Mining Explicit Rules for Software Process Evaluation

Chengnian Sun
School of Computing
National University of Singapore
suncn@comp.nus.edu.sg

Jing Du
Lab for Internet Software Technologies
Institute of Software
Chinese Academy of Sciences
dujing@nfs.iscas.ac.cn

Ning Chen
School of Computer Engineering
Nanyang Technological University
nchen1@ntu.edu.sg

Siau-Cheng Khoo
School of Computing
National University of Singapore
khoosc@comp.nus.edu.sg

Ye Yang
Lab for Internet Software Technologies
Institute of Software
Chinese Academy of Sciences
yangye@nfs.iscas.ac.cn

ABSTRACT

We present an approach to automatically discovering explicit rules for software process evaluation from evaluation histories. Each rule is a conjunction of a subset of attributes in a process execution, characterizing why the execution is normal or anomalous. The discovered rules can be used for stakeholder as expertise to avoid mistakes in the future, thus improving software process quality; it can also be used to compose a classifier to automatically evaluate future process execution. We formulate this problem as a contrasting itemset mining task, and employ the branch-and-bound technique to speed up mining by pruning search space. We have applied the proposed approach to four real industrial projects in a commercial bank. Our empirical studies show that the discovered rules can precisely pinpoint the cause of all anomalous executions, and the classifier built on the rules is able to accurately classify unknown process executions into the normal or anomalous class.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software process models*

General Terms

Management

Keywords

Software Process Evaluation, Contrasting Rule Mining

1. INTRODUCTION

A software process is a set of activities, policies, practices and procedures, which is used in a software organization to develop, deploy and maintain software and the associated artifacts (e.g., requirement, design documents, source code, test cases, bug repositories) [9]. The maturity of a software process is the extent to which the process is explicitly defined, managed, measured and executed. As such, it is directly related to the productivity of the organization and the quality of the software products [10].

Software process evaluation is a task conducted in an organization to measure the maturity of software processes within this organization, in order to further identify and assess its weakness, strength, and areas improved or to be improved. The evaluation procedure is usually done by an internal or external software process expert against a reference model based on the data collected by means of questionnaires, interviewing with team-members, checking project artifacts, etc. [13]. Then the organization can base on the evaluation result to make action plans for future improvement [4].

However, as stated in [3], existing approaches for the software process evaluation task have the following main limitations. First, the evaluation procedure is usually manual, and thus can be time-consuming for large projects. Second, due to authority constraint, external experts conducting the evaluation usually do not have full access to every collection of data; consequently, the evaluation results may not be precise. Third, the evaluation is often subjective, especially relying on the expertise of experts in specific software processes and related knowledge in the organization. Therefore, it will be useful if we can automate the evaluation without involving different levels of experts.

Chen et al.[3] proposed a machine learning approach to semi-automating the software process evaluation task. Generally, they assume that the software organization possesses adequate process execution history data, from which a classifier can be learnt for evaluating future process executions. In detail, they target at history data of sequence form, namely, each process execution is recorded as a sequence of state transitions labeled *normal* or *anomalous*; and employ off-the-shelf classification algorithms (i.e. C4.5, Naive Bayes and SVM) to perform sequence classification. Their experiments show the effectiveness of their approach.

However, the learnt classifier represents a set of implicit evaluation rules, which can be hard for people to understand, and cannot be *further* used to guide and improve process execution in the future. For example, the classification model learnt by SVM with a linear kernel is a weight vector, of which each dimension weighs the importance of the corresponding feature of a process execution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSSP '13, May 18–19, 2013, San Francisco, USA

Copyright 13 ACM 978-1-4503-2062-7/13/05 ...$15.00.

Given that there are 100 features in [3], a 100-dimension weight vector is not friendly for humans to interpret why a process execution is classified into a class.

Different than the machine learning-based approach in [3], we take a mining stance to discover explicit rules from history data. We stick to the same assumption that the organization has performed several process evaluations before, and owns adequate history data (i.e. both process execution information and evaluation results are preserved). Next, a set of features is extracted from each process execution as its representation. After processing all executions, we obtain a database of feature sets, with each set labeled as either *normal* or *anomalous* indicating whether the corresponding process execution violates process regulations or not.

We formulate the discovery of explicit rules as mining of *contrasting itemset patterns* from the database. Informally, a contrasting itemset pattern is a set of features which is observed in only either *normal* or *anomalous* process executions. Given such a pattern $\{a_1, a_2, \cdots, a_i\}$ *only* included in executions of type $label$ (i.e. *normal* or *anomalous*), we generate a rule in the following way:

$$a_1 \wedge a_2 \wedge \cdots \wedge a_i \Rightarrow label$$

Namely, if the feature representation of a process execution includes the attributes $a_1, a_2, \cdots$ and a_i, then the evaluation result of this execution is $label$.

Compared to the learnt classifier in [3], each of our mined rules is explicit and human-readable. Therefore stakeholder can investigate and draw conclusions from these rules and make future action plans to improve the maturity of software processes in their organizations. Furthermore, a set of rules can be composed into a classifier, fulfilling the same task of designating *normal* or *anomalous* label for future process execution, as the one described in [3].

