iDice: Problem Identification for Emerging Issues

Qingwei Lin Jian-Guang Lou Hongyu Zhang Dongmei Zhang
Microsoft Research, Beijing 100080, China
{qlin, jlou, honzhang, dongmeiz}@microsoft.com

ABSTRACT

One challenge for maintaining a large-scale software system, especially an online service system, is to quickly respond to customer issues. The issue reports typically have many categorical attributes that reflect the characteristics of the issues. For a commercial system, most of the time the volume of reported issues is relatively constant. Sometimes, there are emerging issues that lead to significant volume increase. It is important for support engineers to efficiently and effectively identify and resolve such emerging issues, since they have impacted a large number of customers. Currently, problem identification for an emerging issue is a tedious and error-prone process, because it requires support engineers to manually identify a particular attribute combination that characterizes the emerging issue among a large number of attribute combinations. We call such an attribute combination *effective combination*, which is important for issue isolation and diagnosis. In this paper, we propose iDice, an approach that can identify the effective combination for an emerging issue with high quality and performance. We evaluate the effectiveness and efficiency of iDice through experiments. We have also successfully applied iDice to several Microsoft online service systems in production. The results confirm that iDice can help identify emerging issues and reduce maintenance effort.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging - diagnostics; D.2.8 [**Software Engineering**]: Management - software quality assurance

Keywords

Emerging issues, problem identification, effective combination, problem diagnostic, issue reports

1. INTRODUCTION

Despite immense efforts spent on software quality assurance, various types of failures often occur in software systems in actual operations. Whenever customers encounter a problem, they can report the issue to the technical support team of the software system. For a widely-used, large-scale software system, the support team could receive a large number of issue reports from customers over a period of time. Furthermore, in order to enable quick response to changing market requirements, today's software systems evolve quickly - developers constantly update existing features, add new features, and release new versions within a relatively short time. Therefore, support engineers often encounter new issues reported by customers. The maintenance task could be even more challenging for large-scale online service systems (such as Microsoft Office 365, Windows Azure, and Visual Studio Online) [20, 7], which are attracting growing number of customers and are updated more frequently.

A typical customer issue report consists of many categorical attributes such as product version, the problematic product feature, product configuration, client OS, service package, country, etc. Each attribute has a set of distinct values. An issue report also includes a time stamp recording the time at which the report was received. Therefore, the issue report data can be treated as multi-dimensional, time series data.

Oftentimes, it is observed that the volume of issue reports under a certain attribute combination could suddenly increase significantly at a certain point of time. Such a "burst" may be due to major feature changes, configuration mistakes, environmental incidents, or software bugs. We call these issues *emerging issues*. For example, in June 2013, a bug in a new update of a Microsoft online service caused a sudden increases of issue reports related to a certain attribute combination. This upward trend of issue reports continued for several days and incurred significant increase of support cost. The emerging issues have negative impact on a large number of customers, therefore, they should be assigned a high priority for fixing. Otherwise, more customers could be affected and more issue reports could be received. In order to detect emerging issues, the support engineers need to identify a particular attribute combination that can characterize the issues. The attribute combination helps isolate the problem, find the root cause of the issue, and bring the system back to normal. We call such an attribute combination *effective combination* since it characterizes an emerging issue.

In current practice, among the Microsoft support engineers we have talked with, the identification of effective combinations is largely performed manually, e.g., they use Excel Pivot Table to look for the candidate attribute combina-

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884795

tions that may indicate the significant upward trend in the number of issue reports. Clearly, when the number of attributes and attribute values are many, support engineers need to search through a large number of possible attribute combinations in order to identify the effective ones reflecting emerging issues. Detecting emerging issues could be very difficult, because the burst of an emerging issue can be easily lost within the background noisy issue reports and no clear burst can be observed in the overall trend of all issues. Detection of such hidden emerging issues is a rather labor intensive and time-consuming process. We will elaborate more about this problem using a motivating example in Section 2.

In this paper, we propose iDice, an automated algorithm that helps support engineers identify the effective combinations that are associated with emerging issues. We formulate the problem of identifying emerging issues as a pattern mining problem: given a volume of customer issue reports over a period of time, the goal is to search for an attribute combination that isolates the entire multi-dimensional time series dataset into two partitions: one showing a significant increase of issue volume, and the other not showing such an increase. As the number of attribute combinations could be huge, we design several pruning techniques to significantly reduce the search space.

We have performed experiments to evaluate the effectiveness and efficiency of iDice, on both real-world and synthetic datasets. The F-measure values achieved by iDice are all above 0.85. We have also successfully applied iDice to the maintenance of Microsoft online services, and confirmed the usefulness of iDice in industrial practice.

The contributions of this paper are as follows:

- We propose iDice, an automated approach that helps support engineers effectively and efficiently identify the effective combinations that isolate the emerging issues.

- We evaluate iDice through in-house experiments. In addition, we also apply iDice to Microsoft online service systems in production. The results confirm the effectiveness and efficiency of iDice.

The rest of this paper is organized as follows. In Section 2, we introduce the motivation of this work. We formulate the problem of emerging issue identification and introduce iDice in detail in Sections 3 and 4, respectively. In Section 5, we evaluate iDice using experiments. In Section 6, we describe the case studies where iDice is applied to production online services. We present the related work in Section 7 and conclude this paper in Section 8.

2. MOTIVATION

2.1 An Example

The research described in this paper is motivated by the real-world requirements arisen from the maintenance of a Microsoft online service. Every day, the support team receives a lot of issue reports from customers around the world. Table I shows a sample of issue reports. In the table, each row represents one issue report. Each issue report has a set of associated attributes, such as *TenantType, ProductFeature, ProductVersion, SubscriptionPackage, DataCenter, Country, UserAgent, ClientOS*, and so on. Each attribute has multiple possible values and depicts a certain aspect of the context about the issue. For example, the attribute *ProductFeature* specifies the product feature with which customers encountered problems. Besides the attributes, each issue report also has a time stamp that records the time at which the issue was reported.

Most of the time, the support team receives a relatively stable number of issue reports every day. The issue reports contain many combinations of attributes, such as:

Country="India"; TenantType="Home"; DataCenter="DC1"
Country="India"; TenantType="Edu"; DataCenter="DC6"
Country="UK"; TenantType="Edu"; DataCenter="DC3"
...

