STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLV**, Number 1, 2000

CLASSIFYING SOFTWARE BUG REPORTS USING METHODS FROM FORMAL CONCEPT ANALYSIS

DANIEL BORCHMANN, RAFAEL PEÑALOZA, AND WENQIAN WANG

Abstract. We provide experience in applying methods from formal concept analysis to the problem of classifying software bug reports characterized by distinguished features. More specifically, we investigate the situation where we are given a set of already processed bug reports together with the components of the program that contained the corresponding error. The task is the following: given a new bug report with specific features, provide a list of components of the program based on the bug reports already processed that are likely to contain the error. To this end, we investigate several approaches that employ the idea of *implications* between features and program components. We describe these approaches in detail, and apply them to real-world data for evaluation. The best of our approaches is capable of identifying in just a fraction of a second the component causing a bug with an accuracy of over 70 percent.

1. Motivation

Maintaining large software systems is a non-trivial task, and processing bug reports efficiently is a crucial part of this process. Modern software systems can easily contain thousands of lines of code, distributed over several modules and subsystems. When the system reaches such a size, and no single programmer can oversee its overall complexity, finding components of the program which are likely to contain the error that causes a given bug report becomes much more demanding. This is a known challenge in software development. For example, a recent study has shown that in average it takes 19 days for the Eclipse project and 38 days for the Mozilla project to find a first component assignment for a bug report [6], and there is no guarantee that this first assignment is correct. Finding the responsible component is a main bottleneck in the debugging process, and it may even require more time than fixing the error itself. In such cases, speeding up the process of identifying the responsible components would increase maintainability, and thus the quality, of the software.

The purpose of this work is to share some experimental experience we have obtained while trying to solve this problem. The approaches we follow in this work are all based on ideas from formal concept analysis. More precisely, we employed the idea

Key words and phrases. Formal Concept Analysis, Classification, Software Maintenance, Implications.

D. Borchmann supported by DFG Graduiertenkolleg 1763 (QuantLA). R. Peñaloza partially supported by DFG within the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfAED).

that the information contained in a bug report (its so-called "features") somehow determine in an *implicational manner* the component of the program containing the error. Therefore, we devised several methods based on the notion of *implications* in formal contexts to find such components, and tried to evaluate them experimentally on some real-world data obtained from bug reports in a large software company.

Obviously, one could argue here that the assumption that features of bug report determine the responsible components precisely is somehow simplified: it is not unlikely—and it is in fact not very hard to come up with an example for this—that two identical bug reports are caused by two completely unrelated errors in the software system. Clearly, this defeats our main assumption of an implicational dependency between features of bug report and responsible components. On the other hand, one could argue that the cause for such situation is that the bug reports are *underspecified*, and that the implicational dependencies between features of bug report and responsible components would still hold if we would include more features, which add information that can separate the two reports. This could be achieved by requesting more information from the user reporting the bug. However, even in the case where we do not request additional information, we can still use our assumption to find a *set* of likely components that caused the bug report, thus reducing the number of components which need to investigated.

The main practical problem we have to face when following the indicated approaches is to *find* the implicational dependencies between features of bug report and their responsible components. To cope with this difficulty, our approaches more or less follow a common pattern: all bug reports already processed so far are brought together in a formal context $\mathbb{K}_{\mathsf{reports}}$. This formal context is then examined for implicational dependencies between features and components. Then, if a new bug report, given as a set of features, is received, the implications extracted from the initial context are applied to this set of features, and the components contained in the resulting closure are considered *candidate causes* for the new bug report. Additionally, some of our approaches introduce a meaningful way of *rating* the candidate components according to their likelihood; that is, the higher the rank of a candidate component, the more likely it is that the new bug was caused in that component.

While this idea is relatively simple to describe and understand, it faces several practical issues. As already discussed, if the set of features does not determine responsible components uniquely, the standard approaches of extracting valid implications from the context $\mathbb{K}_{\mathsf{reports}}$ are not applicable. Hence, we have to devise new methods to achieve this goal, relaxing the restrictions that implications must satisfy. Furthermore, already processed bug reports do not always need to be correct; for example, it may happen that the actual cause of some historical report was never fixed, but rather that the circumstances of the bug were altered in such a way that it was not observed any more. In such cases, the component stored as cause for this error in the historical records is itself not correct. Assuming that such cases are possible but unlikely, we have to adapt our approaches to include methods that can handle those exceptional errors correctly. Finally, the context $\mathbb{K}_{\mathsf{reports}}$ itself can be quite large, and existing approaches to extract implications from contexts may simply not work on such large contexts, due to memory and time restrictions. Devising ideas for scalable extraction algorithms is thus also necessary in this setting.

Clearly, the problem of suggesting components that are likely responsible for bug reports is a classification problem in the sense of machine learning [11]. However, it is not the aim of this work, at this early stage of development, to compete with existing approaches in machine learning. Indeed, the purpose of this work is more to share experiences on how to approach this problem from the perspective of formal concept analysis, which we consider a natural, although often neglected, choice for this situation. A comparison with other existing classification approaches, or a combination with them, would be a logical next step in this direction of research. We leave that road open for possible future work.

This work is organized as follows. After giving a formal specification of our problem and some related work in Section 2, we introduce and discuss in Section 3 the approaches we investigate in this paper. Thereafter, we describe our experimental setup, show and discuss our results, and evaluate the individual approaches. This is done in Section 4. We close this paper with conclusions and outlook for further research in Section 5.

2. Problem Specification and Related Work

We start by describing the problem we want to solve in a more precise way. For the rest of this paper, we assume that the reader is familiar with the basic notions of formal concept analysis. More details from this area can be found in [5].

Let $\mathbb{K}_{\mathsf{reports}} = (G, M, I)$ be a finite formal context, which we call the *context of reports*, and let $M = F \cup C$ be a partition of M, i. e. $F \cap C = \emptyset$ and both F and C are non-empty. We call the elements of F *features*, and the elements of C *components*. Intuitively, we will understand $\mathbb{K}_{\mathsf{reports}}$ as the formal context of all *previous issues* (also called *old issues*, or *bug reports*) that have been reported for our software system. For every such issue $g \in G$, the elements of $g' \cap F$ are the *features* of the issue of g; more precisely, the information that was observed and reported when the user encountered the error. Possible such features can be statements like "segmentation fault" or "screen turned blue". On the other hand, the elements of $g' \cap C$ are the *responsible components* of the issue g, i. e. the elements of the software that were the reason for the issue g to exist, and were located when the old issue g was solved. In other words, fixing these components resulted in the issue to disappear.

