Multi-objective Test Report Prioritization using Image Understanding

Yang Feng[†*], James A. Jones[†], Zhenyu Chen[*], Chunrong Fang[*]
[†]Department of Informatics, University of California, Irvine, USA
[*]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
{yang.feng, jajones}@uci.edu, zychen@nju.edu.cn

ABSTRACT

In crowdsourced software testing, inspecting the large number of test reports is an overwhelming but inevitable software maintenance task. In recent years, to alleviate this task, many text-based test-report classification and prioritization techniques have been proposed. However in the mobile testing domain, test reports often consist of more screenshots and shorter descriptive text, and thus text-based techniques may be ineffective or inapplicable. The shortage and ambiguity of natural-language text information and the well defined screenshots of activity views within mobile applications motivate our novel technique based on using image understanding for multi-objective test-report prioritization. In this paper, by taking the similarity of screenshots into consideration, we present a multi-objective optimization-based prioritization technique to assist inspections of crowdsourced test reports. In our technique, we employ the Spatial Pyramid Matching (SPM) technique to measure the similarity of the screenshots, and apply the natural-language processing technique to measure the distance between the text of test reports. Furthermore, to validate our technique, an experiment with more than 600 test reports and 2500 images is conducted. The experimental results show that image-understanding techniques can provide benefit to test-report prioritization for most applications.

CCS Concepts

•Software and its engineering → Maintaining software;

Keywords

Crowdsourced Testing; Test Report Prioritization; Image Understanding; Multi-Objective Optimization

1. INTRODUCTION

Crowdsourced techniques have recently gained wide popularity in the software-engineering research domain [20]. One of the key advantages of crowdsourced techniques is that they can provide engineers with information and operations of real users, and those users provide data from tasks performed on real, diverse software and hardware platforms. For example, crowdsourced testing (*e.g.,* beta testing) provides validation data for a large population of varying users, hardware, and operating systems and versions. Such benefits are particularly ideal for mobile application testing, which often needs rapid development-and-deployment iterations and support many mobile platforms. In addition, crowdsourced mobile testing can provide developers with real users' feedback, new feature requests, and user-experience information, which can be difficult to obtain through conventional software testing practices. For these reasons, several successful crowdsourcing mobile testing platforms (such as uTest,[1] Testin,[1] and AppStori[1]) have emerged in the past five years [20].

Typically, crowdsourced workers provide information for developers in the form of *test reports*, which may consist of screenshots and textual content. Due to the inherent nature of crowdsourced testing, which usually involves a large number of users, the number of test reports can be great and the resulting task of inspecting those test reports can be quite time-consuming and expensive. As such, it is natural for developers to seek methods to assist in identifying and prioritizing new and useful information.

In the past decades, to alleviate tedious test-report inspection, researchers have proposed many full- or semi-automatic methods [6, 9, 11, 26, 30, 34, 35, 37, 39], in which, they mainly focused on the problems of duplicate-report identification, report classification, and report prioritization. To reduce the costs of inspecting duplicate test reports, techniques have been proposed and widely used, such as Bugzilla[1] and Mantis.[1] Similarly, report-classification techniques have been proposed to group similar reports, so that ideally an expert would only need to sample some reports from each group to gain a sufficient understanding of the bugs that they represent [6, 26, 35]. Finally, prioritization techniques have gained wide attention in the software-testing domain. Feng *et al.* [9] first proposed the concept of prioritization of crowdsourced testing reports. Instead of attempting to reduce the time-cost of inspecting, the basic assumption of prioritization techniques is "the earlier a bug is detected, the cheaper it is to remedy," which implies that all of the reports will be eventually inspected.

In almost all such techniques, the test reports are captured and analyzed based on their textual similarity (*e.g.,*

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970367

[1]utest.com, itestin.com, appstori.com, bugzilla.org, mantisbt.org

[9, 11, 30, 34, 35, 37]) or based on their execution traces (*e.g.,* [6, 26, 39]). For software designed for a desktop computer, such techniques are likely sufficient. However, for mobile software, writing long and descriptive test reports may be more challenging on a mobile-device keyboard. In fact, test reports written from mobile devices tend to be shorter and less descriptive, but also to include more screenshots (primarily due to the ease of taking such screenshots on mobile platforms). Due to this paucity of textual information for test reports, and also due to the ambiguity of natural language and prevalence of badly written reports [41], utilizing the screenshots to assist with such mobile crowdsourced testing techniques is appealing. Moreover, the activity views of typical mobile applications often provide distinguishable aspects of the software interface and feature set, and provide more motivation for utilizing such screenshots.

In this paper, we proposed an approach to test-report prioritization that utilizes a hybrid analysis technique, which is both text-based and image-based. This approach is a fully automatic diversity-based prioritization technique to assist the inspection of crowdsourced mobile application test reports. To facilitate this, we capture textual and image information and measure the similarity among these artifacts. For the image analyses, we employed the Spatial Pyramid Matching (SPM) [17] technique to measure the similarity of screenshots. For the textual analyses, we used natural-language textual analysis techniques to measure the similarity of textual descriptions within test reports. Finally, we combine these similarity results using a multi-objective optimization algorithm to produce a hybrid distance matrix among all test reports. Based on these results, we prioritize the test reports for inspection using a diversity-based approach, with the goal of assisting developers of finding as many unique bugs as possible, as quickly as possible.