We summarize our contributions as follows:

1. We propose a pattern mining-based approach for software process evaluation. Each pattern is a rule explicitly stating why a process execution is classified as *normal* or *anomalous*. We formulate the pattern as a contrasting itemset, appearing exclusively in either *normal* or *anomalous* executions.

2. We devise a simple but effective metric called contrasting significance (CS) to mine such rules. We also derive an upper bound for CS and employ the branch-and-bound technique to speed up the mining by aggressively pruning search space.

3. We have applied the proposed technique to four real-world projects in a commercial bank. We first manually check the expressiveness and understandability of the mined rules. With post-processing and a little human effort, the mined rules can be further summarized into several concise rules, which precisely pinpoint the cause of *anomalous* process executions. We also build a classifier from the rules to classify new execution into *normal* or *anomalous* class, and the performance is comparable with the best one (i.e. SVM) in [3].

The paper is organized as follows. Section 2 briefly introduces the concept of software process evaluation. Section 3 formulates the problem and details the technique to discover evaluation rules. Section 4 describes the case studies we have conducted on the interpretability of mined rules and their performance to classify new execution traces. Section 5 surveys related work, and Section 6 concludes this paper.

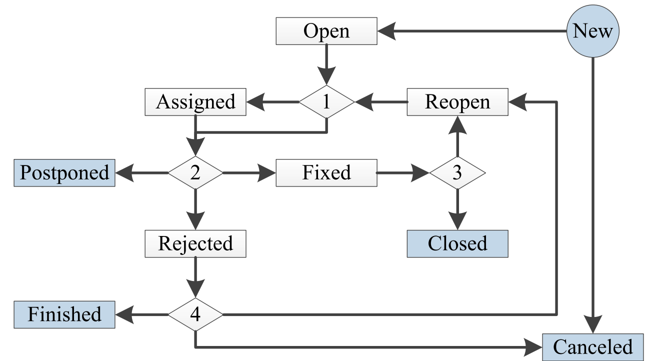


Figure 1: State Transition Flow of Defect Management Process

2. SOFTWARE PROCESS EVALUATION

Software process evaluation is a task conducted in an organization to measure the performance of software processes in a specific scope. Usually, the organization has a formal process specification (e.g. finite state machines, flow diagrams) specifying the expected behaviors of software process executions. Figure 1 shows the state transition flow of the defect management process specified in a commercial bank. A bug report starts with the state *New*, and ends at any of the four states, *Closed*, *Finished*, *Postponed* or *Canceled*.

Table 1: Two Examples of Process Execution Traces

ID	Label	Process Execution Trace
1	*anomalous*	⟨*New, Open, Fixed, Reopen, Fixed, Reopen, Fixed, Reopen, Fixed, Reopen, Fixed, Closed*⟩
2	*normal*	⟨*New, Open, Fixed, Postponed, Closed*⟩

However, there may be divergence between the formal specification and the evaluation result of process executions in two cases. First, an execution conforming to the specification is regarded as *anomalous*, The first process execution in Table 1 shows an execution trace of the defect management process, which follows the specification in Figure 1 exactly, but considered as *anomalous* as the verification of the bug fix fails multiple times.

Second, a process execution which does not follow the formal specification may not be considered harmful. For example in the last row of Table 1 – a normal execution example extracted from a bug repository, a developer fixed a bug, then the developer decided to postpone the bug to the next release of the software. In the next release, the bug fix was verified by the tester, and tagged as *Closed*.

Considering two examples above, we conclude that the software process evaluation task is not a simple process conformance checking problem, and propose the solution based on contrasting itemset mining, presented in the following section.

3. APPROACH

Figure 2 displays the framework of our approach. Generally, the input to our approach is two classes of process execution traces, with each class labeled *normal* (or *anomalous*) representing normal (or anomalous) process executions respectively. Next, for each trace, we extract a set of features to represent the trace. After the extraction for all traces is done, we identify features which are observed in only one class but never in the other via contrasting itemset mining algorithm, as rules for process evaluation. Lastly, these

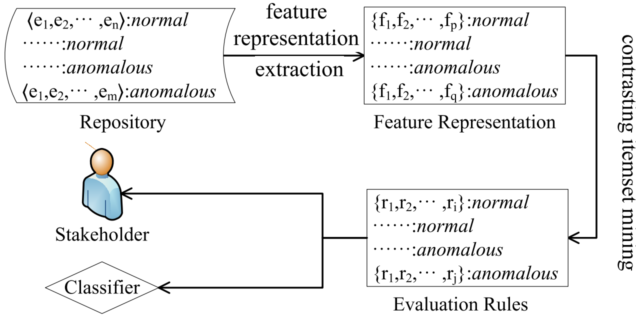


Figure 2: Framework of Our Approach

rules can be used for stakeholder as reference to improve software process quality, or used to construct a classifier to evaluate future process execution.

The rest of this section first formulates the problem of process evaluation rule mining, then details the features we use to represent process execution traces, and finally describes the branch-and-bound technique used to prune search space during mining.