Given such a formal context $\mathbb{K}_{\mathsf{reports}}$ and the partition $M = F \cup C$, we want to find for a given new issue (that is, for a set of features $\mathsf{o} \subseteq F$) a set of components which are "likely" to be responsible for it. To achieve this goal, we want to make use of the historical knowledge from the already solved issues collected in $\mathbb{K}_{\mathsf{reports}}$. Thus, we want to be able to *learn* from the old issues as a means to identifying the components that are responsible for a new issue.

If one sees this formalization of our problem, one may be reminded of a similar approach to model *learning from positive and negative examples* within the framework of formal concept analysis [8]. Within this approach we assume a formal context $\mathbb{L} = (H, N, J)$, and a *target attribute* $\omega \notin M$ which objects in H may or may not have. Let $H_+ \subseteq H$ be the set of objects which are known to have the attribute ω, $H_- \subseteq H$ the set of objects which are known to not have the attribute ω and let $H_? = H \setminus (H_+ \cup H_-)$ be the set of objects for which it is not known whether they have the attribute ω or not. Note that the three sets H_+, H_-, and $H_?$ are mutually disjoint. We call the elements of H_+ *positive examples* for ω, and likewise elements of H_- *negatives examples* for ω. The elements of $H_?$ are called *undetermined examples*.

The sets $H_+, H_-, H_?$ give rise to three subcontexts $\mathbb{L}_+, \mathbb{L}_-, \mathbb{L}_?$ of $\mathbb{L}$ defined as the restrictions of $\mathbb{L}$ to the corresponding sets of objects. The derivation operators of $\mathbb{L}_+, \mathbb{L}_-, \mathbb{L}_?$ are denoted by $(\cdot)^+, (\cdot)^-, (\cdot)^?$, respectively.

To decide for objects in $H_?$ whether they may have the target attribute ω or not, we extract *hypotheses* from $\mathbb{L}_+$ and $\mathbb{L}_-$. In this setting a *positive hypothesis* T for ω is an intent of $\mathbb{L}_+$ such that $T^+ \neq \emptyset$ and T is not contained in any object intent of $\mathbb{L}_-$, i. e. $T \nsubseteq g^-$ for all negative examples $g \in H_-$. *Negative hypotheses* are defined analogously. To decide for an undetermined example $g \in H_?$ whether it has the target attribute ω or not, we consider its object intent $g^?$ in the context $\mathbb{L}_?$. If this set contains positive hypotheses but no negative ones, then g is *classified positively*, and correspondingly, if $g^?$ contains negative hypotheses but no positive ones, g is *classified negatively*. If $g^?$ does not contain any hypotheses at all, then g is *unclassified*, and if $g^?$ contains both positive and negative hypotheses, then the classification of g is *contradictory*.

This method could also be applied to our problem of classifying software issues. In this case, we would consider every component we have as a target attribute, and try to apply the above method to obtain a classification. However, this idea becomes impractical as the number of components increases: for each component we would need to construct the contexts $\mathbb{L}_+, \mathbb{L}_-, \mathbb{L}_?$ and classify using the method sketched above, which is actually known to be hard [7]. This theoretical hardness may or may not be an issue in practical applications.

Furthermore, as already discussed, it may happen that bug reports having the exact same features, actually describe different errors in the software, and thus may have different responsible components. In those cases, we would still like to obtain a meaningful set of potentially responsible components (if possible, with an associated rating). However, the approach for learning from examples [8] would always result in an undetermined or contradictory classification.

Nevertheless, we can draw some inspiration from this approach for our own problem, and we do so in the following section, where we describe some methods for proposing responsible components for new issues.

3. Method Descriptions

We have tried several approaches for detecting the responsible components for a given issue. Each of these approaches is motivated by different ideas, which we describe in detail next. Their common property is that they all make use of a historical collection of old issues stored in the context $\mathbb{K}_{\mathsf{reports}}$ of reports to predict the component of a new issue. After having described these methods, in the next section we provide the results of an experimental evaluation on real-world issues from a software company.

object	a	b	c	X	Y
1	×	×		×	
2	×	×			×
3			×	×	
4	×	×	×	×	
5		×			×
6	×		×	×	

TABLE 1. The context $\mathbb{K}_{\mathsf{exa}}$ used as a running example

For the following descriptions we assume that the attribute set M of $\mathbb{K}_{\mathsf{reports}}$ is partitioned into features and components as described before, i. e. $M = F \cup C$. Furthermore, we assume that we are given a *new issue* $\mathsf{o} \subseteq F$ which we want to classify. For this, each of the following methods *proposes* a set $\mathsf{candidates}(\mathsf{o}) \subseteq C$ of the components that are likely to be responsible for the issue o. Furthermore, all but the first method additionally yield a score $\mathsf{score}(x) \in [0, 1]$ for each component $x \in \mathsf{candidates}(\mathsf{o})$. The higher this score, the more likely the method considers x to be responsible for o.

To help understanding the ideas behind all these methods, we will apply them over the simple context $\mathbb{K}_{\mathsf{exa}}$ shown in Table 1. In this context, the features are a, b, and c, while the components are X and Y. As the new issue to be classified we consider the set of features $\mathsf{o}_{\mathsf{exa}} = \{b, c\}$.

The `new-incident` method. A very simple idea for classifying a new issue would be to search in the historical records $\mathbb{K}_{\mathsf{reports}}$ for a previous occurrence of the same issue. The component that was responsible for the old issue can then be suggested as being responsible also for the new issue. This idea has two obvious problems. On one hand, the historical record is not necessarily complete, and hence there might exist no matching report; in this case, no responsible component would be suggested. On the other hand, since historical records may contain errors, components might change over time, and the set of features might not fully describe all possible problems, there might exist more than one matching issue in $\mathbb{K}_{\mathsf{reports}}$, which may lead to several components being proposed. To alleviate these issues, we slightly generalize this simple classification idea, yielding an approach which we call `new-incident`, which works as follows. Recall that the new issue is described by its set of features $\mathsf{o} \subseteq F$. For every object g in the context $\mathbb{K}_{\mathsf{reports}}$, if $g' \cap F \subseteq \mathsf{o}$, then $g' \cap C$ is suggested as a responsible component, i. e.

$$\mathsf{candidates}(\mathsf{o}) = \{x \in g' \cap C \mid g' \cap F \subseteq \mathsf{o}\}.$$

Note that there is no scoring among the candidates of o, i. e. all proposed components are equally preferred.

In our example context $\mathbb{K}_{\mathsf{exa}}$, for the new issue $\mathsf{o}_{\mathsf{exa}}$ we get that only the objects 3 and 5 are such that $g' \cap F \subseteq \mathsf{o}_{\mathsf{exa}}$: $3' \cap F = \{c\}$, and $5' \cap F = \{b\}$. The proposed components are then X and Y, and these are preferred equally.

