Feature-Based Detection of Bugs in Clones

Daniela Steidl, Nils Göde
CQSE GmbH, Germany
{steidl, goede}@cqse.eu

***Abstract*—Clones bear the risk of incomplete bugfixes when the bug is fixed in one code fragment but at least one of its copies is not changed and remains faulty. Although we find incompletely fixed clones in almost every system, it is usually time consuming to manually locate these clones inside the results of an ordinary clone detection tool. In this paper, we describe in how far certain features of clones can be used to automatically identify incomplete bugfixes. The results are relevant for developers to locate incomplete bugfixes—that is, defects still existing in the system—and for us as clone researchers to quickly find examples that motivate the use of clone management.**

***Index Terms*—Software quality, code clones, bug detection**

I. Introduction

The continuous research and improvement of clone detection algorithms has led to a number of industrial strength approaches, for example [1]–[3], that can detect not only identical code fragments but also similar code fragments with minor differences. These clones are commonly referred to as *inconsistent* clones or *gapped* clones. Figure 1, taken from [2], shows such an example. Multiple studies, for example [2], [4], [5], have shown that these differences are often unintentional and may sometimes point to bugs, which have been fixed only partially. That is, the bug was fixed in some of the fragments, but at least one of the copies was not changed and the bug remained effectively in the system.

As quality experts, we are frequently confronted with large industrial systems from domains such as automotive systems, energy controls, or insurance business. Our clone analysis, an integral part of our quality assessment, shows that incomplete bugfixes exist in almost every system. However, it is unclear which factors induce bugs in cloned code. Bugs in clones are relevant for both developers and researchers. Clones that resemble incomplete bugfixes point to bugs that still exist is software systems and may cause sever damage. Hence, detecting them allows to reduce the number of defects in software systems. Besides improving correctness, incomplete bugfixes are also well-suited examples for clone researchers to motivate the use of clone management. Finding clone-related bugs helps us as researchers and quality experts to argue why clone management should be an integral part of quality control.

Our experience tells us that incompletely fixed clones exist in almost every system. Locating them, however, is a time-consuming process. A number of approaches have been suggested to sort clones according to the probability of containing a bug. However, none of these approaches was evaluated with respect to how effective the sorting really is. Therefore, we are currently left with manual inspection of the clones in random order. Given the number of clones that is usually returned by an ordinary clone detection tool, the inspection takes long since incomplete bugfixes are usually found in only a small percentage of all detected clones [1], [2].

Problem Statement. *Software contains a large number of clones. Currently, there exists no algorithm to efficiently detect gapped clones that contain an incompletely fixed bug as it is unclear which factors induce bugs in cloned code.*

In this paper, we evaluate which factors induce bugs in cloned code and in how far we can speed up the detection process of incompletely fixed bugs by using an automatic clone classification. We often observe that clones with incompletely fixed bugs have specific features.[1] For example, these clones often have only few differences (gaps). For clones with multiple gaps, the differences are much more likely to be intentional. To detect which clone features are relevant for predicting bugs, we employ machine learning algorithms. Please note that we do not think such an algorithm can provide a perfect classification, but any automated classification with higher precision than manual inspection in random order would be beneficial.

Contribution. *We examine which clone features are relevant to predict incompletely fixed bugs.*

Outline. This paper is structured as follows. Section II presents previous work which is related to ours. Section III describes the data used for machine learning. We explain the relevant features of clones in Section IV. Section V, Section VI, and Section VII outline the machine learning algorithm we use along with the evaluation technique. Section VIII presents our results which are discussed in Section IX. Section X contains threats to validity, our conclusions are given in Section XI.

II. Related Work

In this section, we present previous work, which is related to ours. We limit ourselves to publications that investigate the relation between clones and bugs as part of a case study. For a general overview of clone research, please refer to existing surveys—e. g., [6] and [7].