As mentioned in Section 1, sometimes the number of issue reports associated with a certain attribute combination could suddenly increase, i.e. an emerging issue could occur. This could be due to a server-side fault (e.g., a configuration error, a software bug, a compatibility issue, or other software problems), or a hardware-related fault (e.g., hard disk crash, network device failure, etc.). For example, Figure 1 shows a real emerging issue detected in 2013. The number of issue reports containing the following attribute combination significantly increased on December 8:

Country="India"; TenantType="Edu"; DataCenter="DC6"

Further investigation showed that before December 8, 2013, this attribute combination was associated with an average of 70 issue reports per day. Starting from December 8, the number of issue reports associated with this attribute combination rose to over 300 per day. This particular attribute combination characterized the emerging issue and provided useful information for problem investigation. We call such an attribution combination *Effective Combination*.

Having identified the emerging issue, support engineers quickly found out that these issue reports were related to a software configuration error, which failed to create accounts for customers in India, who were subscribed to the service through TenantType EDU. Therefore, many customers contacted the Microsoft support team asking for help. The effective combination associated with the emerging issue provided useful information for support engineers to isolate the problem and locate the root cause.

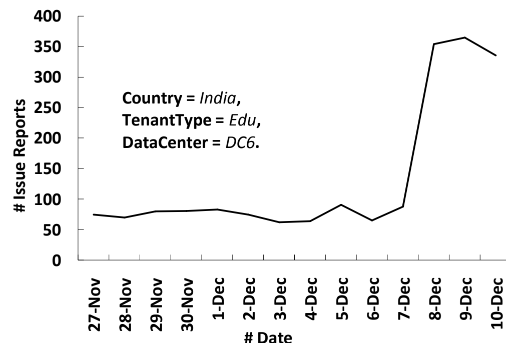

Figure 1: An example of emerging issue

As demonstrated by this example, it is important to effectively and efficiently detect such emerging issues and their associated attribute combinations. This not only helps resolve customer issues and increases customer satisfaction,

Table 1: **Sample Issue Reports for a Microsoft Online Service System**

Time	TenantType	ProductFeature	ProductVersion	Package	DataCenter	Country	UserAgent	ClientOS	...
5-Dec 11:01	Home	Admin	V15 RTM	Basic	DC1	India	IE7.0	Win8.1	...
6-Dec 15:01	Home	Modify	V15 RTM	Basic	DC1	USA	IE6.0	WinXP	...
7-Dec 10:22	Edu	Edit	V16 SP1	Lite	DC3	UK	Firefox30.0	Win8.1	...
8-Dec 05:33	Edu	Share	V16 SP2	Pro	DC6	India	IE7.0	WinXP	...
8-Dec 15:04	Enterprise	Share	V15 RTM	Lite	DC1	Australia	IE11.0	Win8.1	...
8-Dec 15:16	Edu	Admin	V16 SP1	Ultimate	DC6	India	IE11.0	Win8.1	...
8-Dec 15:26	Edu	Edit	V16 SP2	Pro	DC6	India	Firefox31.0	Win7	...

but also helps reduce the support cost as more incoming support requests related to the same issue could be avoided.

2.2 Challenge

To detect emerging issues and identify the associated attribute combinations, in current practice, most of the time the support engineers manually explore different attribute combinations using a pivot table[1], and look for candidate combinations that are associated with significant upward trend in the issue reports. For example, to detect the emerging patterns shown in Figure 1, the support engineers may first aggregate the data by date. They then look for potential emerging issues by selecting an attribute combination iteratively and summarizing the data table by that attribute combination.

However, manual identification of emerging issues could be very difficult when support engineers cannot observe significant changes in the overall number of issues. Some emerging issues can only be observed under certain attribute combinations. They may not always cause significant changes to the overall number of issue reports for the entire system. For example, in the motivating example, the number of issue reports associated with the effective combination {*Country* = "*India*" ; *TenantType* = "*Edu*"; *DataCenter* = "DC6"} experienced a "burst". However, because the support engineers actively resolved other issues, the overall quality of the system was improved. Therefore, the total number of issues for the entire system did not show any noticeable burst. In this scenario, the support engineers need to examine various attribute combinations one by one to detect emerging issues.

Clearly, the manual identification approach has the following problems:

- *Inefficient*: If there are many attributes and each attribute has many different values, there will be an explosive number of possible attribute combinations, which makes manual identification expensive or even impossible. For example, if each issue report has 8 attributes and each attribute has more than 10 values, then there are up to 10^8 possible attribute combinations. In order to locate the emerging issue as shown in Figure 1, support engineers might need to examine a huge number of candidate combinations. Obviously, the manual identification approach does not scale.

- *Ineffective*: The manual approach is often an ad-hoc exploration of different attribute combinations. The effective attribute combinations may be missed. The identified combinations may also contain redundancy and overlap.

To automate the identification of emerging issues, simple frequent itemset mining approaches [10] alone are not sufficient. From Table 1, we can see that the dataset of issue reports has two characteristics. On one hand, it is time series data with temporal order; on the other hand, it is also multi-dimensional data since each record has many attributes. Therefore, the dataset of issue reports can be treated as an combination of multi-dimensional data and time series data. Most of the closed itemset approaches only handle multi-dimensionality without considering the temporal property. They cannot detect the change point at which the volume of issue reports significantly increases. Furthermore, the attribute combination associated with an emerging issue may not be the frequent patterns across the entire dataset. This is because the number of data records associated with an effective combination may only occupy a relatively small portion of the entire data space. For example, the total number of reports associated with the effective combination {*Country* = "*India*" ; *TenantType* = "*Edu*"; *DataCenter* = "DC6"} may be lower than the number of reports associated with the attribute combination {*Country* = "*USA*" ; *TenantType* = "*Home*"} in the same month. However, the number of reports associated with the effective combination experienced a "burst", while the others did not.

Emerging pattern mining [8, 16] can be used to detect attribute combinations whose frequencies change significantly from one dataset to the other. Emerging pattern mining methods mainly target at dataset without temporal order, while the issue report data is time series data. Therefore, they cannot be directly applied to detect changes and isolate issues in issue reports.

In summary, the problems of manual identification reflect a great demand on an automated tool for timely detection of emerging issues. However, there is no existing approach that can be directly used to solve this problem. This motivates us to design a new approach to effectively and efficiently identify the emerging issues in issue reports.

3. PROBLEM FORMULATION

3.1 Effective Combinations

Motivated by the great demand from the maintenance of real-world software systems, we design an approach to automatically identify the attribute combinations that characterize the emerging issues (i.e., the effective combinations). Because issue report data could contain a large number of attribute combinations, the challenge is to effectively identify the effective combinations from all the possible combinations.