This approach is in fact very similar to the one using hypotheses for classification, as we have described in Section 2. Namely, what we do here is to consider for all components $x \in C$ in $\mathbb{K}_{\mathsf{reports}}$ the sets of features belonging to issues in $\mathbb{K}_{\mathsf{reports}}$ with responsible component x. These sets actually correspond to hypotheses in the sense of Section 2. The only difference may be that for one such set of features T it may happen that T is actually contained in some set of features which belongs to a previous issue which had a responsible component different from x. Then, in the approach of Section 2 we would discard T as a hypothesis. However, as we have already argued previously, that is not a wanted behavior in our setting, as otherwise we would end up with a large number of contradictory classifications. Instead, we keep T as a hypothesis, and allow for a classification to more than one component. In this way, `new-incident` is similar to the classification of Section 2.

A drawback of the `new-incident` method is that the whole context needs to be processed whenever a new issue arises. As the historical records can be very large, this might be a very time-consuming task. Thus, we analyze methods based on the pre-computation of *bases* of implications to assist in a more efficient classification of issues.

The `can+lux` method. Recall that the reports context may contain contradictory information or may be incomplete. It thus makes sense to try to use a base capable of producing implications that are violated by a small number of exceptions, like Luxenburger's base [9, 10, 14]. The definition of this base relies on the notions of *support* and *confidence* of an implication [1]. Intuitively, the support describes the proportion of objects that satisfy the implication, while the confidence measures the number of objects that do not violate it.

Definition 1 (support, confidence)**.** Let $\mathbb{K} = (G, M, I)$ be a formal context and $A \subseteq M$. The *support* of A is

$$\mathsf{supp}(A) := \frac{|A'|}{|G|}.$$

The *support* and *confidence* of an implication $A \to B$ are defined, respectively, as

$$\mathsf{supp}(A \to B) := \mathsf{supp}(A \cup B), \qquad \mathsf{conf}(A \to B) := \frac{\mathsf{supp}(A \cup B)}{\mathsf{supp}(A)}.$$

Luxenburger's base includes only implications between intents having support and confidence larger than the given parameters $\mathsf{minsupp}$ and $\mathsf{minconf}$, respectively, which are input values from the interval [0,1] of real numbers, provided by the user. Moreover, the implications belonging to this base can only relate direct neighbors from the lattice of intents of the given formal context.

Definition 2 (Luxenburger's base)**.** For a finite formal context $\mathbb{K}$, the *Luxenburger base* of $\mathbb{K}$ w.r.t. $\mathsf{minsupp}, \mathsf{minconf} \in [0, 1]$ is the set of all implications $A \to B$ such that A and B are intents of $\mathbb{K}$, A is a direct lower neighbor of B in the lattice of intents of $\mathbb{K}$ ordered by $\subseteq$, and both (i) $\mathsf{conf}(A \to B) \geq \mathsf{minconf}$ and (ii) $\mathsf{supp}(A \to B) \geq \mathsf{minsupp}$ hold.

Notice that Luxenburger's base does not include implications that are valid in the formal context $\mathbb{K}$, because for two intents A, B of $\mathbb{K}$ to yield a valid implication $A \to B$ of $\mathbb{K}$, one must have $A = B$, and then A cannot be a direct lower neighbor of B anymore. To ensure that we do not miss implicational dependencies which are actually true in $\mathbb{K}_{\mathsf{reports}}$ we therefore have to take the valid implications separately, and we do so by extending Luxenburger's base with the *canonical base* of $\mathbb{K}_{\mathsf{reports}}$.

Given a new issue defined by a set of features o, the `can+lux` method computes the closure of o over the canonical and Luxenburger's bases of $\mathbb{K}_{\mathsf{reports}}$, and suggests all components appearing in this closure as candidates. Each candidate component $x \in C$ is associated with a score, defined as the maximum of the confidences of all rules $A \to \{\, x \,\}$ such that $A \subseteq \mathsf{o}$, i. e.

$$\mathsf{score}(x) := \max\{\mathsf{conf}(A \to \{x\}) \mid A \subseteq \mathsf{o}\}.$$

Note that this involves an exhaustive search among all subsets of o, and can hence become very expensive. However, for the experimental setup that we discuss in the next section this is not an issue, as the size of o is usually small.

Let us consider our example context $\mathbb{K}_{\mathsf{exa}}$ again. Its canonical base is

$$\{\, \{Y\} \to \{b\}, \{b, X\} \to \{a\}, \{c\} \to \{X\}, \{a, b, Y, X\} \to \{c\} \,\},$$

and the Luxenburger's base of $\mathbb{K}_{\mathsf{exa}}$ with $\mathsf{minsupp} = 0.01$ and $\mathsf{minconf} = 0.01$ consists of the implications

$$\begin{array}{cccc} \emptyset \to \{X\}, & \emptyset \to \{b\}, & \{b\} \to \{Y\}, & \{X\} \to \{c\}, \\ \emptyset \to \{a\}, & \{X\} \to \{a\}, & \{a\} \to \{X\}, & \{b\} \to \{a\}, \\ \{a\} \to \{b\}, & \{a, X\} \to \{b\}, & \{a, b\} \to \{X\}, & \{b, Y\} \to \{a\}, \\ \{a, b\} \to \{Y\}, & \{c, X\} \to \{a\}, & \{a, X\} \to \{c\}, & \{a, b, X\} \to \{c\}, \\ \{a, c, X\} \to \{b\}. & & & \end{array}$$

The closure of our observation $\mathsf{o}_{\mathsf{exa}} = \{b, c\}$ over these two bases includes both components X, Y, and hence both are proposed as responsible. Since the rule $\{c\} \to \{X\}$ is in the canonical base, X is proposed with score 1, while Y is proposed with score $\frac{2}{4}$, which is the confidence of the rule $\{b\} \to \{Y\}$.

The `can+lux` method provides a higher degree of liberty, as it is parameterized on the minimal support and minimal confidence that are used to compute Luxenburger's base. Moreover, the time required for computing the closure of the two bases and the scores of each proposed component is neglectable. Unfortunately, the same is not true for the computation of the bases. Indeed, as we will see in the following section, this computation was very costly in terms of time in our software issue scenario. Moreover, the performance of this classification was, surprisingly, rather disappointing.

