactical node types, while the bugs

Table II
TOP-5 RE-ORDERINGS. $x \mapsto y$ MEANS THAT A REPORT OF A TRUE POSITIVE AT POSITION x IS REORDERED TO POSITION y.

System	Top-5 Re-orderings
Linux	694 $\mapsto$ 18, 672 $\mapsto$ 64, 760 $\mapsto$ 131, 770 $\mapsto$ 179, 792 $\mapsto$ 206
Eclipse	373 $\mapsto$ 4, 348 $\mapsto$ 11, 394 $\mapsto$ 29, 388 $\mapsto$ 43, 370 $\mapsto$ 49
ArgoUML	40 $\mapsto$ 12, 35 $\mapsto$ 15, 34 $\mapsto$ 11, 29 $\mapsto$ 9, 23 $\mapsto$ 8

Table III
TOP 3 FEATURES BASED ON THEIR INFORMATION GAIN: LINUX KERNEL. THE ## SYMBOL IS THE SEPARATOR BETWEEN TWO FEATURES FOR A PAIR FEATURE SET. THE ^{P} SUPERSCRIPT DENOTES A PROPORTIONAL FEATURE.

Top	Feature	Info. Gain
1	extdef^P	0.015941
2	extdef_1^P	0.015941
3	program##extdefs^P	0.015941

Table IV
TOP 3 FEATURES BASED ON THEIR INFORMATION GAIN: ECLIPSE. THE ^{B} SUPERSCRIPT DENOTES A BASIC FEATURE WHILE ^{P} SUPERSCRIPT DENOTES A PROPORTIONAL FEATURE

Top	Feature	Info. Gain
1	BOOL_OR_TK^P	0.01898
2	conditional_or_expression ## conditional_or_expression^P	0.01898
3	BOOL_OR_TK^B	0.01898

Table V
TOP 3 FEATURES BASED ON THEIR INFORMATION GAIN: ARGOUML

Top	Feature	Info. Gain
1	local_variable_declaration_statement^B	0.145772
2	variable_initializer^B	0.145772
3	block_statement ## local_variable_declaration_statement^B	0.145772

in Eclipse and ArgoUML often involve conditionals which may be better captured by our features. In future, we plan to add more features to construct better discriminative models for Linux and more programs.

The top-5 successful re-orderings for the Linux kernel, Eclipse and ArgoUML are shown in Table II. We highlight sample bugs that are successfully reordered. Figure 8 shows a buggy clone group in Linux that is reordered from position 694 to 18. The bug is related to an early unlock of a variable. Figure 9 shows a bug from Eclipse that is successfully reordered from position 373 to 4. The bug is similar to the bug in Figure 1; it misses a null-check in code fragment 2, and was reported to developers and fixed. For ArgoUML, a bug shown in Figure 10 is reordered from position 40 to 12. This bug is related to a missing validation before a variable is used in the next statement.

To further investigate which program elements (*i.e.*, features as described in Section V-A) may be better bug indicators than others, we compute the *information gain* [42] of each feature in Linux and Eclipse bug reports. Information gain is frequently used to find important features that differentiate two contrasting datasets (*i.e.*, in our case, true positives and false positives), e.g., [33].

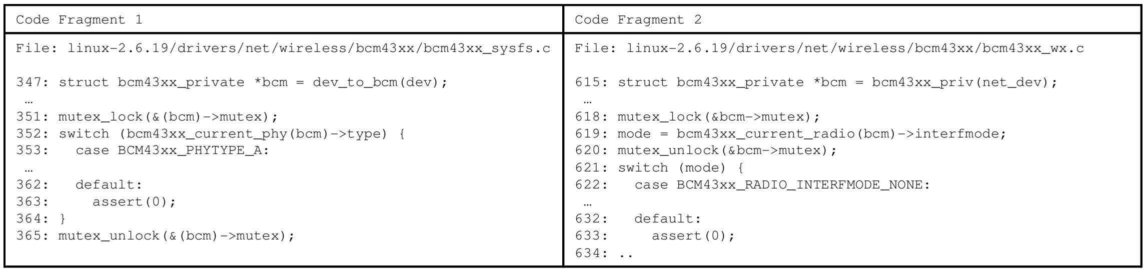
The top 3 features for Linux kernel, Eclipse, and ArgoUML are shown in Table III, IV, and V. We notice that the individual features have low information gain. Thus, individually they are not able to distinguish true positives from false positives. However, composing them into a discriminative model is more effective in improving the rate of true positives found. From the list, one could intuitively infer that if a clone from Eclipse involves inconsistent changes related to conditionals, it is more likely to be buggy. For ArgoUML, the inconsistent changes that involve variable declaration and initialization are more likely to be buggy, e.g., a declared variable being used without further validation or checking (null checking), a variable needs to be converted to an appropriate type, etc. As discriminative features using*information gain* could mean that the features are either highly related to buggy clone or highly related to non-buggy clone, in Linux kernel, if a clone involves inconsistent changes related to global definitions (*e.g.*, extdef), it is more likely not a bug. Overall, our approach helps to better separate false positives from true bugs, making first listed anomaly reports closer towards the top left cell of the ideal four quadrants of Figure 3.