The entire set of attribute combinations form a lattice structure[10] through their subset-superset relationships, as illustrated in Figure 2. Each node denotes an attribute combination, each edge denotes a subset-superset relationship.

[1]http://en.wikipedia.org/wiki/Pivot_table

For two attribute combinations X and Y, if the data containing X is also included in the data containing Y, we call X a subset of Y and Y a superset of X. For example, ABC is a subset of AB and AC.

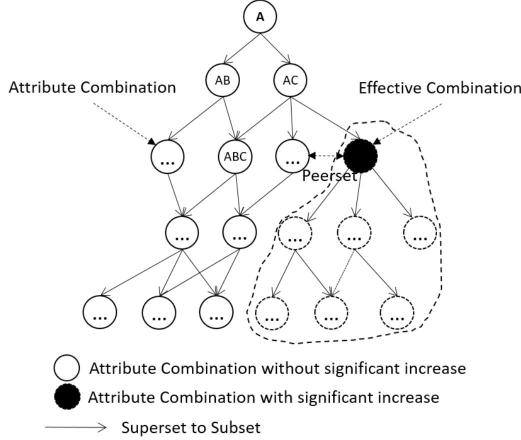

Figure 2: Effective combination

In Figure 2, the node filled with black denotes an effective combination. It can isolate the nodes with significant increases from other nodes that do not exhibit significant increases. If an effective combination X is the symptom of an emerging issue, and it is related to the burst of issue report volume, then all subsets of X should also be affected by the same issue and exhibit certain degree of significant volume increases at the same time, because other attributes are orthogonal to this emerging issue. For example, if {*Country=Japan, ProductVersion=V16SP2, ClientOS=Win8.1*} is an effective combination indicating an emerging issue, then all its subsets, such as {*Country=Japan, ProductVersion=V16SP2, ClientOS=Win8.1, TenantType=Edu*} and {*Country=Japan, ProductVersion= V16SP2, ClientOS=Win8.1, UserAgent=IE7.0*}, should also experience significant increases in issue volume. On the other hand, the supersets of an effective combination may or may not exhibit an increase in volume for the same emerging issue. For example, we may not see a noticeable burst from all reports about $\{ClientOS = Win8.1\}$.

3.2 Requirements

The effective combinations should be able to provide information to help support engineers identify and isolate problems. We have identified the following requirements for an effective combination:

- *Impactful*: Support team needs to allocate limited resource to help as many customers as possible through resolving the most impactful emerging issues. Therefore, an effective combination should associate with an emerging issue that corresponds to a relatively large number of issue reports.

- *Reflecting changes*: The identified attribute combinations should be able to separate themselves from the other attribute combinations. That is, the volume of issue reports associated with the identified attribute combination should exhibit a significant burst after the change point, while the volume of reports associated with other attribute combinations does not exhibit the same burst along time.

- *Less redundant*: The identified attribute combinations should be compact and concise because redundant results decrease detection accuracy and waste investigation efforts.

4. THE PROPOSED APPROACH

We propose *iDice*, an approach to automatic identification of effective combinations characterizing the emerging issues. The key challenges for iDice is that the possible attribute combinations form a huge search space, which makes it difficult or even impossible to check all the combinations one by one. Therefore, the core algorithm of iDice is to effectively reduce the search space without missing the effective combinations. In order to achieve this goal, we have designed the following pruning strategies according to the requirements described in Section 3.2:

- **Impact based Pruning**. Each effective combination should be related to a large volume of issue reports, which means that it has impacted a large number of customers. We adopt an impact-based strategy to remove the attribute combinations associated with small numbers of issue reports.

- **Change Detection based Pruning**. Effective combinations should be able to reflect significant volume increases of issue reports. Therefore, in the search process, we prune off the attribute combinations that exhibit small or no changes in issue volume.

- **Isolation Power based Pruning**. We use this strategy to remove the possible redundancy in the identified effective combinations, and to make the results concise and compact.

Figure 3 shows an overview of the iDice approach. iDice takes as input the issue report data, which is multi dimensional, time series data such as the one shown in Table 1. Each dimension denotes a categorical attribute of issues. After preprocessing, iDice performs the following three steps to efficiently search for effective combinations: impact-based pruning, change detection based pruning, and isolation power based pruning. Finally, the obtained results are ranked and returned to users. In this section, we first introduce each of the major steps and then present the complete algorithm.[2]

4.1 Impact based Pruning

In order to effectively reduce the search space, we perform pruning based on the impact of attribute combinations. We measure the impact of the attribute combinations as their corresponding issue volume. We only consider the attribute sets associated with large volume of issue reports, and prune off those without high enough volume.

To achieve so, iDice utilizes a mining algorithm to identify all the equivalent closed itemsets whose support value

[2]Note that our approach works in batch mode. This is because for the service products we worked with, the issue data is provided in a batch scenario - the data is firstly collected from different channels (including phone call, online form, and onsite-support) and then uploaded to a centralized server for further analysis on regular basis (after post-processings such as data cleansing). Therefore, we are unable to get data in a real-time manner.

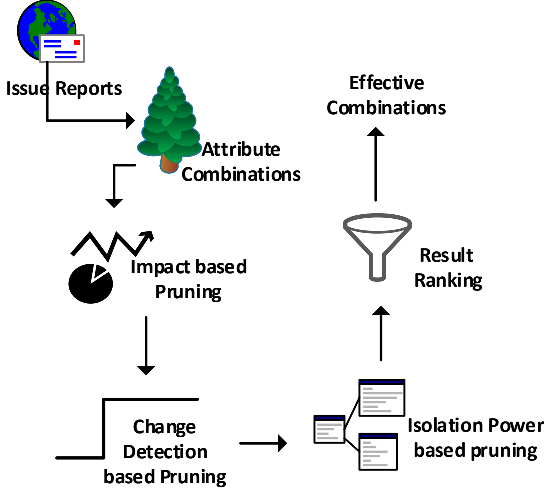

Figure 3: An overview of iDice

is larger than a user-defined support threshold. There are many closed itemset mining algorithms [10] available and any one of them can be applied. In our implementation, we use the BFS (Breadth-first Search) based closed itemset mining approach [10]. We ignore the time stamp information of the issue reports, and apply the BFS-based closed itemset mining algorithm on the entire dataset D. Clearly, the resultant closed itemsets X contain all the attribute combinations leading to the emerging issues. It is straightforward to prove that X obtained from D contain all the closed itemsets obtained from any supersets of D. As a property of effective combination, if one attribute combination has no sufficient volume, it is obvious that all its subsets do not have enough volume either (refer to Section 3). Thus we can directly remove this attribute combination together with all its subsets.