3.1 Problem Formulation

Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of distinct features extracted from software process executions, $\mathcal{C} = \{+, -\}$ be the set of class labels identifying *normal* and *anomalous* executions ("+" for *normal*, "$-$" for *anomalous*), and $\mathcal{D}$ be a database consisting of n transactions (i.e. process executions) $\{(T_1, c_1), \cdots, (T_i, c_i), \cdots, (T_n, c_n)\}$, where $T_i \subseteq \mathcal{I}$ and $c_i \in \mathcal{C}$. For convenience, we define two filtering functions $+$ and $-$ for a set of transactions S,

$$S^+ = \{(T, c) \in S | c = +\}$$

$$S^- = \{(T, c) \in S | c = -\}$$

For example, $\mathcal{D}^+$ (or $\mathcal{D}^-$) denotes all the *normal* (or *anomalous*) transactions in $\mathcal{D}$ respectively. For a feature set (or itemset, pattern interchangeably) $P \subseteq \mathcal{I}$, we define $tx : 2^{\mathcal{I}} \rightarrow 2^{\mathcal{D}}$, returning all transactions in $\mathcal{D}$ containing pattern P.

$$tx(P) = \{(T, c) \in \mathcal{D} | P \subseteq T\}$$

The support of P is defined as the number of transactions containing P, i.e. $|tx(P)|$ denoted as $sup(P)$; moreover, $sup^+(P) = |tx(P)^+|$ and $sup^-(P) = |tx(P)^-|$. The support of a feature set satisfies the following property stated in [1].

PROPERTY 1 (**APRIORI**). *Given a feature set $P \subseteq \mathcal{I}$, $\forall P' \subseteq \mathcal{I}$, if $P' \supset P$, then*

$$sup^+(P') \leq sup^+(P)$$

$$sup^-(P') \leq sup^-(P)$$

A pattern is deemed *contrasting* if it appears in one class of transactions, but never in the other. Namely, a contrasting feature set should be observed in either *normal* or *anomalous* class of process executions. Let $p = sup^+(P)$, $n = sup^-(P)$, then the contrasting significance of the pattern P is defined in Equation 1,

$$CS(p, n) = \begin{cases} p & \text{if } p \neq 0 \wedge n = 0 \\ n & \text{if } p = 0 \wedge n \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Given a pattern P and $CS(sup^+(P), sup^-(P)) \neq 0$, we say the label of P is *normal* if $sup^+(P) > 0$, otherwise *anomalous*. Then given such a pattern $\{a_1, a_2, \cdots, a_n\}$ with label l (abbreviated as $\{a_1, a_2, \cdots, a_n\} : l$), we can interpret it as a process evaluation rule in the following:

$$a_1 \wedge a_2 \wedge \cdots a_n \Rightarrow l$$

That is, if a process execution is observed to contain all the features $\{a_1, \cdots, a_n\}$, then this execution is classified as either *normal* or *anomalous* indicated by the pattern label l.

DEFINITION 1 (**TOP-K CONTRASTING ITEMSET MINING**). *Given a database $\mathcal{D}$, its feature set $\mathcal{I}$ and an integer k, top-k contrasting itemset mining returns a set $\{p_i \subseteq \mathcal{I}\}_{i=1}^k$ from $\mathcal{D}$ such that:*

- $\forall i \in [1, k]$, $CS(sup^+(p_i), sup^-(p_i)) > 0$
- *maximizes* $\sum_{i=1}^k CS(sup^+(p_i), sup^-(p_i))$

Our explicit rules for software process evaluation is defined based on Definition 1. We construct a database from the evaluation histories by extracting features from each process execution as its representation, perform contrasting itemset mining and use the mined patterns as evaluation rules.

3.2 Feature Representation Extraction

This section describes how we extract features from a process execution. In this paper, we focus on one type of software process exectuions, which can be represented as sequences, e.g., software defect management process, software requirement management process. Given a software process P, let Σ denote all the possible states of P, then an execution of this process can be represented as $seq = \langle e_1, e_2, \cdots, e_n \rangle$ where $\forall i \in [1, n] : e_i \in \Sigma$. Given an integer $i \in [1, n]$, $seq[i]$ refers to the i-th element e_i in seq. To facilitate our explanation, we define the auxiliary functions in Figure 3.

Equation 2 returns the set of states in seq. Equation 3 tests the existence of a transition in seq. Equation 4 counts the occurrences of state e in seq. Equation 5 counts the occurrences of transition $\langle e'_1, e'_2 \rangle$ in seq. Equation 6 returns the last state of the given sequence.

Next we extract the four features listed in Figure 4 from a process execution. The function $absence$ captures the states which are not visited in the given execution. The second feature set $vertex$ records the occurrences of each state. Similar to $vertex$, $transition$ records the occurrences of transitions between two states. The last function end captures the final state of an execution. Finally, the feature set extracted from seq is $absence(seq) \cup vertex(seq) \cup transition(seq) \cup end(seq)$. Table 2 shows the four feature sets extracted from the anomalous process execution trace in Table 1.

Note that in this paper, we only consider these four sets of features, but it is possible to devise more features and our mining algorithm is sufficiently generic to process other types of features if only each feature can be represented as an item.