B. Threats to Validity

Threat to construct validity corresponds to the suitableness of our evaluation metric. In this study we adapt a measure commonly used in test case prioritization which also needs to re-sorts (*i.e.*, prioritize) test cases. Their goal is to find an optimal ordering of test cases that would identify the failures early. They measure the quality of an ordering using *average percentage faults detected* (APFD). We propose a similar measure referred to as *average percentage of true positives found* (APPF). Similar like APFD that measures the rate of fault detection, APPF measures the rate of true positives found. We believe this measure is relevant in measuring the performance of a refinement framework. A higher APPF score indicates that within the same period of time a developer can find more true positives.

Threat of internal validity corresponds to the ability of our experiments to link the independent and dependent variables. The threat could be manifested due to experimental or human errors. The labels of the bug reports are decided manually by the authors of [12]. The labeling might be prone to errors. Still, the authors of [12] and us have taken some precautions to prevent these to happen – at least two people are assigned to label each of the inconsistent clone group; for any discrepancy, a third person would break the tie.

Threat of external validity corresponds to the generalizability of our result. We have performed a study on three large real systems that are written in two most popular programming languages: C and Java. Although these help, there is still a threat to external validity. In the future, we plan to investigate more systems written in various programming languages.



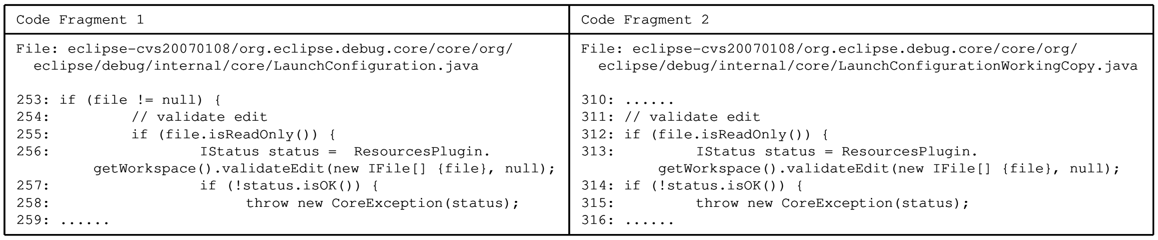
Code Fragment 1

```
File: linux-2.6.19/drivers/net/wireless/bcm43xx/bcm43xx_sysfs.c

347: struct bcm43xx_private *bcm = dev_to_bcm(dev);
 …
351: mutex_lock(&(bcm)->mutex);
352: switch (bcm43xx_current_phy(bcm)->type) {
353:   case BCM43xx_PHYTYPE_A:
 …
362:   default:
363:     assert(0);
364: }
365: mutex_unlock(&(bcm)->mutex);
```

Code Fragment 2

```
File: linux-2.6.19/drivers/net/wireless/bcm43xx/bcm43xx_wx.c

615: struct bcm43xx_private *bcm = bcm43xx_priv(net_dev);
 …
618: mutex_lock(&bcm->mutex);
619: mode = bcm43xx_current_radio(bcm)->interfmode;
620: mutex_unlock(&bcm->mutex);
621: switch (mode) {
622:   case BCM43xx_RADIO_INTERFMODE_NONE:
 …
632:   default:
633:     assert(0);
634: ..
```

Figure 8. Report of a true positive in Linux that is reordered from position 694 to 18