As an example, considering the following two attribute combinations:

X={*Country=India; TenantType=Edu; DataCenter=DC6*}
Y={*Country=USA; TenantType=Edu; DataCenter=DC1*}

If the occurrence of Y is low (i.e., lower than the support threshold) and the occurrence of X is high (i.e., higher than the support threshold), Y and all its subsets will be pruned off and X will remain.

4.2 Change Detection based Pruning

In addition to being impactful, the attribute combinations we look for should also be related to *emerging* issues. In other words, we need to find out the attribute combinations that correspond to significant increases in issue report volume (i.e., the bursts).

For each closed itemset obtained after impact-based pruning, we check whether the corresponding volume of reports has a significant increase. To achieve so, we consider the time stamp information of the data and build time series data for the closed itemsets. Each data point in the time series denotes the volume of reports that contains the corresponding closed itemsets at a particular time stamp. We then apply change detection algorithm to detect the change points (i.e. the points where issue bursts occur) in the time series data.

In our implementation, we adopt GLR (Generalized Likelihood Ratio) [3] as the change detection algorithm. Change detection can be formulated as a hypothesis-testing problem. Suppose the values of a time series fit a distribution θ_0, if there is a change, the values during the change region conform to another distribution θ_1. Here, the hypothesis H_0 corresponds to *"no change"*, and H_1 corresponds to *"change"*.

GLR maintains a threshold. Given a few continuous data points, if the sum of their logarithm-likelihood-ratio is greater than the threshold, these continuous data points are detected as a change region. The first point of the continuous data points is deemed as the change point [3]. For example, in Figure 1, the points from "Dec 8" to "Dec 10" constitute a change region, and the point of "Dec 8" is a change point. For the time series data without any change points, the corresponding attribute combinations will be pruned.

As an example, considering the following two attribute combinations:

X={*Country=India; TenantType=Edu; DataCenter=DC6*}
Y={*Country=UK; TenantType=Home; DataCenter=DC1*}

Their corresponding time series are S_X and S_Y, respectively. If we detect that the occurrence of S_X has a significant change (e.g., from 100 to 300) starting from Dec 8 (i.e. change point), while S_Y does not have a significant change over time, then Y will be pruned, and the change point of S_X will be used in follow-up isolation power based pruning.

4.3 Isolation Power based Pruning

As discussed in Section 3, an effective combination should be able to isolate the attribute combinations that exhibit changes from the other combinations that do not. This property can help us further remove the possible redundancy in the obtained itemsets. To achieve so, we propose the notion of *Isolation Power*:

$$IP(X) = -\frac{1}{\overline{\Omega_a} + \overline{\Omega_b}} \Big(\overline{X_a} \ln \frac{1}{P(a|X)} + \overline{X_b} \ln \frac{1}{P(b|X)} + (\overline{\Omega_a} - \overline{X_a}) \ln \frac{1}{P(a|\bar{X})} + (\overline{\Omega_b} - \overline{X_b}) \ln \frac{1}{P(b|\bar{X})} \Big) \quad (1)$$

Let S_X be the time series data corresponding to the attribute combination X, X_a denotes the volume of time series data in S_X during the change region of X, and X_b denotes the volume of time series data in S_X before the change point of X. Ω_a denotes the entire volume during the change region of X, and Ω_b denotes the entire volume before the change point of X. All $\overline{*}$ denote the mean value of the corresponding time series. Also:

$$P(a|X) = \frac{\overline{X_a}}{\overline{X_b} + \overline{X_a}},\ P(b|X) = \frac{\overline{X_b}}{\overline{X_b} + \overline{X_a}},$$
$$P(a|\bar{X}) = \frac{\overline{\Omega_a} - \overline{X_a}}{\overline{\Omega_a} + \overline{\Omega_b} - \overline{X_b} - \overline{X_a}},$$
$$P(b|\bar{X}) = \frac{\overline{\Omega_b} - \overline{X_b}}{\overline{\Omega_a} + \overline{\Omega_b} - \overline{X_b} - \overline{X_a}}.$$

The proposed Isolation Power is based on the idea of Information Entropy [1]. As discussed in Section 3 and illustrated by Figure 2, the entire set of attribute combinations form a lattice, and every node in the lattice can split the dataset into two parts: the issue reports that contain the attributes, and the reports that do not contain the attributes. If an attribute combination is an effective combination, all its subset

nodes in the lattice exhibit significant increases in the same change region, but its sibling nodes do not. Therefore, an effective combination is the node that can exactly split the entire dataset into two parts: with and without a significant increase. According to the information theory [1], the overall entropy of two datasets (e.g., A and B) where each dataset (A or B) contains samples with an identical property (e.g., all of them exhibit increases, or all of them do not exhibit increases) is much smaller than the entropy of two datasets where samples with different properties are mixed together. Based on this concept, we calculate Isolation Power to mimic the calculation of entropy.

During the search process, if the current set has a higher isolation power than its direct supersets and subsets, then the current set is an effective attribute combination, satisfying the requirements for effective combinations described in Section 3. In this case, all its subsets will not be searched. In this way, we can reduce search space using the Isolation Power measure. Considering a simple example with three attribute combinations:

X={Country=India}
Y={Country=India; TenantType=Edu; DataCenter=DC6}
Z={Country=India; TenantType=Edu; DataCenter=DC6; Package=Lite}

If Y has a higher isolation power than its subset Z and superset X, Y will be considered and Z and X will be removed from the search space.

4.4 Result Ranking

In real-world scenarios, we may obtain a set of effective combinations from the data. We rank the effective combinations according to their relative significance. We adopt a score similar to Fisher distance [10] for the ranking:

$$R = p_a * \ln \frac{p_a}{p_b} \tag{2}$$

In the above equation, p denotes the ratio: $p = \frac{V_{Xt}}{V_t}$, where V_{Xt} denotes the volume of current effective combination during a time period t and V_t denotes the total volume during t. p_a, p_b are the ratios during the detected change region (i.e. the time period of a change) and before the detected change point (i.e. the starting point of a change region) respectively.

We can see from Equation 2 that the score R: (1) considers the global impact of the combination. If two combinations have the same change ratio (which means that they have the same trending significance), we will rank the one with larger volume higher; (2) reflects the change ratio through $\frac{p_a}{p_b}$.