Since the approach of considering Luxenburger's base turned out to be inappropriate, we studied different approaches for producing implications that are tolerant to a few exceptions. The main idea of the following three methods is to partition the context into smaller pieces, and compute only valid implications in these subcontexts. The intuition is that a small number of exceptions will violate such implications in only a few of all the subcontexts.

object	a	b	c
1	×	×	
3			×
4	×	×	×
6	×		×

object	a	b	c
2	×	×	
5		×	

TABLE 2. The subcontexts $\mathbb{K}_X$ (left) and $\mathbb{K}_Y$ (right) from the `subcontext` method

The `subcontext` method. For this method, we first create one subcontext $\mathbb{K}_x$ for every component $x \in C$ appearing in $\mathbb{K}_{\mathsf{reports}}$. The context $\mathbb{K}_x$ is defined as the restriction of $\mathbb{K}_{\mathsf{reports}}$ to the set of objects x' and thereafter removing all components and attributes which have an empty extent. In other words,

$$\mathbb{K}_x = (\bar{G} := x', \bar{F} := \{\, m \in F \mid m' \cap x' \neq \emptyset \,\}, I \cap \bar{G} \times \bar{F}).$$

Intuitively, the intents from the context $\mathbb{K}_x$ are sets of attributes that are always together whenever component x is responsible, and can hence be used as a premise for suggesting this component. To get rid of exceptions, we consider only implications whose premise have a support larger than a threshold, which is given as a parameter to the method. Formally, K is a *frequent intent* of a context $\mathbb{K}$ w.r.t. $\mathsf{minsupp}$ if it is an intent of $\mathbb{K}$ and $\mathsf{supp}(K) \geq \mathsf{minsupp}$. For every component x, and every frequent intent K of $\mathbb{K}_C$, we include the implication $K \to \{x\}$. Notice that every intent L that is a subset of a frequent intent K is also a frequent intent. Thus, it suffices to consider only the *minimal* frequent intents as premises for the implications. The proposed components are then

$$\mathsf{candidates}(\mathsf{o}) = \{x \mid K \text{ frequent non-empty intent of } \mathbb{K}_x, K \subseteq \mathsf{o}\}.$$

Note that this is the same as considering all components in the closure of o under all implications $K \to \{x\}$ where K is a frequent, non-empty intent of $\mathbb{K}_x$.

Consider again our example context $\mathbb{K}_{\mathsf{exa}}$. The two subcontexts $\mathbb{K}_X$ and $\mathbb{K}_Y$ are shown in Table 2. If we set the minimal support to $\mathsf{minsupp} = 0.1$, then the minimal frequent intents of $\mathbb{K}_X$ are $\{a\}$ and $\{c\}$, and the only minimal frequent intent of $\mathbb{K}_Y$ is $\{b\}$. Thus, we obtain the rules $\{a\} \to X$, $\{b\} \to Y$, and $\{c\} \to X$. Given the new issue $\mathsf{o}_{\mathsf{exa}} = \{b, c\}$, both components X and Y are suggested as potentially responsible for the issue.

To provide a more fine-grained suggestion of the responsible components, we score these implications according to their relevance among the context of reports. More precisely, for each component we set

$$\mathsf{score}(x) := \max\{\mathsf{conf}(K \to \{x\}) \mid K \text{ frequent non-empty intent of } \mathbb{K}_x\}.$$

object	a	b	c	X
1	×	×		×
3			×	×
6	×		×	×

object	a	b	c	X	Y
2	×	×			×
4	×	×	×	×	
5		×			×

TABLE 3. A partition $\mathbb{K}_1, \mathbb{K}_2$ of $\mathbb{K}_{\mathsf{exa}}$ for the `partition` and `partition-pp` methods

In our example, the scores are:

$$\mathsf{score}(X) = \max\{\mathsf{conf}(\{a\} \to X), \mathsf{conf}(\{c\} \to X)\} = 1,$$
$$\mathsf{score}(Y) = \max\{\mathsf{conf}(\{b\} \to Y)\} = \frac{2}{3}.$$

As a result, component X is suggested with a higher priority (namely, 1) than Y ($\frac{2}{3}$).

The `partition` and `partition-pp` methods. A different method for partitioning the context of all historical reports is to divide it in several subcontexts of equal size, regardless of the component they are associated with. Under the assumption that exceptions occur rarely, we expect these exceptions to violate the implications in only a few of the generated subcontexts. As the first step in the `partition` and `partition-pp` methods, we randomly partition $\mathbb{K}_{\mathsf{reports}}$ into contexts of a specified size n. These subcontexts are then simplified by removing all attributes that appear in no object. For instance, the context $\mathbb{K}_{\mathsf{exa}}$ can be partitioned into two contexts of size 3 as shown in Table 3.

Notice that in the first context we have removed the attribute Y, since it appears in no object of this context. Given a new issue o, the `partition` method computes, for every context $\mathbb{K}$ in the partition, the closure $\mathsf{o}''_{\mathbb{K}}$ of o over $\mathbb{K}$. The proposed components are those that appear in any of these closures; that is, we propose

$$\mathsf{candidates}(\mathsf{o}) = C \cap \bigcup\{\mathsf{o}''_{\mathbb{K}} \mid \mathbb{K} \text{ subcontext in the partition}, \mathsf{o}'_{\mathbb{K}} \neq \emptyset\} \quad (1)$$

as candidates for the observation, where $(\cdot)'_{\mathbb{K}}$ and $(\cdot)''_{\mathbb{K}}$ denote the derivation and double derivation operator in the corresponding context $\mathbb{K}$.

The score of each proposed component $x \in C$ is given by the proportion of subcontexts $\mathbb{K}$ in the partition such that $x \in \mathsf{o}''_{\mathbb{K}}$, i. e.

$$\mathsf{score}(x) := \frac{|\{\mathbb{K} \mid \mathbb{K} \text{ subcontext in the partition}, x \in \mathsf{o}''_{\mathbb{K}}\}|}{k}$$

where $k = \lceil \frac{|G|}{n} \rceil$ is the number of contexts in the partition.

The closure of our example observation $\mathsf{o}_{\mathsf{exa}} = \{b, c\}$ over the subcontexts $\mathbb{K}_1$ and $\mathbb{K}_2$ is $\{a, b, c, X\}$ in both cases. Thus, component X is proposed with score 1 (since it appears in the closure for all the subcontexts), and component Y is not proposed.

While the `partition` method behaves well in our scenario of software issues, as shown in the following section, one might still increase its accuracy by allowing more components to be suggested. The `partition-pp` method achieves this by considering the *proper premises* for the components in each subcontext, rather than a direct closure.

Definition 3 (proper premise)**.** Given a set of attributes $B \subseteq M$, we define $B^\bullet$ as

$$B^\bullet := B'' \setminus (B \cup \bigcup_{S \subsetneq B} S'').$$

B is called a *proper premise* if $B^\bullet \neq \emptyset$. It is a *proper premise of* $m \in M$ if $m \in B^\bullet$.