3.3 Branch-And-Bound Mining

The search space of contrasting itemset mining constitutes a lattice. Each node is a subset of $\mathcal{I}$, the bottom node is $\emptyset$ and the top node is $\mathcal{I}$. The partial order relation of the lattice is proper subset inclusion. Namely, an edge $n_1 \rightarrow n_2$ from the bottom to the top represents $n_1 \subset n_2$. The pattern mining starts with the bottom node $\emptyset$, and gradually visits nodes above the current lattice level,

$$S(seq) = \{e \in \Sigma | \exists i \in [1,n] \text{ s.t. } seq[i] = e\} \quad (2)$$

$$\langle e_1', e_2' \rangle \sqsubseteq seq = \begin{cases} true & \text{if } \exists i \in [1,n) \text{ s.t. } e_1' = seq[i] \wedge e_2' = seq[i+1] \\ false & \text{otherwise} \end{cases} \quad (3)$$

$$times(e, seq) = |\{i \in [1,n] | seq[i] = e\}| \quad (4)$$

$$times(\langle e_1', e_2' \rangle, seq) = |\{i \in [1,n) | seq[i] = e_1' \wedge seq[i+1] = e_2'\}| \quad (5)$$

$$last(seq) = seq[n] \quad (6)$$

Figure 3: Auxiliary Functions for Feature Extraction

$$absence(seq) = \{(e,0) | e \in \Sigma \backslash S(seq)\} \quad (7)$$

$$vertex(seq) = \{(e,i) | e \in S(seq) \wedge i \in [1, times(e, seq)]\} \quad (8)$$

$$transition(seq) = \{(\langle e_1, e_2 \rangle, i) | \langle e_1, e_2 \rangle \sqsubseteq seq \wedge i \in [1, times(\langle e_1, e_2 \rangle, seq)]\} \quad (9)$$

$$end(seq) = \{(last(seq), -1)\} \quad (10)$$

Figure 4: Features for Process Evaluation

Table 2: Features Extracted from the Anomalous Execution Trace in Table 1

Name	Extracted Features
absence	$(Assigned, 0)$, $(Postponed, 0)$, $(Finished, 0)$, $(Canceled, 0)$, $(Rejected, 0)$
vertex	$(New, 1)$, $(Open, 1)$, $(Fixed, 1)$, $(Fixed, 2)$, $(Fixed, 3)$, $(Fixed, 4)$, $(Fixed, 5)$, $(Reopen, 1)$, $(Reopen, 2)$, $(Reopen, 3)$, $(Reopen, 4)$, $(Closed, 1)$
transition	$(New, Open, 1)$, $(Open, Fixed, 1)$, $(Fixed, Reopen, 1), \cdots, (Fixed, Reopen, 4)$, $(Reopen, Fixed, 1), \cdots, (Reopen, Fixed, 4)$, $(Fixed, Closed, 1)$
end	$\{(Closed, -1)\}$

until it reaches the top $\mathcal{I}$ or no more patterns of interest is available.

However, the size of search space is exponential to the number of items $\mathcal{I}$ (i.e., $2^{|\mathcal{I}|}$) and thus we employ branch-and-bound (BAB) technique to aggressively prune the search space without loss of soundness. Generally, during mining, each time we take a branch and explore for interesting patterns from small to large. But if we can predict that along this branch, there will be no more *contrasting* patterns, we can safely stop this branch and move to the next one. The following describes how we decide to stop exploring a branch.

Assume that we know a set of transactions $ux(P) \subseteq tx(P)$ which contains all super patterns $P' \supseteq P$ where $sup(P') > 0$, referred to as *unavoidable* transactions.

$$ux(P) = \bigcap_{P' \supseteq P \wedge sup(P') > 0} tx(P')$$

The super pattern P' (s.t. $sup(P') > 0$) may not only be the immediate parent pattern of P, (i.e., $P' = P \cup \{e\}$ where $e \in \mathcal{I}$), but also any super pattern with a transitive closure computation over P (i.e., $P' = P \cup S$ where $S \subseteq \mathcal{I}$).

The unavoidable transactions of a pattern may be empty if all qualified super patterns share no common transactions. But in other cases that $ux(P) \neq \emptyset$, this notion can provide valuable information for estimating the upper bound of CS for super patterns of P, which is essential in BAB search. Based on the Apriori in Property 1, we get the following theorem.

THEOREM 1 (**UPPER BOUND OF** CS). *Given a pattern P, let $p = sup^+(P)$, $n = sup^-(P)$, $up = |ux(P)^+|$ and $un = |ux(P)^-|$, then the contrasting significance of all its qualified super patterns is upper bounded by the following formula:*

$$UB(P) = \begin{cases} n & \text{if } up = 0 \wedge un \neq 0 \\ p & \text{if } up \neq 0 \wedge un = 0 \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

PROOF. Given any qualified super pattern P' of P, let $p' = sup^+(P')$, and $n' = sup^-(P')$. Based on Property 1 and the definition of unavoidable transactions, we have $up \le p' \le p$, and $un \le n' \le n$. For the first case, if $up = 0 \wedge un \neq 0$, we have $CS(p', n') \le CS(0, n') \le CS(0, n) = UB(P)$.

The same reason applies to the second case. For the last case, as neither p' nor n' can be 0, $CS(p', n') = 0$. □

This upper bound can be employed in BAB search for top-k contrasting itemset pattern mining, based on the following property.