The attribute combination with a very low R score is considered less significant and can be pruned away. In our implementation, we prune away the attribute combinations whose R score is lower than a *cutoff threshold* (which is empirically set to 1.0 in our implementation). Finally, we rank the remaining attribute combinations and output them as the *effective combinations*.

4.5 The Overall Algorithm

The pseudo code for identifying effective combinations is shown in Algorithm 1. The algorithm takes the issue report data (which is multi-dimensional, time-series data) as input, and searches for effective combinations that are associated with emerging issues. The preprocessing at Line 1 includes data cleaning, which is to filter out the obvious noise attributes in the dataset, e.g., all the "null" values.

Algorithm 1: iDice(D)

```
   Input: D, issue reports (multi-dimensional time series
          data.)
   Output: P, the effective combinations.
1  PreProcessing dataset D;
2  while (true) do
3      //BFS-based Closed Itemset Mining on D
       p_i = BFS_Closed_Itemset().getNext() ;
4      if (p_i == NULL) then
5          Break;
6      Impact_Evaluation (p_i);
7      if impact of p_i < minimum support then
8          Remove the current set p_i;
9          Skip all subsets of p_i;
10         Continue;
11     s_i <-Map p_i into time series form;
12     Perform Change Detection on s_i ;
13     if s_i has no change point then
14         Continue;
15     Evaluate isolation power of p_i;
16     if isolation power of p_i > isolation power of p_i's
       direct subsets then
17         Add p_i to the candidate list P;
18         Skip all subsets of p_i;
19         Continue;
20 /*Result ranking*/
21 foreach p in P do
22     Calculate the ranking score R_p;
23     if R_p < cutoff_threshold then
24         Remove p from P;
25         Continue;
26     Rank p by its significance R_p;
27 Return P;
```

For each closed itemset p_i returned by the BFS (Breadth First Search) based closed itemset mining process, iDice performs Impact-based pruning (lines 6-10), Change Detection based pruning (lines 11-14), and Isolation Power based pruning (lines 15-19). These steps prune away attribute combinations and reduce the search space, making it possible to identify effective combinations from a large number of attribute combinations. Lines 21 to 26 denote the result ranking part of iDice.

5. EVALUATION

In this section, we describe our experimental evaluation of iDice. We aim to answer the following questions:

RQ1: How effective is iDice in detecting emerging issues?

RQ2: How efficient is iDice in detecting emerging issues?

RQ3: How does iDice perform under different configurations?

5.1 Setup

Datasets. We use two datasets to evaluate the effectiveness of iDice.

Microsoft Service X: Service X is a geographically distributed, external-facing, web-based online service, serving hundreds of millions of end users. During a certain period of time, the support team encountered a "burst" in the number of issue reports. We collected the actual issue report data, which contains more than ten thousand issue reports collected in early 2015. Each report has two attributes Application and DataCenter (apart from the Time and ID attributes). The Application attribute has 5 values (A1-A5). The DataCenter attributes has 10 values (DC1-DC10). There are total 92 emerging issues in this dataset, which are verified by the domain experts in the product team.[3] Due to sensitivity reasons, we omit and anonymize the detail information about the dataset.

Synthetic: To further evaluate iDice, we also design a simulation dataset. We first randomly generate a 60-day dataset with 100 issue reports on each day. Each issue report has 8 attributes, and each attribute has 4 distinct values. A randomly generated combination of attribute values is then assigned to each report. In this way, we create a synthetic dataset with 60*100 data points. We then simulate the occurrence of a single emerging issue as follows. First, we randomly select an attribute combination f_e to represent an error pattern and a day d_c as the change point, i.e. the day when f_e occurs. Second, we increase the number of issue reports containing f_e by a random value, e.g., between 15 and 20. Third, we perform such increment for f_e each day after the change point d_c until the 60th day. In this way, we get the resultant dataset seeded with one emerging issue. Similarly, we generate a dataset to simulate the occurrence of two emerging issues. During the evaluation, we randomly seed the emerging issue(s), perform the detection, and calculate the evaluation measures. We repeat such process N times and compute the average results.

Measures. We use *F-measure*, *Recall*, and *Precision* metrics to evaluate the effectiveness of iDice. The *F-measure* is defined as follows:

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3}$$

where $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TP+FN}$. Here, TP (true positive) is the number of actual effective combinations correctly reported by iDice. FP (false positive) is the number of wrongly reported effective combinations. TN (true negative) is the number of non-effective combinations that are correctly reported by iDice. FN (false negative) is the number of effective combinations that are not reported by iDice. The higher the metric values, the better the detection performance. In addition, we measure the execution time (in seconds) to evaluate the efficiency of iDice.

Hardware. The experiments were run on a PC with (Intel(R) Core(TM) i7-3770 CPU @ 3.34GHz with 16GB RAM).

Comparison algorithm. We compare iDice with DPMiner [16], which is one of state-of-the-art algorithms for mining emerging patterns. DPMiner detects emerging patterns, which are itemsets whose support rates increase significantly from one dataset to the other. As described in Section 2, emerging pattern mining techniques such as DPMiner is not originally designed for mining effective combinations because it does not support change detection in time series data. Also, there are no pruning strategies based on isolation power. In our experiment, to enable comparison, we manually set the data before the change point d_c as one database, and the remaining data as the other database. We then apply DPMiner and compare its results with those produced by iDice.

[3]We choose this dataset because it has quality labels verified by domain experts. Although we do have other real datasets that have a larger number of attributes, the quality of their labels may be insufficient due to the difficulties of manual identification effort as described in Section 2. Furthermore, such real issue reports cannot be labeled by non-domain experts due to lacking of domain knowledge.

Table 2: Evaluation results on Microsoft Service X dataset

Metrics	iDice	DPMiner
Recall	0.84	0.83
Precision	0.88	0.37
F-meaure	0.86	0.51

Table 3: Evaluation results on synthetic dataset

	Single Issue			Two Issues		
Metrics	R	P	F	R	P	F
iDice	0.98	0.95	0.96	0.84	0.90	0.86
DPMiner	1.00	0.16	0.27	1.00	0.14	0.25

(R: Recall, P: Precision, F: F-measure)

5.2 Experimental Results

5.2.1 RQ1: The effectiveness of iDice

According to the setup described in Section 5.1, we conduct experiments to evaluate iDice using both Microsoft Service X data and the synthetic data. The impact threshold is set to 1%.