Li et al. [4] detected copy-paste related bugs based on the consistency of identifiers in clone pairs. In particular, they calculated the *UnchangedRatio* that describes to which degree

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A". The responsibility for this article lies with the authors.

[1] In this paper we use the term *feature*, taken from the machine-learning terminology, to describe characteristics of code clones.

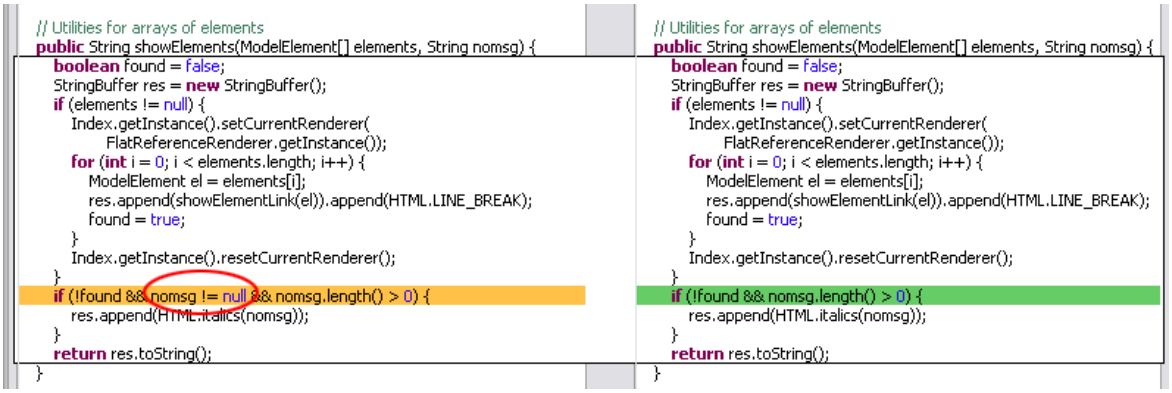

Fig. 1. Example of an inconsistent clone

identifiers are consistent. Their hypothesis is the lower the *UnchangedRatio* (the more identifiers have been changed), the more likely the clone pair contains a bug—except for when the *UnchangedRatio* is 0. Although Li et al. use the *UnchangedRatio* to prune the results and sort clone pairs according to their likelihood of having a bug, there is no empirical investigation of how useful the *UnchangedRatio* really is to identify clones with bugs.

An alternative approach to detect bugs based on inconsistencies has been presented by Jiang and colleagues [8]. Instead of inconsistencies between the cloned fragments themselves, inconsistencies in the contexts of the cloned fragments have been used to locate bugs. Jiang et al. identified different types of context inconsistencies which potentially indicate whether the inconsistency is a bug or not. But again, the usefulness of the types of inconsistencies has not been empirically evaluated.

Higo et al. [5] presented another approach to detect bugs based on inconsistencies. Clones which might contain a bug were extracted by subtracting the results of a more restrictive detector from those of a more tolerant clone detector to extract certain types of clones which potentially contain a bug. Whether these particular clones are more likely to contain a bug than others has not been investigated.

An elaborate study on bugs in inconsistent clones has been conducted by Juergens et al. [2]. For systems from different domains, experts rated whether a given inconsistent clone contained a bug or not. The results show that a notable number of inconsistencies in clones are actually unintentional and contain a bug. However, apart from the inconsistency, no other features of the clones have been analyzed according to whether they might help to find bugs.

In summary, all of the previous studies identified bugs in clones, using a particular pre-sorting based on certain features to identify clones likely to contain a bug. However, no previous work investigated how good the pre-sorting actually is. The usefulness of certain clone features to categorize the clone as defective or not is unkown. In this paper, we extend previous research by establishing a list of features motivated by earlier work and our practical experience and use machine learning to analyze the usefulness of these features.