The idea of considering proper premises arises from the existence of the partition $M = F \cup C$ of the attribute set. More precisely, what we are actually interested in are implicational dependencies "from F to C," i. e. implications $A \to B$ satisfying $A \subseteq F$ and $B \subseteq C$. Then sets $\mathcal{L}$ of implications of this type are *iteration-free*, i. e. the computation of closures $\mathcal{L}(\bar{F})$ of sets $\bar{F} \subseteq F$ can be achieved by

$$\mathcal{L}(\bar{F}) = \bar{F} \cup \bigcup \{B \mid (A \to B) \in \mathcal{L}, A \subseteq \bar{F}\}.$$

In other words, the computation given by the right-hand side of this equation does not need to be iterated to compute $\mathcal{L}(\bar{F})$ [5].

We now want to compute bases of this type of implications for each subcontext in `partition` and to use them instead of $(\cdot)''$. Of course, one would like to have such a set to be as small as possible, and indeed proper premises provide a way to obtain such a base. In other words, the set

$$\{B \to B^\bullet \cap C \mid B \subseteq F \text{ is a proper premise for some element in } C\}$$

is a minimal iteration-free base for all implications from F to C [2, 5]. This motivates why we would want to consider proper premises. Note that proper premises allow for interesting optimizations with respect to their computation [13].

We put this idea into action as follows: for each subcontext in the partition of $\mathbb{K}_{\mathsf{reports}}$, the `partition-pp` method computes the proper premises of the components appearing in it. We only include those proper premises which have positive support withing this subcontext. For each such proper premise B for a component x, we collect the implication $B \to \{x\}$ into a set $\mathcal{L}$. Responsible components are then proposed by finding all collected implications $(B \to \{x\}) \in \mathcal{L}$ such that $B \subseteq \mathsf{o}$, and suggesting their associated components x. The score of suggesting x is given by the maximal confidence of an implication $(B \to \{x\}) \in \mathcal{L}$ such that $B \subseteq \mathsf{o}$, i. e.

$$\mathsf{score}(x) = \max\{\mathsf{conf}(B \to \{x\}) \mid (B \to \{x\}) \in \mathcal{L}, B \subseteq \mathsf{o}\}$$

where the confidence is computed in $\mathbb{K}_{\mathsf{reports}}$.

The proper premises for X in the context $\mathbb{K}_1$ from Table 3 are $\{a\}, \{b\}$, and $\{c\}$. In $\mathbb{K}_2$ there are no proper premises for Y and the only proper premise for X is $\{c\}$. The confidence of the implications $\{a\} \to \{X\}, \{b\} \to \{X\}$, and $\{c\} \to \{X\}$ over $\mathbb{K}_{\mathsf{exa}}$ is $\frac{3}{4}, \frac{2}{4}$, and 1, respectively. Thus, given our observation $\mathsf{o}_{\mathsf{exa}}$, only the component X is suggested with score 1, due to the implication $\{c\} \to \{X\}$.

We can expect `partition-pp` to return more candidates than `partition`, which is also confirmed by our experiments. This is due to the following reason: in (1), a candidate set $\mathsf{o}''_{\mathbb{K}}$ is excluded when $\mathsf{o}'_{\mathbb{K}} = \emptyset$, i. e. if no object in the subcontext has all the features in o. In other words, we always consider the whole set o in every such subcontext. However, it may still be the case that there exists a subset $\mathsf{p} \subseteq \mathsf{o}$ which meaningfully entails responsible components, in the sense that $\mathsf{p}'_{\mathbb{K}} \neq \emptyset$ and $\mathsf{p}_{\mathbb{K}} \neq \mathsf{p}''_{\mathbb{K}}$. Those sets p are ignored in `partition`, but not in `partition-pp`: If $x \in \mathsf{p}''_{\mathbb{K}}$, then there exists a proper premise as subset p for x with positive support, and thus `partition-pp` proposes x as a candidate.

4. Results and Discussions

We have implemented all methods described in the previous section on top of `conexp-clj`, a general-purpose library for formal concept analysis.[1] Our implementation has then been applied to data describing software issues, collected by a large German software company. We considered this data as a multivalued context. The original data had six features that received several different manifestations. As a first step, we scaled this context nominally, resulting in a formal context of size 2951×2973 with incidence density of roughly 0.002. Using this data, we have conducted the following experiments to measure the quality of these methods with respect to classifying bug reports: for varying $n \in \mathbb{N}_+$, we have randomly chosen a subcontext with n objects, removing all attributes which have empty extent in the corresponding subcontext. Then $\lfloor 0.9 \cdot n \rfloor$ of these items have been used to *train* the methods; i. e. formed the context of reports $\mathbb{K}_{\mathsf{reports}}$, and the remaining $\lceil 0.1 \cdot n \rceil$ data items have been used to *test* them. A test consisted in classifying the set of features of the data items, and comparing the proposed components with the known responsible component. For each fixed n, the whole procedure was repeated five times; that is, 5 different, randomly chosen subcontexts have been considered. We recorded the averages of all the values measured during each of these five executions.

To evaluate the testing data, and obtain a better evaluation of our proposed methods, we also implemented a *random* classifier. This method simply proposes a randomly chosen proportion of all the available components. The number of proposed components is determined by an input parameter. This method allows us to determine whether the components are uniformly represented in the data, and avoid giving special importance to over-represented components.

The testing of the methods was done in two steps, when applicable: first of all, every method proposes a set of components as being responsible for the issue at hand. We say that such a test is *positive* if and only if the original responsible component is among those proposed. In addition, we also measured the mean percentage of proposed components among all components in the data. In this way, we can discern methods that yield positive answers simply because they propose a large amount of components, from those that yield more informative answers. Most of our methods also graded the proposed components. For those methods we say that a test is *correct* if and only if the original responsible component is among the top-most proposed ones. Again, we also measure the mean percentage of the topmost proposed components.