For Microsoft Service X data, iDice is able to achieve good results, as shown in Table 2. The Recall, Precision, and F-measure values are 0.84, 0.88, and 0.86, respectively. Furthermore, iDice significantly outperforms the DPMiner approach. The improvement on F-measure is 69%.

For the synthetic data, we run iDice with single and two seeded emerging issues. Each experiment is performed 50 times. The average results are shown in Table 3. For the single-issue and two-issue experiments, the F-measure achieved by iDice is 0.96 and 0.86, respectively. These results are considered satisfactory. iDice also outperforms DPMiner in terms of F-measure. Although DPMiner is able to detect all effective attributes (Recall is 100%), it produces many redundant and irrelevant results, leading to low Precision values.

The reason why DPMiner performs worse than iDice is that it does not utilize the Isolation Power pruning strategy as designed by iDice. As a result, there are many redundant and noisy itemsets in the results.

5.2.2 RQ2: The efficiency of iDice

To evaluate the efficiency of iDice, we measure the execution time of iDice on the two datasets (Microsoft Service X and synthetic). For each dataset, we select an increasing percentage of records and run both iDice and DPMiner, until all records are selected. We then compare the performance of iDice and DPMiner under different data sizes.

The evaluation results are shown in Figure 4 and Figure 5. On both datasets, the execution time of DPMiner increases

sharply when the data size increases, while the execution time of iDice stays relatively steady. The results confirm the usefulness of the pruning strategies adopted by iDice.

Overall, our experiments show that iDice is effective and efficient in identifying effective combinations for emerging issues. iDice also performs better (in terms of both effectiveness and efficiency) than the related approach (DPMiner).

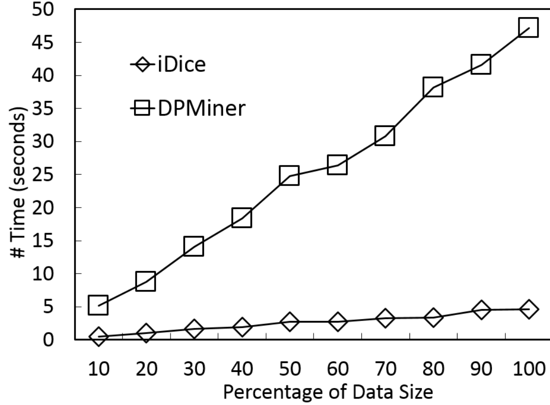

Figure 4: The performance of iDice on Microsoft Service X dataset

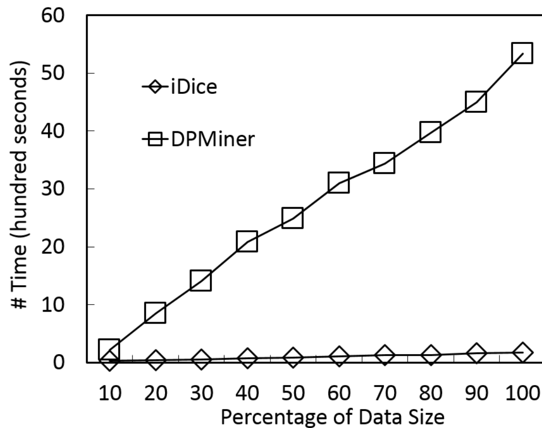

Figure 5: The performance of iDice on the synthetic dataset

5.2.3 RQ3: iDice under different configurations

As described in Section 4.1, iDice uses a parameter, the impact threshold (i.e. the minimum support), for impact-based pruning, because we are only interested in the attribute combinations that are associated with a large volume of issue reports (i.e. above the threshold). Currently, we allow users to set this parameter empirically based on their domain knowledge. Different values of the parameter could lead to different results. For example, if the impact threshold is too high, some useful information may be removed during the search process. On the other hand, if the threshold is too low, the results could contain some redundant and noisy information.

To understand the impact of different threshold settings on the results, we evaluate the effectiveness of iDice under different minimum support rates varying from 0.1% to 5% of the total data. The results for the synthetic dataset is shown in Figure 6. Initially, when the minimum support rate is 0.1%, the F-measure is about 0.86. When the minimum support rate increases from 0.5% to 2%, the F-measures achieved by iDice are around 0.9 and remain relatively constant. When the minimum support continues increasing, the F-measure starts to drop. This is because the number of data points exceeding 2% minimum support is rare. To summarize, the results show that, when minimum support is properly set, iDice is able to achieve consistently good results.

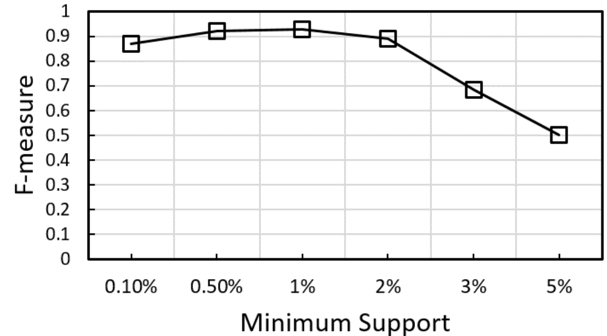

Figure 6: Effectiveness of iDice with different minimum support values

As described in Section 4.2, iDice selects only attribute combinations that are associated with significant volume changes, and uses the GLR method to detect the changes. In our experiments on the synthetic dataset, we simulate an emerging issue by increasing its occurrence by a certain amount (between 15 and 20). Here, we study the impact of different degrees of volume changes (i.e., different degrees of burst) on the results.

To do so, for the seeded attribute combination, after the change point, we insert the number of issue reports randomly drawn from the following intervals: [3, 5], [5, 10], [10, 15], [15, 20], [20, 25], and [25, 30]. In this way, we create 6 different synthetic datasets. We then evaluate the effectiveness of iDice on each dataset. The results are shown in Figure 7. We can see that when the number of seeded issues increases from [3, 5] to [25, 30], the effectiveness of iDice increases too. The results indicate that iDice performs better if the degree of burst is high.

5.3 Threats to Validity

We have identified some threats to validity that should be taken into consideration when using iDice.

Large number of issue reports: iDice is designed to help support engineers quickly detect emerging issues in a large number of issue reports based on data mining and statistical analysis. It is thus suitable for large-scale, software-intensive systems that are experiencing a long period of evolution and have received a large number of issue reports over time. For a small or short-lived system, the number of issue reports is often small, thus making the statistical analysis inappropriate.