TABLE I
SUBJECT SYSTEMS

Name	Organization	Language	Age [Years]	Size [kLOC]
A	MunichRe	C#	6	317
B	MunichRe	C#	4	454
C	MunichRe	C#	2	495
Sysiphus	TUM	Java	8	281

III. Case Study Data

In the following, we describe the training data used to predict incompletely fixed bugs in gapped clones. To obtain reasonable results, it is important to have a sufficiently large set of clone classes where experts for the corresponding code determined if they contain an incompletely fixed bug. Since this process is very time-intensive for us and the experts, we reuse data of a previous study [2], which had a different goal, but nevertheless collected data suitable for this approach. These data include the code base of the subject systems, the detected clones, and the experts' rating.

Systems. The training data comprises five systems studied in [2]. Due to our syntax-based classification, we excluded the COBOL system because of the significant syntactic differences between COBOL and the C-like languages C# and Java. The remaining subjects are three systems developed by different organizations for the *Munich Re Group*—one of the largest re-insurance companies in the world—and a collaboration environment for distributed software development projects, *Sysiphus*, developed at the Technische Universität München (TUM). Details about the systems are given in Table I.

Clone Detection. We also reused the clones from the former study, which were detected using our tool ConQAT.[2] Generated code was excluded prior to the detection. The clone detection algorithm is in principle token-based. However, the individual tokens of each statement were merged into a single artificial token representing that statement. Hence, detection is based on the sequence of statements rather than the sequence

[2]www.conqat.org

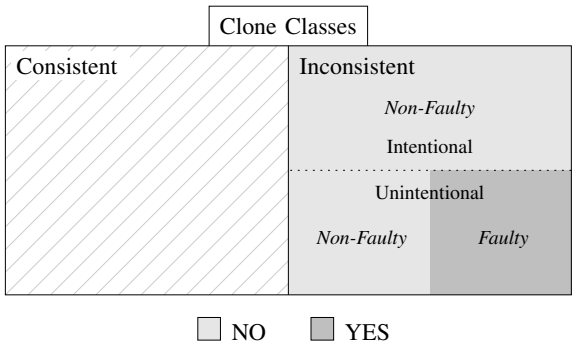

Fig. 2. Rating of clone classes as faulty and non-faulty

of original tokens. Identifiers and literals have been normalized except for regions of repetitive code. Clones were not allowed to cross method boundaries. The minimal clone length was set to 10 statements. For a more detailed description of the algorithm and its parameters, please refer to [2].

Rating. We also reused the developers' rating of the clones from the previous study. The detected clone candidates were presented to the developers who rated them in three steps:

1) The developers removed false positives of the clone detection, i. e., code fragments without semantic relationship although the tool detected them as clones. From the true positives, we removed clones without inconsistencies (gaps).
2) For inconsistent clones, the developers rated the inconsistencies as unintentional or intentional.
3) If a clone class was unintentionally inconsistent, developers classified it as faulty or non-faulty, or don't know in case their decision was unsure. Intentional inconsistent clones are non-faulty by definition.

Figure 2 visualizes the sets of different clones. In cases where developers could not determine intentionality or faultiness, e. g., because none of the original developers was available for rating, the inconsistencies were treated as intentional and non-faulty. We used the developers' answers from the third step to assign each clone class one of the following labels:

- **YES.** Differences between gapped clones were unintentional and faulty: the clone class contains an incompletely fixed bug (dark gray in Figure 2).
- **NO.** The differences between the cloned fragments are on purpose or were unintentional but do not contain an incompletely fixed bug (light gray in Figure 2).

Data Size. In total, the developers rated 582 gapped clones, among which 104 clones were labeled with *YES*, i. e., contained an incompletely fixed bug. 456 clones were labeled with *NO*. We excluded 22 clones from the data set which were labeled with *don't know*. The resulting data set is an *imbalanced data set* as one class (with label *NO*) is highly overrepresented compared to the other class (with label *YES*). We describe in Section VI how we handle the imbalanced data set in our approach.