PROPERTY 2 (**UPPER BOUND-BASED PRUNING**). *Given an itemset pattern P and a contrasting significance score $CS^* > 0$, if*

$$UB(P) < CS^*$$

then $\forall P' \supseteq P : CS(sup^+(P'), sup^-(P')) < CS^*$

For top-k contrasting itemset mining, during mining we maintain a list containing the most k contrastingly significant patterns in the visited patterns, and update this list with newly discovered patterns. The minimum CS in this list is denoted as CS^*. Based on the property above, if we see that $UB(P) < CS^*$, we can stop exploring the search space along the current path, and switch to another path in the lattice.

We use the generic framework proposed in [19] with our customized measure CS and its upper bound to mine contrasting itemsets. For more information about the search space of itemset mining, the way to compute unavoidable transactions, and the implementation of BAB, we refer the readers to [19].

4. CASE STUDIES

We have built a prototype of the proposed technique named MOSPER (Mining of Software Process Evaluation Rules) in C++, and have evaluated it with the same dataset[1] as [3], 2622 sequences of defect report histories from four projects, which are extracted from the repository of the test management system HP Quality Center version 9.0 used in a commercial bank in China.

Table 3: Summary of Dataset

Project	#Defects
Electronic Commercial Draft System	1019
Wealth Management System (Phase 1)	478
Wealth Management System (Phase 2)	665
Financial Leasing System (Phase 2)	460

Table 3 displays the project names and the number of defects of each project collected in our case studies. All experiments are carried out on a Debian Linux PC with Intel Core 2 Quad CPU 3.00 GHz and 8 Gb memory.

The case studies consist of two experiments. In the first experiment, we mined all the rules, manually interpreted them, and checked their correctness. In the second one, we built a classifier based on the rules to classify un-labeled process execution into either *normal* or *anomalous* class, and compare the performance with the best classifier in [3].

4.1 Interpretation of Mined Rules

We applied MOSPER to the whole dataset with a very large k in order to get all contrasting rules, 2021 in total. Then the rules are separated into two lists based on their labels, and the rules in each list are sorted in descending order of their CS. Next for each list L, we scan it from the beginning and perform the pre-processing listed in Algorithm 1 to get L' with redundant rules removed.

Algorithm 1 Preprocessing Mined Rules

Input: L, a list of sorted rules for one class
Output: L', a list of rules without redundancy

1: $\mathit{Covered} = \emptyset$
2: $L' = []$
3: **for each** rule $r \in L$ **do**
4:   **if** $tx(r) \not\subseteq \mathit{Covered}$ **then**
5:     $L' = L' \cup \{r\}$
6:     $\mathit{Covered} = \mathit{Covered} \cup tx(r)$
7:   **end if**
8: **end for**

We get 25 rules for *anomalous* class, which cover all (638 out of 638) *anomalous* process executions; and 14 rules for *normal* class, covering 187 out of 1984 correct process executions. Table 4 shows the top 5 anomalous rules. The first rule shows that a bug report cannot be *open* twice. The second rule indicates that a report cannot be closed twice without any *postponed* or *finished* actions. The third shows that a report cannot be reopened twice. The fourth rule shows that after rejected, a report should be reopened before it is closed in the future. The last shows that after reopening a report, we should verify the fix first before we close it.

We interpret all the anomalous rules, and summarize them into 7 categories for comprehension purpose.

[1]Available at `http://www.cais.ntu.edu.sg/~nchen1/SPE.htm`

Table 4: Anomalous Example Rules

Rank	Rule
1	$\{(\mathit{Open}, 2)\}$:*anomalous*
2	$\{(\mathit{Closed}, 2), (\mathit{Postponed}, 0), (\mathit{Finished}, 0)\}$:*anomalous*
3	$\{(\mathit{Reopen}, 2)\}$:*anomalous*
4	$\{(\mathit{Closed}, 1), (\mathit{Rejected}, 1), (\mathit{Reopen}, 0)\}$:*anomalous*
5	$\{(\mathit{Reopen}, \mathit{Closed}, 1)\}$:*anomalous*

1. A bug can only be *open* once, and we should use *reopen* for the rest of times.

2. In order to close a bug, we need to fix it first.

3. A bug should not be *reopen* more than once.

4. Once a bug is *open* or *reopen*, some operations must be performed to it before it reaches *close* or *reopen*, e.g., *fixed* and *rejected*.

5. We cannot directly close a bug right after it is assigned.

6. In order to fix a bug again, there should be *reopen*.

7. The states of a bug after it is *open* but before it becomes *reopen* should contain only one of {*postponed*, *rejected*, *fixed*}.

These rules are double checked with the persons, who originally conducted the manual evaluation of all the software process executions. And it is confirmed that they were indeed implicitly used in the evaluation task. Thus we conclude that our mining algorithm can effectively identify explicit software process evaluation rules. Note that these rules are organization-dependent, and new rules mined from execution histories in different organizations may vary.