[1]`http://github.com/exot/conexp-clj`

Table 4. Experimental Results

Method	n	train (ms)	test (ms)	positive	proposed	correct	proposed
`random`(0.2)	1000	7.25	26.32	20.81%	20.00%	–	–
`random`(0.2)	2000	15.12	24.87	19.09%	19.72%	–	–
`random`(0.5)	1000	4.85	16.64	51.38%	50.00%	–	–
`random`(0.5)	2000	22.63	30.65	51.16%	49.82%	–	–
`random`(0.9)	1000	12.58	26.21	90.50%	89.61%	–	–
`random`(0.9)	2000	13.37	33.40	87.00%	89.68%	–	–
`new-incident`	1000	0.02	19856.95	36.00%	3.11%	–	–
`new-incident`	2000	0.02	59371.25	44.50%	3.00%	–	–
`subcontext`(0.05)	750	3181598.62	11.98	69.33%	28.15%	30.67%	0.55%
`subcontext`(0.05)	1000	2841258.02	14.10	73.00%	29.94%	51.00%	0.54%
`subcontext`(0.01)	750	3355400.12	12.65	73.33%	25.48%	37.33%	0.50%
`subcontext`(0.01)	1000	2923139.74	15.09	72.00%	27.93%	41.00%	0.50%
`can+lux`(0.01,0.7)	1000	1375682.73	113.01	9.00%	0.05%	9.00%	0.05%
`can+lux`(0.01,0.7)	2000	3260189.07	219.74	8.50%	0.03%	8.50%	0.03%
`can+lux`(0.01,0.9)	1000	1359721.46	148.44	14.00%	0.07%	14.00%	0.07%
`can+lux`(0.01,0.9)	2000	3378045.62	199.46	7.50%	0.03%	7.50%	0.03%
`can+lux`(0.05,0.7)	1000	310803.29	0.24	0.00%	0.00%	0.00%	0.00%
`can+lux`(0.05,0.7)	2000	724341.14	0.13	0.00%	0.00%	0.00%	0.00%
`can+lux`(0.05,0.9)	1000	340270.58	119.62	5.00%	0.03%	5.00%	0.03%
`can+lux`(0.05,0.9)	2000	787725.99	0.09	0.00%	0.00%	0.00%	0.00%
`partition`(3)	1000	18961.75	193.25	33.94%	0.84%	30.00%	0.23%
`partition`(3)	2000	73898.59	519.63	49.13%	0.69%	47.00%	0.19%
`partition`(10)	1000	6161.62	109.95	34.00%	0.21%	34.00%	0.21%
`partition`(10)	2000	23895.38	268.83	46.00%	0.34%	45.00%	0.19%
`partition`(15)	1000	4943.22	109.95	36.00%	0.23%	34.06%	0.17%
`partition`(15)	2000	16468.57	245.98	41.69%	0.27%	40.56%	0.16%
`partition`(30)	1000	2488.06	94.70	40.00%	0.23%	40.00%	0.20%
`partition`(30)	2000	8529.07	218.02	44.50%	0.25%	43.50%	0.18%
`partition-pp`(3)	1000	89773.62	83.68	78.19%	31.12%	64.00%	0.50%
`partition-pp`(3)	2000	418524.89	217.82	88.38%	37.96%	71.00%	0.39%
`partition-pp`(10)	1000	142692.24	63.23	77.38%	7.32%	68.00%	0.52%
`partition-pp`(10)	2000	498504.77	151.32	82.19%	10.79%	72.22%	0.43%
`partition-pp`(12)	1000	182192.81	57.21	78.63%	7.30%	69.69%	0.49%
`partition-pp`(12)	2000	491516.69	120.77	81.66%	9.29%	70.84%	0.39%
`partition-pp`(15)	1000	267326.75	53.75	76.88%	7.08%	61.31%	0.51%
`partition-pp`(15)	2000	630232.06	109.49	80.91%	7.56%	70.91%	0.38%
`partition-pp`(30)	1000	1083661.64	38.19	66.63%	3.62%	59.00%	0.44%
`partition-pp`(30)	2000	2201360.65	85.13	80.94%	4.16%	71.56%	0.36%

The results of our experiments are shown in Table 4. This table includes the training and testing times, which however should be considered with care: the experiments were conducted in parallel on a 24 core machine, and the times measured are the overall execution times, not the ones per thread. Thus, the actual computation times could be lower than the ones stated in the table. However, we still included these numbers to give a feeling on how these methods perform, and these times provide at least an upper bound for this. Also note that we applied a timeout of 5 hours for each experiment, including repetitions.

From the experimental results we first see that the random classifiers behave as expected: if we choose randomly 20% of all components we have, then roughly 20% of the tests are positive; that is, the responsible component is among those proposed. The same is true for 50% and 90%. Thus, our data behaves mostly like random data with respect to our classification task.

With respect to this random selection, we can observe that even our simple approach `new-incident` performs much better: for $n = 1000$, only around 3% of the components are proposed while 36% of all tests were positive. This performance increases for $n = 2000$. However, while the training time is negligible (there is, in fact, no training phase), the testing time is quite high, and increases with the size of the data. This may render this approach difficult to apply in realistic scenarios, where the classification time is the real bottleneck.

Fortunately, only the `new-incident` method has such long testing times. In all the following approaches, the testing time is negligible. However, the price to pay for this are rather huge training times, which are sometimes even larger than the timeout used. On the other hand, in comparison to testing, training is usually conducted rarely, which means that huge training times can still be acceptable in practical applications.

The first method in this category we want to discuss is `subcontext`, which we applied with parameters 0.05 and 0.01 to our data. We see that the rate of positive tests is quite high, but also the percentage of components proposed. On the other hand, the scoring function provides a good method for further reducing the set of proposed components: only one out of each 200 components is rated with the highest score, and the correct answer is still provided in approximately half of the cases. However, the training times for this method were the largest among all the approaches we tested, by a broad margin. As an overall comparison with other methods, we conclude that the `subcontext` method is not the best suited for achieving a convincing classification.

The approach `can+lux`, based on combining the canonical and Luxenburger's bases, has an even worse performance, much to our surprise: although the proportion between the number of proposed components and positive, respectively correct, tests is comparatively good, the latter is too low to be of any use in our classification problem. Moreover, the rating provided by this method yields no improvement over the unrated classification. As the percentage of proposed components is almost the same, we can conclude that most components receive the same (highest) score. This behavior is not necessarily an intrinsic problem of the method, but could be attributed to a faulty choice of the scoring function. Notice, however, that the method proposes in average *less* than one component as the responsible one. Thus, the same behavior would be observed, regardless of the scoring function used.

This picture changes drastically when we come to the approaches based on partitioning the training data into smaller subcontexts. For `partition` we not only achieved rather high positive rates, but additionally, the number of proposed components was radically reduced. Interestingly, the rating provided by this method also behaves well in the sense that it keeps high rates of correct tests but reduces the number of proposed components. This is especially true if the partitioned contexts are very small (e. g. have 3 objects), but is also observable for larger contexts. Finally, the training times are practically irrelevant, and should scale rather nicely for even larger data sets. Notice that the training time depends linearly on the number of objects in the training data; if we additionally want to restrict to only the highest-ranking components, then the training time becomes $\mathcal{O}(n \log n)$ since an additional sorting step is needed.