IV. Features

Our hypothesis is that we can identify incompletely fixed bugs based on certain features of gapped clones. Therefore, we evaluate in how far this hypothesis holds for a given set of features. The features we use originate from recurring patterns that we observed during many years of manually inspecting gapped clones. This experience includes but is not limited to the clones in our training data set. Please note that the set of all possible features is infinite and our selection provides only a first starting point. However, our analysis can be repeated for other features with only little effort. The remainder of this section describes the selected features that turned out to be useful for machine learning and documents our assumptions why we believe a feature is relevant for predicting faulty inconsistencies.

We use features to describe both the whole content of cloned fragments (*global* features) as well as only the gaps between the clones (*local* features).

A. Global Context Features

- **length (Integer)** – *Length of copied code fragment ignoring gaps.*
  Bugs are more likely to occur when there are differences within long similar code sequences compared to only short fragments.
- **nesting_depth (Integer)** – *Maximal nesting depth of a code fragment.*
  Faulty inconsistencies between code fragments are more likely to occur in algorithmic code with higher nesting depth than in data code.
- **comment (Boolean)** – *Any gap is preceded by a comment.*
  Often developers comment on a gap when fixing a bug and denote an issue-id or a short bug-fix discription. Hence, we assume commented gaps to indicate an incomplete bugfix.
- **preceded_by_assignments (Boolean)** – *Any gap is preceded by two assignment statements.*
  It is easy to forget a single assignment while writing a set of assignments, e. g., setting various attributes of an object. Hence, a gap preceded by a set of assignments can indicate a faulty inconsistency. We use the simple heuristic to detect assignments by searching for lines containing = but not `if`, `else`, `for`, and `while`.

B. Local Gap Features

- **attribute (Boolean)** – *Any gap contains an attribute declaration.*
  An attribute declaration seems to be a major change, indicating that the change was on purpose.
- **new (Boolean)** – *Indicator for any gap containing the keyword `new`.*
  The keyword `new` signals the creation of an object. We assume the creation of a new object to represent a major change on purpose.

- **equal_null (Boolean)** – *Any gap contains a condition including* `== null`.
  While writing `if`, `for`, or `while` statements, it is easy to forget a null check. Hence, we believe a gap containing a null check indicates an incomplete bugfix.
- **not_equal_null (Boolean)** – *Any gap contains a condition including* `!= null`.
  Analogous as the previous feature, we believe that an easily forgotten `!= null` check indicates a faulty inconsistency.
- **continue (Boolean)** – *Any gap contains the keyword* `continue`.
  During the implementation of loops, developers easily forget necessary continues. We assume this indicates a faulty inconsistency.
- **break (Boolean)** – *Any gap contains the keyword* `break`.
  During the implementation of loops, developers easily forget necessary breaks. We assume this indicates a faulty inconsistency.
- **num_token_type (Integer)** – *Number of different token types in the gaps.*
  We believe inconsistencies are more likely to occur in algorithmic code than in data code with fewer token types than algorithmic code.
- **method_call (Boolean)** – *Any gap matches the method call pattern*
  We use `[a-zA-Z]+\.[a-zA-Z]+\(.*\)` as a regular expression for a method call and assume that a method call in a gap indicates a bugfix.

C. Feature Implementation Details

Both global and local features of gapped clones contain boolean and integer features. In general, machine learning algorithms are applicable not only for feature representations with a unique type (e. g., an integer vector) but also for feature representations with mixed types (e. g., a feature vector with booleans and integers). However, including integer features with a broad range of possible values (e. g., the *length* feature with a value range of [*minimal clone length*..∞]) in a feature vector with mostly boolean features causes problems during machine learning. The classifier predominantly uses this feature to achieve a high precision/recall: The classifier makes a decision by splitting the large feature value range into very small subintervals such that most subintervals contain exactly one training data point. Consequently, the classifier separated the data set and does not need to consider other features. Solely based on this large-range integer feature, the classifier performs reasonably well. However, it strongly overfits the training data and cannot be generalized.