4.2 Classification

Basically, each rule is a set of features with a class label in the form $\{f_1, f_2, \cdots, f_n\} : label$, then we can build a classifier such that for each testing instance t, if it contains all the features of a rule i.e. $\{f_1, f_2, \cdots, f_n\} \subseteq t$, then we predict t as class $label$. As we have shown in the previous section that we can get rules covering all *anomalous* executions, but only a portion of *normal* ones, we use the *anomalous* rules to build a classifier, and if a testing instance cannot be classified, we simply label it as *normal*. The performance of the classifier is evaluated in terms of the following three metrics.

$$\begin{aligned} \mathit{Precision} &= \frac{\mathit{TP}}{\mathit{TP} + \mathit{FP}} \\ \mathit{Recall} &= \frac{\mathit{TP}}{\mathit{TP} + \mathit{FN}} \\ \mathit{F\text{-}measure} &= \frac{2 \times \mathit{Precision} \times \mathit{Recall}}{\mathit{Precision} + \mathit{Recall}} \end{aligned}$$

where *TP*, *FP* and *FN* refer to the number of true positives, false positives and false negatives respectively. In the experiments, we take *normal* as positive class and *anomalous* as negative class.

We first perform 10-fold validation on the dataset (randomly partitioning the dataset into 10 pieces, choosing one piece as testing set and the rest as training set, and repeating 10 times) and compare with the result reported in [3].

Table 5 shows the performance comparison. The second to fourth rows are the classification results achieved by C4.5, Naive Bayes classifier and SVM reported in [3], and the last row is by our mined

Table 5: Performance Comparison of 10-Fold Validation

	Precision	Recall	F-measure
C4.5	92.9%	94.7%	93.8%
NB	92.9%	91.1%	92.0%
SVM	97.6%	99.6%	98.6%
MOSPER	100%	99.5%	99.8%

Table 6: Statistical Detail of *Precision*

Percentage	Median(%)	Mean(%)	Standard Deviation
10	98.5	98.2	0.012
20	99.4	99.3	0.005
30	99.6	99.5	0.004
40	99.8	99.7	0.003
50	99.9	99.8	0.002
60	100	99.9	0.002
70	100	99.9	0.001
80	100	99.9	0.001
90	100	99.9	0.002

rules. As the performance scores of SVM and MOSPER are very close, we conclude that MOSPER is comparable with SVM.

We also test the impact of the size of training set on the classification performance. We randomly select 10%, 20%, $\cdots$, 90% of the dataset as the training set and test the built classifier against the remaining executions. To mitigate the randomness, all experiments are repeated 100 times, and all performance metrics are averaged for comparison.

Figure 5 displays the performance comparison in terms of precision, recall and F-measure. As indicated, the performance improves with the size of the training set. Furthermore, even with 30% training set, the classification performance (precision = 99.6%, recall = 98.8%, F-measure = 99.2%) is already comparable with SVM which uses 90% of the dataset as training set.

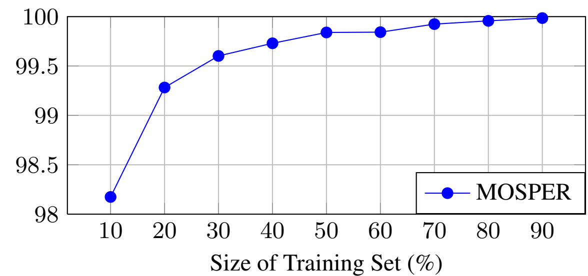
Table 6, Table 7 Table 8 show the median, mean and the standard deviation of the *Precision*,*Recall* and*F-Measure* respectively for training sets of different sizes. The standard deviation for each metric is very small, thus we conclude that the average performance of our approach is stable.

In the classifier above, we use both the normal and anomalous rules for classification. We also build another classifier using solely *anomalous* rules. They perform quite similarly. With 90% dataset, its average precision is 100%, average recall is 99.36% and average F-measure is 99.67%.

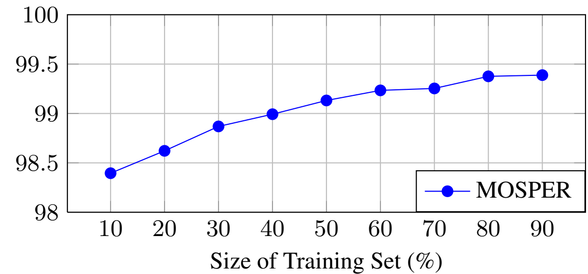
4.3 Efficiency

Figure 6 shows the runtime (in $\times 10^{-2}$ milliseconds) to mine evaluation rules from training set of various sizes ranging from 10% to 100% of the whole data set. The mining algorithm is very efficient, as no experiment takes more than one millisecond. The runtime is linear to the number of transactions in the dataset, based on the plots in Figure 6.

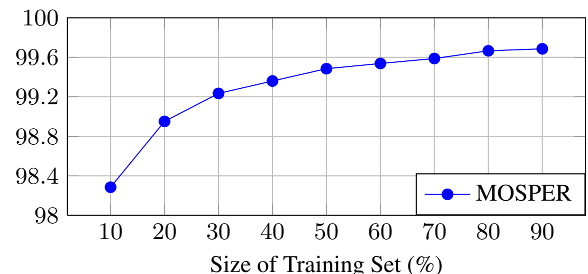
However the time complexity of our technique is theoretically exponential to the number of distinct features extracted from process evaluation history, i.e., $|\mathcal{I}|$. As aforementioned, the mining of contrasting itemsets is a search over a lattice, of which the bottom is $\emptyset$, the top is $\mathcal{I}$, and the relation between two nodes is subset relation $\subset$. Thus there are $2^{\mathcal{I}}$ nodes in the search space. As each node represents a pattern – a set of features, we need to explore all these patterns in the worst case. Hence, the worst time complexity is $O(2^{\mathcal{I}})$.