Lack of ground truth for validation: In practice, the amount of issue reports for a large-scale system is huge and

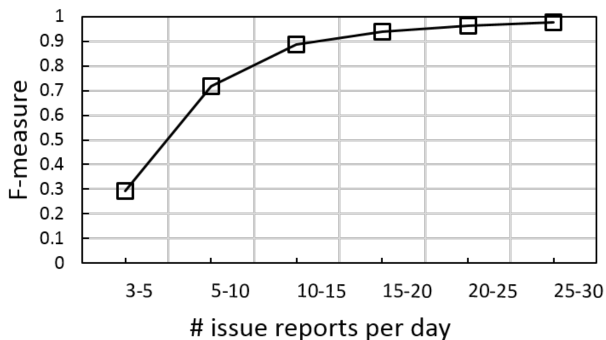

Figure 7: The effectiveness of iDice under different degree of burst

keeps increasing. Furthermore, the issue reports could contain a large number of attribute combinations. It is very time-consuming for support engineers to manually identify and verify each and every emerging issue as well as the associated effective combinations. Furthermore, such real issue reports cannot be labeled by non-domain experts due to the lack of domain knowledge. Therefore, it is very difficult for us to obtain a large-scale real dataset with high-quality labels. To overcome this threat, in our experiments we used a relatively smaller dataset from Microsoft Service X, which has quality labels verified by domain experts in the product team. We have also designed a synthetic dataset and manually seeded issues into the dataset. Furthermore, we have applied iDice to real-world online service systems and confirmed its effectiveness in practice.

6. SUCCESSFUL STORIES IN INDUSTRIAL PRACTICE

We have successfully applied iDice to the maintenance of Microsoft Service Y, which is a large-scale, widely-used online service system. The dataset of Service Y contains 11 attributes (including Feature, Country, Topic, Package, QuantitySize, DataCenter, AppVersion, OSVersion, etc.), and $5.2 * 10^{12}$ attribute combinations. Using a brute-force method, it may take 10^8 seconds to analyze the combinations, which is inapplicable in practice. iDice helped the support team of Service Y identify and resolve emerging issues. In this section, we present two successful stories of iDice. We omit and anonymize some details for confidentiality reasons.

6.1 The Mobile Client Issue

In June 2013, the Service Y team experienced a sudden increase of support requests. This upward trend of issue reports continued for several days, and the team was not able to locate the problem manually during those days. Although Service Y team eventually found the problem and fixed it, the maintenance cost significantly increased due to the large number of support requests received during this period.

In order to reduce the support cost, Service Y team looked for techniques to help them quickly and effectively detect the emerging issues. As requested, we ran iDice over the issue data of Service Y and successfully detected the significant upward trend in the support request volume starting in June 2013. As suggested by the identified effective combination, we found that this emerging issue was related to a version update of Service Y's mobile client. A software bug in the new mobile client caused an UI problem, which triggered the high volume of issue reports asking for help from the support team. Had iDice been used immediately after this incident occurred, a large amount of support cost (in millions dollars) could have been reduced.

6.2 The Double-Billing Issue

iDice helped Service Y team detect more emerging issues after it was used in production. One example is the double-billing issue as follows. In November 2013, iDice detected a significant upward trend in the support request volume (Figure 8). The corresponding effective combination is: $\{IssueCode = DoubleBilled, Country = UnitedStates, Feature = CreditRequest\}$. As indicated by the effective combination, this emerging issue occurred in the US market and was related to credit request and double billing. Using this information, the support team quickly diagnosed the problem and located the root cause, which was related to a vendor F. For the customers who used the charging service provided by F, a software bug in the service caused the double-billing problem. iDice enabled the support team to quickly diagnose and fix this emerging issue, thus reducing the cost for handling support requests and improving customer satisfaction.

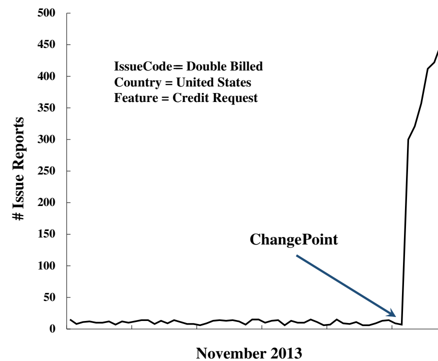

Figure 8: An emerging issue of double billing for Microsoft Service Y

7. RELATED WORK

7.1 The Analysis of Issue Reports

Many customer issues could be reported against a large and complex software system. The issues could be caused by program bugs, misconfigurations, operation errors, or environments. Over the years, there have been many empirical studies on the characteristics of issues of software systems. Various methods have also been proposed to detect such issues. For example, Li et al. [18] and Chou et al. [5] manually collected bugs from large-scale open source projects (such as Mozilla and Apache Web Server) and analyzed the bug characteristics. They classified the bugs into different categories (such as root causes, impacts and software components) and studied the correlation between categories. Zhang and Kim [28] studied the changes in bug report volume over time. They identified six common quality evolution patterns such as Impulse (short, dramatic increase of bug volume) and Hills (long-lasting high volume of bugs).

Their research suggests that the quality of an evolving software system is constantly changing. Kenmei et al. [14] applied ARIMA time series to model and forecast the trend of issue requests for open source projects. Li et al. [17] studied 18 software usage characteristics and investigated how they are related to field quality and how they differ between pre- and post-release. They identified the five most important usage characteristics through general linear regression. Kastner et al. [12, 13] analyzed complete configuration spaces of a highly configurable software system to detect some classes of issues. Menzies et al. [21] proposed machine-learning based methods for learning from bug datasets succinct rules that explain quality issues. Our work on iDice integrates closed itemset mining, change detection, and pruning techniques to detect emerging issues in multi-dimensional, time-series issue report data.

Many bug prediction techniques (e.g, [6, 22, 24, 27]) select a small set of important features that can best predict the defect-proneness of a software module. Our work focuses on the selection of features (attributes) that characterizes the emerging issues.

Kim et al. [15] found that only 10 to 20 crashes account for the large majority of crash reports for FireFox. If these top crashes are not fixed, more crash reports are expected. They trained a machine learner on the features of top crashes of past releases, and predict the top crashes before a new release. Bird et al. [4] found that the reliability of a software system depends on the environment it operates in. They performed an empirical study of more than 200,000 Windows users, and identified many factors that affect system reliability. They also applied association rule mining to detect the influence of factor combinations on reliability. Our work detects a set of attributes (features) that are associated with a significant change of issue reports.