To avoid this phenomenon, we manually limit the number of different values for integer features that have a broad value range, i. e., the *length* and *num_token_type* feature. (The *nesting_depth* feature, in contrast, reveals a small value range in our data set with most values being in the interval $[0..4]$ and a barely occurring maximum value of 6. Hence, the chance of overfitting based on this feature is low.)

For the *length* features we split the value range as follows: [*minimal clone length*..20] is mapped to value 0, $[21..\infty]$ to 1. For the *num_token_type* feature, values in $[0..5]$ are transformed to 0, $[6..10]$ to 1 and $[11..\infty]$ to 2. The feature value transformations were obtained based on preliminary experiments and manual inspection of the data set.

V. Decision Trees

We assume that the features outlined in the previous section are relevant to classify a gapped clone as incomplete bugfix or not. Using a machine learning algorithm, we analyze how relevant each feature truly is. In this paper, we use *decision trees* as classifier and the standard machine learning library WEKA[3] for implementation.

A decision tree is a tree with decision nodes as interior nodes and class labels as leaves. To classify an instance, the decision tree is a set of `if-then-else` statements: Based on the value of a single feature, a decision is made at each interior node to further traverse the left or the right branch of the tree until a leaf is reached. The class label represented by the leaf is assigned to the instance as its classification.

After preliminary experiments comparing the performance of various different classifiers (support vector machines, naives bayes, AdaBoost, etc.) we chose decision trees due to their results being the most promising and easy to understand: the result of a decision tree can be easily visualized and understood by humans as opposed to, e. g., the high-dimensional hyperplanes of support vector machines. In particular, we used the J48 implementation of a decision tree [9].

VI. Cost-Sensitive Training

As described in Section III, our underlying training data set is imbalanced. This is problematic as machine learning algorithms operate under two assumptions:

1) The goal is to maximize the prediction accuracy.
2) The test data has the same underlying distribution as the training data.

In our data set, 18.6% of the data is labeled with *YES* (minority class), 81.4% is labeled with *NO* (majority class). Consequently, if the classifier predicted label *NO* for all data, it would already achieve an accuracy of 81.4% by default without learning from the data.

Machine learning research has proposed a variety of methods to overcome issues with imbalanced data sets, [10], i. e., artificially sampling the data set to balance class distributions or adapting the learning algorithms: To improve the data set, upsampling approaches (interpolating more data samples of the minority class) or downsampling approaches (removing samples of the majority class) exist. In our case, the naive approach of randomly downsampling (using the WEKA SpreadSubsample) did not succeed as the results strongly varied with the randomly chosen negative data points and hence overfitted the data. More sophisticated approaches to

[3] http://www.cs.waikato.ac.nz/ml/weka/

TABLE II
COST MATRIX FOR FALSE POSITIVES AND FALSE NEGATIVES

Classified as →	*YES*	*NO*
YES	0	5 (FN)
NO	1 (FP)	0

upsample the data (e. g., with the SMOTE algorithm [11]) did not significantly improve the results.

Instead of balancing the distribution of the data by sampling, we used cost-sensitive training to adapt the learning algorithm to the imbalanced data set: Cost-sensitive learning addresses imbalanced data by using cost matrices describing the costs for misclassifying any particular data example [10]. We used the WEKA CostSensitiveClassifier in combination with a decision tree to reweight the data points according to the cost matrix in Table II which was obtained by preliminary experiments. The matrix indicates that false negative mistakes are penalized five times more than false positive mistakes which makes the classifier more sensitive to the minority class of the training data. For more detailed information about cost-sensitive training, refer to [10] and the WEKA documentation.

VII. EVALUATION METHOD

We evaluate the cost-sensitive decision with k-fold cross-validation [12], using precision and recall. Precision and recall are calculated for each label l separately and then (weighted) averaged over all labels. As we use the experts' rating (*YES* or *NO*) the classification problem is binary.