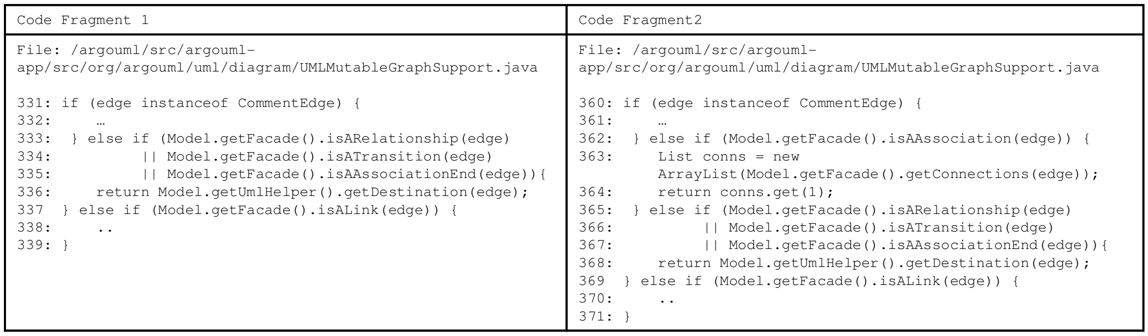
Code Fragment 1

```
File: eclipse-cvs20070108/org.eclipse.debug.core/core/org/
  eclipse/debug/internal/core/LaunchConfiguration.java

253: if (file != null) {
254:          // validate edit
255:          if (file.isReadOnly()) {
256:                  IStatus status =  ResourcesPlugin.
          getWorkspace().validateEdit(new IFile[] {file}, null);
257:                  if (!status.isOK()) {
258:                        throw new CoreException(status);
259: ......
```

Code Fragment 2

```
File: eclipse-cvs20070108/org.eclipse.debug.core/core/org/
  eclipse/debug/internal/core/LaunchConfigurationWorkingCopy.java

310: ......
311: // validate edit
312: if (file.isReadOnly()) {
313:         IStatus status = ResourcesPlugin.
       getWorkspace().validateEdit(new IFile[] {file}, null);
314: if (!status.isOK()) {
315:         throw new CoreException(status);
316: ......
```

Figure 9. Report of a true positive in Eclipse that is reordered from position 373 to 4



Code Fragment 1

```
File: /argouml/src/argouml-
app/src/org/argouml/uml/diagram/UMLMutableGraphSupport.java

331: if (edge instanceof CommentEdge) {
332:     …
333:  } else if (Model.getFacade().isARelationship(edge)
334:           || Model.getFacade().isATransition(edge)
335:           || Model.getFacade().isAAssociationEnd(edge)){
336:     return Model.getUmlHelper().getDestination(edge);
337  } else if (Model.getFacade().isALink(edge)) {
338:     ..
339: }
```

Code Fragment2

```
File: /argouml/src/argouml-
app/src/org/argouml/uml/diagram/UMLMutableGraphSupport.java

360: if (edge instanceof CommentEdge) {
361:     …
362:  } else if (Model.getFacade().isAAssociation(edge)) {
363:     List conns = new
         ArrayList(Model.getFacade().getConnections(edge));
364:     return conns.get(1);
365:  } else if (Model.getFacade().isARelationship(edge)
366:           || Model.getFacade().isATransition(edge)
367:           || Model.getFacade().isAAssociationEnd(edge)){
368:     return Model.getUmlHelper().getDestination(edge);
369  } else if (Model.getFacade().isALink(edge)) {
370:     ..
371: }
```

Figure 10. Report of a true positive in ArgoUML that is reordered from position 40 to 12

VIII. Conclusion and Future Work

Code clones have been widely studied in the literature. Various techniques have been proposed to recover clones from a code base. One important usage of clones is to find bugs by detecting inconsistencies among members of the same clone group. These correspond to bugs that might arise due to inconsistent updates among parallel code fragments or violation of a common programming practice. Past techniques, e.g., [12], have demonstrated the ability of clone-based bug detection tools to detect true positives in large systems. However, often the number of false positives are too many. This could affect the usability of such a system as a developer could spend a lot of time in performing a debugging activity, which at the end might be futile, as the reported anomaly is a false alarm.

Our work tries to address this issue by proposing an approach to automatically refine bug reports by the incorporation of user feedback. Rather than having a *static* sorted list of bug reports, our bug reports are *dynamic*. As a user investigates the top few bug reports and feedback to the system, the system automatically re-sorts the remaining uninvestigated bug reports, and thus *refines* it. This refinement process is performed multiple times as more feedback is available. For each refinement, we perform feature extraction, pre-processing (feature selection and dataset re-balancing), and discriminative model learning. To evaluate the quality of a list of ordered bug reports we use *average percentage of true positives found* (APPF) which measure the rate true positives are found. We evaluate our refinement process on three sets of clone-based anomaly reports from three large real programs: the Linux kernel (C), Eclipse, and ArgoUML (Java), extracted by a clone-based anomaly detection tool. The results show that, compared to the original ordering of bug reports, we can improve APPF by 11%, 87%, and 86% for Linux kernel, Eclipse, and ArgoUML, respectively.