While `partition` behaves relatively well, the proportion of positive tests remains always below 50%. It would clearly be nicer to increase this number, even if the rate of proposed components increases. This is achieved by introducing implicational dependencies as in `partition-pp`, where both the positive and correct rates are increased. The cost of this improvement is to propose more components in both cases, but the ratios between proposed and positive or correct rates are still very good. What is very surprising, though, is that the rating provided by this approach is working very effectively, reducing the number of proposed components by factors of 10 or more while keeping high rates of correct tests. This is especially true for $n = 2000$, and one can conjecture that this even improves for larger training sets. Moreover, we can also see that the larger the subcontexts we consider in our partition, the smaller the sets of proposed components are. However, we have to pay for this with an increase in the training time, which may or may not be relevant in practice. In particular, this method proposes in average less than six top-rated components, and it might not be worth spending many resources trying to reduce this number further.

The results of Table 4 are further depicted in Figure 1 for positive classification and Figure 2 for correct classification. In the figures, the horizontal axis corresponds to the percentage of positively or correctly classified tests, respectively, while the vertical axis shows the percentage of suggested components. Thus, the ideal situation is an element in the lower-right corner: a high percentage of success, while suggesting only a few candidates. In the plots, the different methods are depicted using different node shapes, while the shade of gray expresses the size of the training set: a darker shade means a larger set. As it can be seen, the nodes sharing the same figure and the same shade of gray form natural clusters in these plots. This suggests that the quality of the results depends mainly on the method chosen, and the size of training set, while the parameters used in the specific method are not that relevant. As described before, the best results were obtained by the `partition-pp` method with a training set of size 2000. This corresponds to the cluster of nodes depicted by ✸ in the plots. It can be easily seen that this method indeed showcases the best behavior. The only

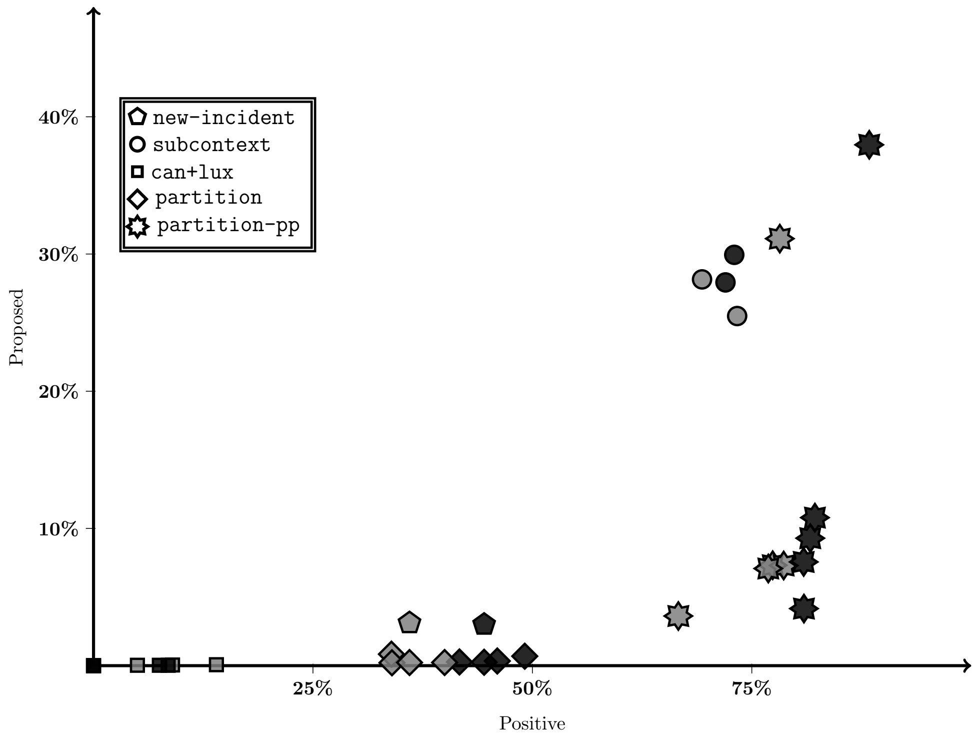

FIGURE 1. Proportion of positive vs. proportion of proposed components

exception is the case where partitions have size three, which is the single node in the upper-right corner of Figure 1. This case was the most successful w.r.t. positive classification, but at the cost of suggesting over a third of all available components.

These experimental results suggest that our initial idea of using implicational dependencies between attributes to classify bug reports is not completely unreasonable, however only if considered "locally" as it has been done in `partition` and `partition-pp`. If those dependencies are considered in the overall training data, then the resulting classification fails miserably, as shown by the results for `can+lux`. This also suggests that the partitions used in `partition` and `partition-pp` should not contain too large, nor too small, contexts.

For putting these methods into practice, we can also think of a combined approach of both `partition` and `partition-pp`: the former one has an acceptable performance and suggests only very few components. Therefore, considering those components may be a good starting point. If it turns out that the responsible component is not among those proposed by `partition`, one can resort to considering those proposed by `partition-pp`, which may be more (especially if not rated), but which are also more likely to contain the real cause of the issue. Here, also different sizes of the partition are imaginable, increasing the performance of the classification but also the number