Precision and recall. Pecision represents the probability that the classifier makes a correct decision when predicting that the instance is labeled with l. Recall denotes the probability that an instance with label l is detected by the classifier.

For this approach we are particularly interested in the precision for the positive label (*YES*): The algorithm's goal is to detect clones that contain incompletely fixed bugs. To achieve this, we are currently left with manual detection. For a quality engineer, finding incomplete bugs in an unfamiliar code base corresponds to inspecting code clones in a random order. Hence, the probability for the quality engineer to find an incomplete bugfix is equivalent to the relative frequency of incomplete bugfixes among all gapped clones. Hence, if the algorithm has a higher precision for the positive label than visiting clones in random order, it speeds up the process of retrieving incomplete bug fixes. For the scenario of a beginning continuous control process, precision is more important than recall to convince developers of the necessity of clone management. In our training data set, the relative frequency of incomplete bugfixes is 18.6% ($= 104/560$). Hence we aim for a precision above 19%.

Cross validation. The overall evaluation of the algorithm is based on k-fold cross validation, a standard evaluation technique in machine learning to determine the quality of a classifier [12]: The training data set is split into k subsets.

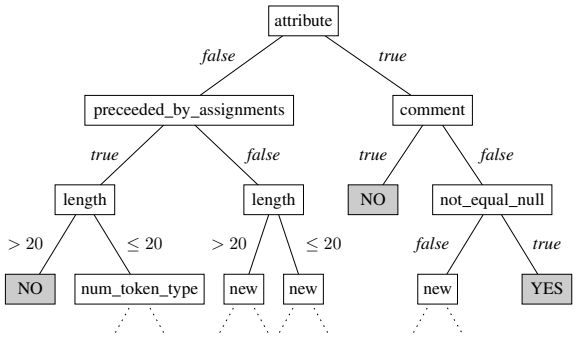

Fig. 3. Resulting classification model for bug detection

The classifier is built k times with $k-1$ out of k subsets as training data. For each run, the remaining subset is used as test data and the classifier's precision and recall are calculated on this fold. To get an overall evaluation, precision and recall are averaged over all k runs. Common values for k are 5 or 10. In this paper, we only present results of 5-fold cross validation as they did not differ significantly from 10-fold cross validation.

VIII. RESULTS

This section describes the results of our approach obtained by applying a decision tree algorithm on the real-world data set presented in Section III with the features described in Section IV. We first interpret the classification model learned and show precision, recall, and classification errors based on the results from cross-validation.

A. Classification Model

Figure 3 shows the top levels of the resulting decision tree. The complete tree with a maximum depth of ten is not included in this paper due to length restrictions. The visualization of the classification model provides insights about the most relevant features for bug identification. The closer a feature is located to the root of the decision tree, the more information gain it provides for clone classification. The feature used for the root is the local feature *attribute* indicating that at least one gap contains an attribute declaration. At the next depth levels, the tree uses the features *preceded_by_assignments* and *comment* (depth 1), *length* and *not_equal_null* (depth 2), and *new* and *num_token_types* (depth 3).

In the following, we present a simplified, top-level interpretation of the decision nodes: The existence of an attribute declaration in any gap indicates a non-faulty gap if the gap was commented. If no gap contained an attribute declaration, but a gap was preceded by variable assignments, the decision tree mostly decides for a faulty gap. If neither the *attribute* nor the *preceded_by_assignments* features are true, the decision tree splits the data based on the *length* and the *new* feature. In most of the cases, if a new object was created in any gap, the tree classifies the majority of data points as faulty.