As future work, we plan to extend our approach to refine not only clone-based anomaly reports but also other types of anomaly reports. We also plan to investigate the applicability of models learned from one or more software systems to refine bug reports of other software systems.

Acknowledgments

We appreciate the valuable feedbacks from anonymous reviewers for earlier versions of this paper.

REFERENCES

[1] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder," in *ICSE*, 2007.

[2] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS©: Program transformations for practical scalable software evolution," in *ICSE*, 2004.

[3] B. S. Baker, "Finding clones with Dup: Analysis of an experiment," *IEEE TSE*, 2007.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE TSE*, 2002.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007.

[6] Y. Higo, S. Kusumoto, and K. Inoue, "A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system," *Journal of Software Maintenance*, vol. 20, no. 6, pp. 435–461, 2008.

[7] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *ICSE*, 2007.

[8] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone-aware configuration management," in *ASE*, 2009.

[9] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ESEC/FSE*, 2005.

[10] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *WCRE*, 2006.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE*, 2009.

[12] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *ESEC/SIGSOFT FSE*, 2007.

[13] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *WCRE*, 2007.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *OSDI*, 2004.

[15] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and systematic detection of buggy inconsistencies in source code," in *OOPSLA*, 2010.

[16] Y. Hayase, Y. Y. Lee, and K. Inoue, "A criterion for filtering code clone related bugs," in *DEFECTS*, 2008.

[17] J.-G. Lee, J. Han, X.Li, and H. Cheng, "Mining discriminative patterns for classifying trajectories on road networks." *IEEE Trans. Knowl. Data Eng.*, 2011.

[18] A. Bosch, A. Zisserman, and X. Munoz, "Scene classification using a hybrid generative/discriminative approach." *IEEE Trans. Pattern Anal. Mach. Intell.*, 2008.

[19] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies." *IEEE Trans. Software Eng.*, vol. 28, pp. 159–182, 2002.

[20] V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *SCAM*, 2004.

[21] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *SAS*, 2001.

[22] A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior," *ACM TOSEM*, 1993.

[23] S. Jarzabek and S. Li, "Eliminating redundancies with a "composition with adaptation" meta-programming technique," in *ESEC/FSE*, 2003.

[24] D. C. Rajapakse and S. Jarzabek, "Using server pages to unify clones in web applications: A trade-off analysis," in *ICSE*, 2007.

[25] P. Jablonski and D. Hou, "CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in *ETX*, 2007.

[26] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller, "Predicting faults from cached history," in *ICSE*, 2007.

[27] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *ESEC-FSE*, 2007.

[28] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *ICSE*, 2008.

[29] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process," in *ESEC/FSE*, 2009.

[30] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *ICSE*, 2003, pp. 465–477.

[31] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximation," in *SAS*, 2003, pp. 295–315.

[32] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *ESEM*, 2008, pp. 41–50.

[33] J. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[34] G. Holmes, A. Donkin, and I. Witten, "Weka: A machine learning workbench." in *Proc Second Australia and New Zealand Conference on Intelligent Information Systems*, 1994.

[35] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[36] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 30–39.

[37] B. Martin, "Instance-based learning : Nearest neighbor with generalization." in *Thesis, University of Waikato, Hamilton, New Zealand*, 1995.

[38] S. Tan, "An effective refinement strategy for knn text classifier," *Expert Syst. Appl.*, 2006.

[39] A. Moore, "An introductory tutorial on KD-Trees," Computer Laboratory, University of Cambridge, Tech. Rep. 209, 1991, extract from Andrew Moore PhD Thesis: Efficient Memory-based Learning for Robot Control.

[40] S. Salzberg, "A nearest hyperrectangle learning method," *Machine Learning*, 1991.

[41] D. Wettschereck and G. Dietterich, "An experimental comparison of the nearest-neighbour and nearest-hyperrectangle algorithms." *Machine Learning*, 1994.

[42] T. M. Mitchell, *Machine Learning*. New York, USA: McGraw-Hill, March 1997.