(a) Precision



(b) Recall



(c) F-Measure

Figure 5: Performance Comparison with Training Set of Various Sizes

Fortunately, with effective pruning technique such as estimation of upper bound of CS, we can avoid exploring the whole space. This is why the runtime of our technique is neglectable as shown in Figure 6, even though there are 120 features in $\mathcal{I}$ of the whole data set. Therefore, we conclude that under the current feature extraction scheme in this paper, our algorithm is scalable to the size of the dataset.

4.4 Threats to Validity

There are mainly three threats to validity. First, it is unknown whether our approach can generalize to software process evaluation tasks in other organizations, although the performance on the four commercial projects in a commercial bank is promising. However, our approach assumes that process executions of *normal* and *anomalous* classes should be differentiable to human experts with certain rules, and this assumption is general to other evaluation tasks, thus we believe the performance on other organizations should not deviate much.

Second, when the mined evaluation rules are applied to classify future process executions into a class, the classification performance may degrade if the training set is not adequate. This can be shown by Figure 5. There is a 0.004 difference in precision

Table 7: Statistical Detail of *Recall*

Percentage	Median(%)	Mean(%)	Standard Deviation
10	98.6	98.3	0.009
20	98.7	98.7	0.004
30	98.9	98.8	0.004
40	99.0	99.0	0.004
50	99.2	99.2	0.003
60	99.3	99.2	0.003
70	99.3	99.3	0.004
80	99.5	99.4	0.004
90	99.4	99.3	0.006

Table 8: Statistical Detail of *F-Measure*

Percentage	Median(%)	Mean(%)	Standard Deviation
10	98.4	98.3	0.007
20	99.0	99.0	0.002
30	99.2	99.1	0.002
40	99.4	99.3	0.002
50	99.5	99.5	0.002
60	99.6	99.6	0.002
70	99.7	99.6	0.002
80	99.7	99.7	0.002
90	99.7	99.7	0.003

and 0.007 difference in recall between 30% and 90% training sets. Although the difference is ignorable, our approach with small training set may still not work well. Therefore, it is desirable to use our technique when the organization already has a certain amount of process evaluation data.

Third, the feature engineering scheme may not be effective for all kinds of software processes. For more complex processes than that in this paper, our features may become less discriminative to distinguish *normal* and *anomalous* classes. Thus it will be necessary to devise more features to characterize traces besides these used in this paper. However, our mining algorithm is generic to directly accept new features without any modification.

5. RELATED WORK

This work is inspired by the one done by Chen et al. in [3]. They transform each process execution into a vector with each dimension corresponding to a k-gram of the alphabet set. In their case studies, they use bigrams to construct dataset and test the classification performance of various algorithms and conclude that SVM is the best. However, the model learnt by the classification algorithm is implicit, that is, it is not easy to understand the meaning of the classification rules in the model. For example, a linear SVM model is a weight vector, of which each dimension weighs the importance of the corresponding feature (i.e. a bigram in [3]). Given that there are 100 features in [3], it is not easy to identify the reason why a process execution is classified as *anomalous* just based on a 100-dimension weight vector. A decision tree model learnt by C4.5 [15] is more human-readable than an SVM model. However, due to the training method of C4.5, its classification performance is not as good as SVM in process execution classification as reported in [3]. Consequently the rules represented by C4.5 are not as precise as ours, since our algorithm performs comparably with SVM. In this work, we take a mining stance to identify the contrasting features between *anomalous* and *normal* executions. The mined patterns serve as *explicit* rules which are much easier to understand. Our explicit rules can further be naturally used to build a classifier, and the performance is comparable with SVM in terms of precision, recall and F-measure.

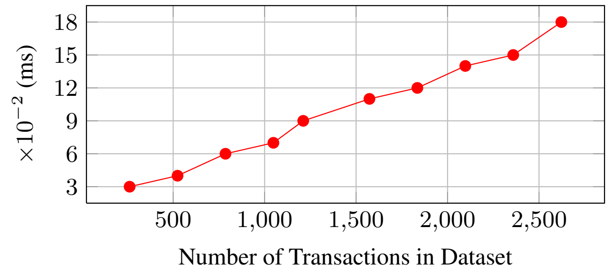


Figure 6: Runtime of Different-sized Dataset

Another line of related research is software process conformance checking, which focuses on measuring the difference between the formal process model and process executions [7, 6]. Our work differs from those approaches in that we aim to mine explicit and human-readable evaluation rules from two classes (*normal* or *anomalous*) of process execution histories, whereas they compare the real process executions with the formal specifications. Our work is also related to software process model discovery by mining event logs in software repositories. Rubin et al. present the applications of process mining techniques to mine software development processes in [16]. Samalikova et al. [17] propose a mining algorithm based on ProM framework [20] to construct change control board process model from event logs in software configuration management systems, and compare the mined model against the formal process model, in order to provide feedbacks to the development team. Similar to theirs, our mined rules can also serve as feedbacks to improvement process quality. Yet differently, instead of mining process models, our approach mines explicit evaluation rules by contrasting *normal* and *anomalous* process executions. And these rules serve as explanations why a process execution is classified as *normal* or *anomalous*.