B. Classification Evaluation

Table III shows the classification's confusion matrix. Rows represent the real label of the data, columns indicate the

TABLE III
CONFUSION MATRIX

Classified as →	*YES*	*NO*
YES	62	42
NO	218	238

TABLE IV
RESULTS OF J48 FOR CLONE CLASSIFICATION

Class	TP Rate	FP Rate	Precision	Recall
YES	0.596	0.478	0.221	0.596
NO	0.522	0.404	0.850	0.522
Weighted Average	0.536	0.418	0.733	0.536

label assigned by the classifier. Hence, entries on the matrix diagonal are correct classifications, other entries indicate classification errors. In total, the classifier labels 300 instances correctly (54% of all data), 62 clones with incomplete bugfixes and 238 clones without bug. The classifier did not detect 42 incomplete bugfixes and misclassified 218 clones without bug.

Table IV shows the classification's precision, recall, true-positive rate, and false-positive rate which are calculated for each label individually and summarized in a weighted average over all instances (weights for each label correspond to the relative frequency of the label within the training data). Precision is calculated as the number of true positives over true positives plus false positives, recall as number of true positives over true positives plus false negatives.

For clones with incomplete bugfixes, the algorithm achieves a precision of 22.1% ($= \frac{62}{62+218}$), better than the target precision of 18.6% (see Section VII). The recall of 59.6% ($= \frac{62}{62+42}$) indicates that the classifier detects more than every other incomplete bugfix. For clones without bug, precision is 85%, recall 52%. However, these metrics were not the primary target of our approach. In total, the classifier results in a weighted average precision of 73% and recall of 53%.

We also used a 10-fold cross validation and ran both the 5-fold and 10-fold cross validation multiple times with different random seeds but did not detect a significant change in the output.

IX. DISCUSSION

The results revealed that the machine learning approach achieves a higher precision for incompletely fixed bugs than manual detection in an unfamiliar code base where clones are inspected in random order and hence, the probability to find an incomplete bug fix corresponds to the relative frequency of incomplete bug fixes in the data set. However, our experiments also show limitations of feature-based approaches which are discussed in this section. First, we enumerate additional features that were not beneficial for bug prediction. Second, we show why not all information relevant for bug detection can be captured in features. Third, we discuss the challenges in obtaining a data base for classification.

A. Excluded Features

Initially, we experimented with more features than presented in Section IV. However, including some features in the data representation caused a lower performance of the classifier—a phenomenon frequently seen during machine learning. Nevertheless, it is valuable information which features do not provide information for detecting incompletely fixed bugs in clones. The following list describes our initial assumptions more in detail and explains the consequences when adding a single feature to the feature set.

- **gap (Integer)** – *Total number of gaps in a clone class.* The more the code fragments differ, the less likely the clone class contains a bug. If there are only few differences, the probability of a bug is higher. Similar assumptions have been stated in previous work [4], [8]. Contrary to our assumption, our experiments showed that using the *gap* instead of the *length* feature or both together lowers precision and recall for faulty inconsistencies.
- **if / else (Boolean)** – *Any gap contains the keyword* `if` *or* `else`.
  We assumed that a forgotten `if` or `else` statement indicates an incompletely fixed bug. Both features reduce the performance of the decision tree and do not seem to have significant information gain. The *else* feature does not even occur in the pruned version of the decision tree, the *if* feature only at high depths close to the leaves.
- **boolean_check (Boolean)** – *Any gap contains a == together with a boolean literal*
  Similar to the *not_equal_null* and *equal_null* features, we assumed that a forgotten condition in a `if`, `while`, or `for` statement can indicate incomplete bugfixes. Adding or removing this feature from the data representation does not influence the result significantly. It neither provides information gain nor causes the decision tree to make wrong decisions.
- **boolean_assign (Boolean)** – *Any gap contains an = assignment of a boolean literal*
  We assumed that forgetting to assign a boolean variable with control structures such as `loop`, `while`, or `if` often causes bugs when the boolean variable is used as a control condition. Adding this features lowers precision and recall.

B. Limitations of Features

Experimenting with different sets of features provides some insights about successes and limitations of a feature-based approach. The feature-based approach is well-suited to correlate syntactic properties of cloned fragments with the existence of incomplete bugfixes such as a missing `!= null` condition.