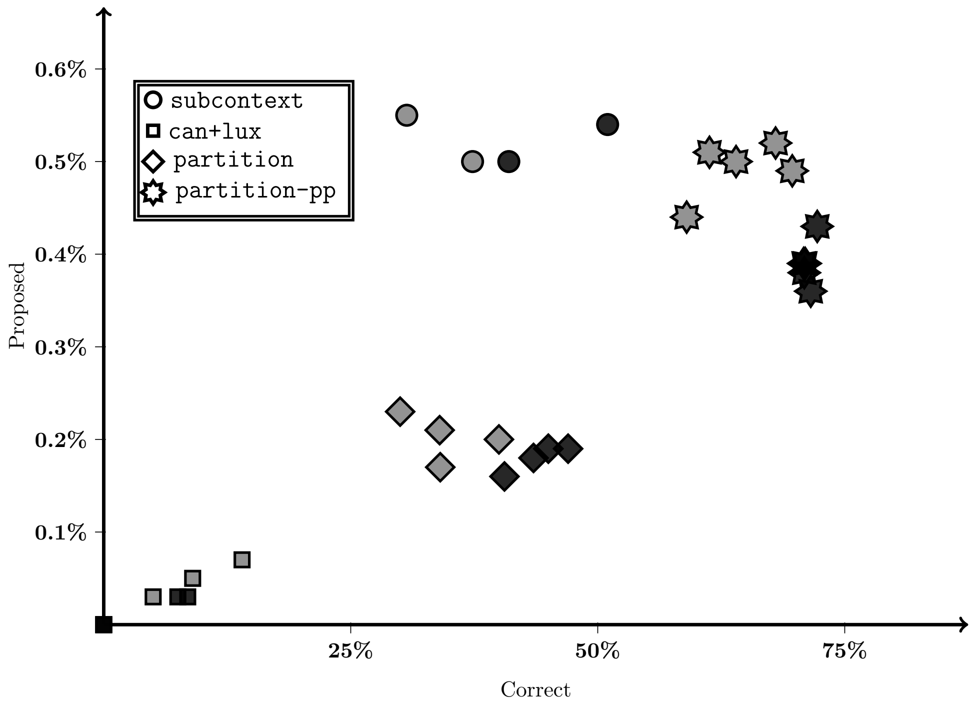

FIGURE 2. Proportion of correct vs. proportion of proposed components

of proposed components. Of course, if all this fails, one has to fall back to manual classification. However, this last resource is needed only sporadically.

5. Conclusions

The goal of our study was to analyze whether formal concept analysis tools can be useful for classifying software issues according to their responsible component. Contrary to standard machine learning techniques, formal concept analysis methods provide logical implications between the symptoms (features of the bug) and the causes (the responsible component). These implications can be understood by users, and provide more detailed information of the software system itself.

The use of association rules to detect faults and vulnerabilities in software systems has been studied previously [3, 4, 12]. The main difference with this paper is that we study and compare different approaches for handling erroneous and incomplete information, and detected empirically which is best suited for our software bug classification scenario.

We have tried several approaches, all based in ideas developed in the area of formal concept analysis. Each of the methods presented was inspired by different approaches towards the problem. In particular, one of the important issues was how to deal with potential errors, incomplete knowledge, and change of the software structure over time. Surprisingly to us, the obvious idea of using Luxenburger's base to handle uncommon exceptions yielded relatively bad results: the responsible component was usually not among those proposed, regardless of the chosen minimal support and confidence.

The method that behaved best in our scenario was to compute a base of proper premises over a partition of the historical records, together with a scoring function for the proposed components. This method behaves very well from partitions of size 3 up to 30, yielding the right answer in over two-thirds of the cases, while proposing less than 0.5% of the available components. This method also scales well: whenever new historical records are added, only the proper premises over a partition of the new cases need to be computed. All previous records remain unchanged. Moreover, it is easy to get rid of old historical records, by simply deleting their corresponding partitions.

In general, our experimental results show that it is feasible to classify objects from large historical records using formal concept analysis, provided that training can be done in an off-line stage. Indeed, while training in the `partition-pp` method could take more than 10 minutes, in a context of 1000 objects, the classification time was almost instantaneous, taking less than 100ms. For our software issue scenario, these conditions are satisfactory: new issues would be entered to the training data sporadically, and training may take place over-night. However, lower classification times, with higher success rates and small sets of candidate components, translate into faster repair of software bugs.

We have not compared our approach with existing classification methods from machine learning and other areas. Given that we obtained promising results with our approach, we will make such comparison in future work. We also notice that our implementation is based on prototype tools, and requires further optimization for industrial-strength use. Studying some applicable optimization techniques will also be a focus of future work.

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. "Mining Association Rules between Sets of Items in Large Databases". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 1993, pp. 207–216.

[2] Karell Bertet and Bernard Monjardet. "The multiple facets of the canonical direct unit implicational basis". In: *Theoretical Computer Science* 411.22-24 (2010), pp. 2155–2166.

[3] Peggy Cellier. "Formal concept analysis applied to fault localization". In: *Companion Volume of the 30th International Conference on Software Engineering.* (Leipzig, Germany). Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. ACM, 2008, pp. 991–994. ISBN: 978-1-60558-079-1.

[4] Peggy Cellier et al. “Formal Concept Analysis Enhances Fault Localization in Software”. In: *Proceedings of the 6th International Conference on Formal Concept Analysis.* (Montreal, Canada). Ed. by Raoul Medina and Sergei A. Obiedkov. Vol. 4933. Lecture Notes in Computer Science. Springer, 2008, pp. 273–288.

[5] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Berlin-Heidelberg: Springer, 1999.

[6] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. “Improving bug triage with bug tossing graphs”. In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering.* (Amsterdam, The Netherlands). Ed. by Hans van Vliet and Valérie Issarny. ACM, 2009, pp. 111–120.

[7] Sergei O. Kuznetsov. “Complexity of learning in concept lattices from positive and negative examples”. In: *Discrete Applied Mathematics* 142.1-3 (2004), pp. 111–125.

[8] Sergei O. Kuznetsov. “Machine Learning and Formal Concept Analysis”. In: *Proceedings of the Second International Conference on Formal Concept Analysis.* (Sydney, Australia). Ed. by Peter W. Eklund. Vol. 2961. Lecture Notes in Computer Science. Springer, 2004, pp. 287–312. ISBN: 3-540-21043-1.

[9] Michael Luxenburger. “Implications partielles dans un contexte”. In: *Mathématiques, Informatique et Sciences Humaines* 29.113 (1991), pp. 35–55.

[10] Michael Luxenburger. “Implikationen, Abhngigkeiten und Galois-Abbildungen”. German. PhD thesis. TH Darmstadt, 1993.

[11] Thomas M. Mitchell. *Machine Learning.* 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[12] Stephan Neuhaus and Thomas Zimmermann. “The Beauty and the Beast: Vulnerabilities in Red Hats Packages”. In: *Proceedings of the 2009 USENIX Annual Technical Conference.* 2009.

[13] Uwe Ryssel, Felix Distel, and Daniel Borchmann. “Fast algorithms for implication bases and attribute exploration using proper premises”. In: *Annals of Mathematics and Artificial Intelligence* Special Issue 65 (2013), pp. 1–29.

[14] Gerd Stumme et al. “Intelligent Structuring and Reducing of Association Rules with Formal Concept Analysis”. In: *KI 2001: Advances in Artificial Intelligence, Proceedings of the Joint German/Austrian Conference on AI.* (Vienna, Austria). Ed. by Franz Baader, Gerhard Brewka, and Thomas Eiter. Vol. 2174. Lecture Notes in Computer Science. Springer, 2001, pp. 335–350.

Theoretical Computer Science, TU Dresden, Germany
E-mail address: daniel.borchmann@mailbox.tu-dresden.de

Theoretical Computer Science, TU Dresden, Germany. Center for Advancing Electronics Dresden
E-mail address: penaloza@tcs.inf.tu-dresden.de

SAP, Germany
E-mail address: wenqian.wang@sap.com