Other studies on software process methodologies and models are also relevant. They specify a set of guidelines and practices for assessing and improving the general software process capability of large and small software development organizations [4, 8, 14, 21, 12]. As mentioned in Section 1, these approaches usually require human effort and expertise, which could be time-consuming, subjective and imprecise. And our work complements them.

Our work is also related to discriminative model-based classification for software engineering, as the concept of identifying contrasting patterns is similar to discriminative analysis for classification. In [11], Lo et al. propose an automatic approach to classifying executions into correct or faulty sets. They first mine discriminative sequence patterns from execution histories as features to contrast faulty and correct executions, and then represent each execution trace as a binary vector, of which each dimension indicates the occurrence of a distinct discriminative pattern. Finally, an SVM model is learnt to predict the failure status of future execution. In [18], Sun et al. employ discriminative model approach to analyzing the difference between a pair of duplicate bug reports and a pair of non-duplicate reports, and train an SVM model with the difference for duplicate bug report retrieval task. Anvik et al. [2] propose a technique to triage new bug reports to appropriate developers via text categorization algorithms. In [5], Clenland-Huang et al. propose a model to classify none functional requirements from structured and un-structured documents.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a mining-based technique to identify explicit rules for software process evaluation. These rules can serve as guidelines for improving software process maturity in an organization, or further compose a classifier to automatically evaluate future software process executions. We define the problem of identifying these rules as contrasting itemset mining, propose a metric CS to quantify the contrasting significance of a rule, and derive a tight upper bound for CS for aggressive pruning of rule search space.

In the future, we plan to directly generate explanations of mined rules in natural languages. Currently each evaluation rule is a conjunction of several features, and we need to manually translate them. We believe this will promote the usability of our technique further. Another possible future direction of this work is to identify evaluation rules from non-structured process execution data, as a considerable proportion of software process execution data is non-structured, e.g., document revisions, weekly progress reports, and code revision histories, and it contains uncovered wealthy information for process evaluation tasks.

Acknowledgment

We are grateful to the reviewers for their valuable comments. This work is supported by a research grant R-252-000-403-112 at National University of Singapore.

7. REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, 1994.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who Should Fix This Bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[3] Ning Chen, Steven Chu-Hong Hoi, and Xiaokui Xiao. Software Process Evaluation: A Machine Learning Approach. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, 2011.

[4] Mary Beth Chrissis, Mike Konrad, and Sandy Shrum. *CMMI Guidelines for Process Integration and Product Improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[5] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. The detection and classification of non-functional requirements with application to early aspects. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, RE '06, pages 36–45, Washington, DC, USA, 2006. IEEE Computer Society.

[6] Jonathan E. Cook and Alexander L. Wolf. Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. *ACM Trans. Softw. Eng. Methodol.*, 8:147–176, 1999.

[7] Marcos Aurelio Almeida da Silva, Xavier Blanc, and Reda Bendraou. Deviation Management During Process Execution. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 528–531, 2011.

[8] Khaled El Emam. *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[9] Alfonso Fuggetta. Software Process: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 25–34, 2000.

[10] M. S. Krishnan, C. H. Kriebel, Sunder Kekre, and Tridas Mukhopadhyay. An Empirical Analysis of Productivity and Quality in Software Products. *Manage. Sci.*, 46(6):745–759, June 2000.

[11] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 557–566, 2009.

[12] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[13] Leon J. Osterweil. Software Processes Are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 540–548, 1997.

[14] Francisco J. Pino, César Pardo, Félix García, and Mario Piattini. Assessment Methodology for Software Process Improvement in Small Organizations. *Inf. Softw. Technol.*, 52(10):1044–1061, 2010.

[15] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[16] Vladimir Rubin, Christian W. Günther, Wil M. P. Van Der Aalst, Ekkart Kindler, Boudewijn F. Van Dongen, and Wilhelm Schäfer. Process Mining Framework for Software Processes. In *Proceedings of the 2007 international conference on Software process*, ICSP '07, pages 169–181, 2007.

[17] Jana Samalikova, Rob Kusters, Jos Trienekens, Ton Weijters, and Paul Siemons. Toward Objective Software Process Information: Experiences from a Case Study. *Software Quality Control*, 19(1):101–120, 2011.

[18] C. Sun, D. Lo, X. Wang, J. Jiang, and S-C. Khoo. A Discriminative Model Approach for Accurate Duplicate Bug Report Retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 45–54, 2010.

[19] Chengnian Sun, Siau-Cheng Khoo, David Lo, and Hong Cheng. Efficient and Accurate Mining of Succinct Bug Signatures. `http://www.comp.nus.edu.sg/~suncn/mbs/paper.pdf`, 2012.

[20] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, and W. M. P. van der Aalst. The Prom Framework: a New Era in Process Mining Tool Support. In *Proceedings of the 26th international conference on Applications and Theory of Petri Nets*, ICATPN'05, pages 444–454, 2005.

[21] Christiane Gresse von Wangenheim, Alessandra Anacleto, and Clenio F. Salviano. Helping Small Companies Assess Software Processes. *IEEE Softw.*, 23(1):91–98, January 2006.