However, we observed that the presence of an incomplete bugfix strongly depends on implicit context information that cannot be expressed in features. Often, it is hard for any human, who is not an active developer of the code base, to judge whether an inconsistency is unintentional or error-prone.

For example, the presence of a comment preceding a gap may indicate two things: Either a bug was fixed and

commented but forgotten in the other cloned fragment—indicating an incomplete bugfix—or the developer cloned code and commented on intentionally added new functionality—indicating an intentional inconsistency. A human might be able to differentiate both cases by referring to the natural language information of the comment. A feature-based representation, however, fails to capture this difference. In case the comment contains signal words such as "bug", "fix", or "added", natural language processing can be useful for classification, but this does not generalize to an arbitrary commenting style.

The presence of the keywords `if` or `else` is another such example that can either indicate an incomplete bugfix (forgotten `if` condition in one code fragment) or the intentional adaptation of a clone fragment to a different scenario. In such cases, it is even for a human hard to make a decision about the correctness without deep knowledge of the code.

C. Challenge of Data Collection

The success of any machine learning approach depends on the training data used. In particular, we experienced that the imbalanced class distribution of our data hinders further success. With random downsampling, we achieved a precision for the positive label up to 68% in some cases. However, this performance varied so strongly with the random sample that it cannot be generalized. Nevertheless, the partial success of the approach makes us believe that using automated clone classification has potential for future work if a better (larger and balanced) data set of inconsistent clones is available.

X. Threats to Validity

This section summarizes various factors that threaten the general validity of our results. First of all, our choice of features was based on syntactic properties of C-like languages like Java, C++, and C#. Therefore, we cannot generalize our results to all programming languages.

Second, we mainly focused on the application of our approach in the domain of quality control to motivate the use of clone management. Hence, we argued that the precision of the positive label is the most important evaluation aspect. For other use cases, which include the retrieval of remaining bugs in the system, recall is also crucial. Future work should study features that improve the recall while maintaining precision.

Third, using experts' ratings as evaluation contains several threats already described in [2]. However, within this context, we rely on the experts' rating as no other evaluation method is available.

Furthermore, the configuration of the clone detection tool strongly influences detection results shown to the experts. Refer to the justification in [2], which is based on a pre-study and long-term experience with clone detection.

XI. Conclusion and Future Work

In this paper, we examined how much a feature representation of gapped clones can help to retrieve incomplete bugfixes in cloned code. We experimented with a broad range of features and used machine learning to determine their relevance in bug prediction. The machine learning classifier achieved a precision of 22% for clones with faulty inconsistencies which is better than manual inspection in random order.

For developers, our approach can be used as a recommendation tool for finding incomplete bugfixes to remove errors still remaining in the system. For the use-case of motivating clone management, our approach helps to quickly find faulty clones in our customers' code. We often observed that one or two examples of unintentional faulty inconsistent clones from the customers' own code can significantly support our argument to use clone management. In that sense, our approach is a first step to minimize the effort of retrieving convincing examples.

Our approach is a good starting point for future work as it has shown that feature-based bug detection and automated clone classification has a certain potential. The two most promising directions for future work are the creation of a larger data set for learning and the evaluation of other features. These may, for example, even include non-syntactic features like change frequency or authorship.

References

[1] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 311–320.

[2] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 485–495.

[3] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th International Conference on Program Comprehension*. IEEE Computer Society, 2008, pp. 172–181.

[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[5] Y. Higo, K. Sawa, and S. Kusumoto, "Problematic code clones identification using multiple detection results," in *Proceedings of the 16th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2009, pp. 365–372.

[6] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[7] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queens University at Kingston, Ontario, Canada, Technical Report, 2007.

[8] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2007, pp. 55–64.

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

[10] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. on Knowl. and Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.

[11] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002.

[12] R. Kohavi, "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, ser. IJCAI '95. Morgan Kaufmann, 1995, pp. 1137–1143.