Epifani et al. [9] targeted at the problem of identifying change points concerning the reliability and performance of a software service. Their approach is based on executions trace produced by client invocations, and only tries to detect change points. Nguyen et al. [23] detected performance regressions by analyzing a large number of performance counters using control charts. iDice works on customer issue reports. It identifies not only the change points, but also the effective attribute combinations that are associated with the changes.

7.2 Frequent and Emerging Pattern Mining

Frequent pattern mining has been widely used in software engineering research for problems such as bug detection [19, 26] and API usage mining [30, 25]. Our work on iDice is related to emerging pattern mining [8], which is not well explored by SE community. An emerging pattern is defined as an itemset whose support rate increases significantly from one dataset to the other[8]. Zhang et al. [29] proposed an improved algorithm for emerging pattern mining named ConsEPMiner, which utilizes two types of constraints, External constraints and Inherent constraints, to prune the search space effectively. Although these constraints are the basis of many posterior filtering methods, the algorithm could delete some important patterns, which adversely affects the detection accuracy.

Bailey et al. [2] introduced the first tree-based approach for mining Jump Emerging Patterns, which are patterns not occurring in the background data. Unlike the traditional FP-Trees [11], in their approach, the count of each item is split into two values: count of an item in the first dataset, and count of an item in the second dataset. Li et al. introduced DPMiner [16], which is an improved algorithm of [2]. In our work, DPMiner is used as a baseline algorithm for comparison.

Different from aforementioned methods for mining emerging patterns, iDice has the following characteristics: 1) The data characteristic is different: the related emerging pattern mining methods mainly target at dataset without temporal order, while iDice supports time series data. 2) The goal is also different: iDice targets at mining the attribute combination with the most isolation power in one dataset, while the related methods focus on mining all itemsets whose support values are different from one dataset to the other. 3) Regarding the pruning criteria, the related methods mainly define impact thresholds for pruning, while iDice performs pruning based on impact, change detection, and isolation power. Therefore, the results of iDice have not only isolation power, but also less redundancy. These characteristics make iDice more suitable for issue diagnosis, where both effectiveness and efficiency are desired.

8. CONCLUSION

Problem identification for emerging issues is important for the maintenance of large-scale software systems, especially online service systems. In this paper, we aim to identify an effective combination that is associated with a significant volume increase of issue reports. We have proposed iDice, an approach for identifying effective combinations for emerging issues. Our experiments demonstrate the effectiveness and efficiency of our approach. We have also applied iDice to the maintenance of several Microsoft online services, and the results confirm the usefulness of iDice in practice.

In the future, we will make iDice a distributed algorithm that is capable of supporting ultra-large-scale datasets. Furthermore, we will also apply iDice to solve other software engineering problems, such as the identification of misconfigurations for a large and complex software system.

Acknowledgment

We thank the intern students Chenhui Wang, Chen Luo, Hongzhi Chen, Hongbo Zhao, Kelu Diao and Yudong Xiao for their helpful discussions and the initial implementation of iDice. We thank partners at Microsoft product teams for their collaboration and suggestions on the application of iDice in practice.

9. REFERENCES

[1] C. Arndt. *Information Measures: Information and its Description in Science and Engineering.* Springer, 2004.

[2] J. Bailey, T. Manoukian, and K. Ramamohanarao. Fast algorithms for mining emerging patterns. In *Principles of Data Mining and Knowledge Discovery*, pages 39–50. Springer, 2002.

[3] M. Basseville, I. V. Nikiforov, et al. *Detection of abrupt changes: theory and application*, volume 104. Prentice Hall Englewood Cliffs, 1993.

[4] C. Bird, V.-P. Ranganath, T. Zimmermann, N. Nagappan, and A. Zeller. Extrinsic influence factors in software reliability: A study of 200,000 windows machines. In *Companion Proceedings of the*

36th International Conference on Software Engineering, ICSE'14, pages 205–214, 2014.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, 2001.

[6] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.

[7] R. Ding, Q. Fu, J. G. Lou, Q. Lin, D. Zhang, and T. Xie. Mining historical issue repositories to heal large-scale online service systems. In *Proc. the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 311–322, 2014.

[8] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 43–52. ACM, 1999.

[9] I. Epifani, C. Ghezzi, and G. Tamburrelli. Change-point detection for black-box services. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 227–236. ACM, 2010.

[10] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan kaufmann, 2006.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record*, volume 29, pages 1–12. ACM, 2000.

[12] C. Kastner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'11)*, pages 805–824. ACM Press, 2011.

[13] C. Kastner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'12)*, pages 773–792. ACM, 2012.

[14] B. Kenmei, G. Antoniol, and M. Di Penta. Trend analysis and issue prediction in large-scale open source systems. In *Proc. the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 73–82, April 2008.

[15] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

[16] J. Li, G. Liu, and L. Wong. Mining statistically important equivalence classes and delta-discriminative emerging patterns. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 430–439, 2007.

[17] P. L. Li, R. Kivett, Z. Zhan, S.-e. Jeon, N. Nagappan, B. Murphy, and A. J. Ko. Characterizing the differences between pre- and post- release versions of software. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 716–725, 2011.

[18] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[19] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings ESEC/FSE-13*, pages 296–305, New York, NY, USA, 2005. ACM.

[20] J.-G. Lou, Q. Lin, R. Ding, Q. Fu, D. Zhang, and T. Xie. Software analytics for incident management of online services: An experience report. In *Proc. the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13)*, pages 475–485, 2013.

[21] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Software Engineering, IEEE Transactions on*, 39(6):822–834, June 2013.

[22] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, Dec. 2010.

[23] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, 2012.

[24] S. Shivaji, E. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *Software Engineering, IEEE Transactions on*, 39(4):552–569, April 2013.

[25] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 319–328, 2013.

[26] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings ESEC-FSE '07*, pages 35–44, New York, NY, USA, 2007. ACM.

[27] H. Zhang. An investigation of the relationships between lines of code and defects. In *Proc. IEEE International Conference on Software Maintenance (ICSM'09)*, pages 274–283, Sept 2009.

[28] H. Zhang and S. Kim. Monitoring software quality evolution for defects. *IEEE Software*, 27(4):58–64, 2010.

[29] X. Zhang, G. Dong, and R. Kotagiri. Exploring constraints to efficiently mine emerging patterns from large high-dimensional datasets. In *Proc. of the 6th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 310–314, 2000.

[30] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, Genoa, pages 318–343. Springer-Verlag, 2009.