To evaluate this proposed hybrid test-report prioritization technique, we implemented the technique and conducted an experiment. The experiment was conducted with three companies and more than 300 students, who simulated the crowdsourcing of testing of five widely-used mobile applications. In all, we received and analyzed 686 crowdsourced test reports from the crowd workers. We assessed effectiveness of our technique using the Average Percentage of Faults Detected (APFD) [28] metric and the fault detection rate. To serve as our baseline effectiveness results, we calculated the results of two strategies: an *Ideal* strategy, which is a best-case ordering to find all bugs in the shortest order possible, and a *Random* strategy, which is a random ordering.

The results of our empirical study shows that: (1) Screenshots are critical in the test report of mobile application, which could significantly improve the effectiveness of the prioritization technique and the efficiency of test-report inspection; (2) For certain classes of mobile applications, our multi-objective optimized prioritization technique can outperform the single image-based optimized technique, the text-based optimized technique, as well as the random technique.

The main contributions of this paper are as follows:

- To the best of our knowledge, this is the first work to take the image information as well as the text information of test reports into consideration to assist the inspection procedure.

- A novel multi-objective optimization-based technique is proposed to combine the image similarity and text similarity, which improves the effectiveness and efficiency of test-report maintenance.

- Five mobile applications with more than 2500 screenshots are used to evaluate our test prioritization techniques. Based on the experimental results and our experiences, we provide some practical guidance for crowdsourced mobile test-report prioritization.

2. BACKGROUND

Test and Bug Report Resolution. Software-maintenance activities are known to be generally expensive and challenging. One of the most important maintenance tasks is bug-report resolution. However, current bug-tracking systems such as Bugzilla, Mantis, the Google Code Issue Tracker, the GitHub Issue Tracker, and commercial solutions such as JIRA rely mostly on unstructured natural-language bug descriptions. These descriptions can be augmented with files uploaded by the reporters (*e.g.,* screenshots).

Although test descriptions and execution traces are currently used to characterize and analyze test reports, how to involve screenshots remains unsolved. Specifically for mobile crowdsourced testing, the reporters often prefer to provide only short text descriptions along with necessary screenshots. In this situation, how to combine short text processing with image processing is important for test-report prioritization.

Artificial-intelligence and computer-vision researchers created a class of analyses classified as *image understanding*, which extracts features from images and uses them for analysis. Within the software-engineering research domain, image-understanding techniques have been used in cross-browser issues for web applications. Cai *et al.* propose the VIPS algorithm [2], which segments a web page's screenshot into visual blocks to infer the hierarchy from the visual layout, rather than from the DOM. Choudhary *et al.* proposed a tool called WEBDIFF to automatically identify cross-browser issues in web applications. Given a page to be analyzed, the comparison is performed by combining a structural analysis of the information in the page's DOM and a visual analysis of the page's appearance, obtained through screen captures.

However, to date, there has been no work that addresses the use of screenshot images for use with test reports, particularly for mobile test reports produced by crowd workers in crowdsourced testing. Unfortunately, the crowd workers tend to describe bugs with a direct screenshot and short descriptions rather than verbose and complex text descriptions. At the same time, the developers are also interested in screenshots rather than inspecting the workers' long natural language descriptions. But, due the complexity of image understanding, there is a paucity of study on automated processing of screenshots in crowdsourcing testing.

In this paper, we overcome the difficulties in understanding the screenshots by applying advanced image matching techniques.

Image Understanding. Image matching is an important problem in the area of computer vision. Matching images of real world objects is particularly challenging as a matching algorithm must account for factors such as scaling, lighting, and rotation. Fortunately, the images that we compare in this work are screen captures of application views rendered by different devices by different workers for different apps. In

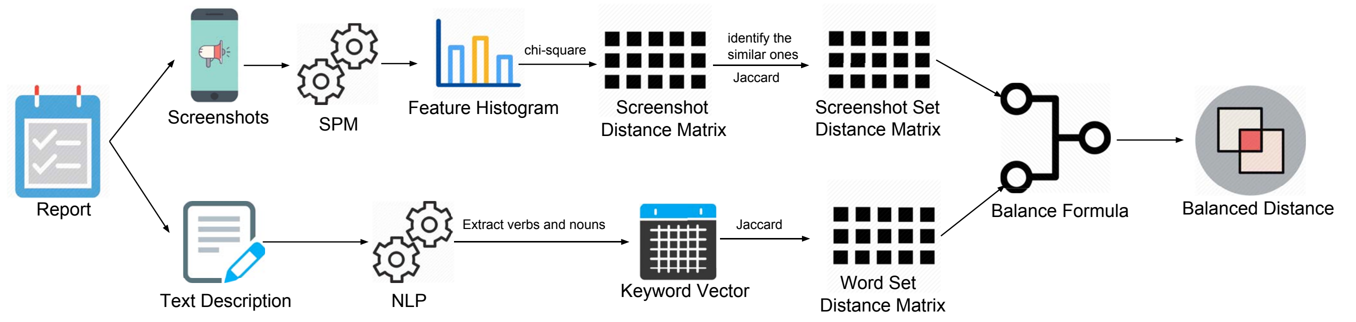


Figure 1: Test-report processing framework

this context, the above issues are ameliorated, and the main problems are, for instance, the shifting of GUI elements or the fact that some elements are not displayed at all.

A basic technique for comparing two images is to compare their histograms, where an image histogram represents the distribution of the value of a particular feature in the image [1]. In particular, a color histogram of an image represents the distribution of colors in that image (*i.e.,* the number of pixels in the image whose color belongs in each of a fixed list of color ranges, or "bins"). Obviously, if two images are the same, their color distributions will also match. Although the converse is not true, and two different images can have the same histogram, this issue is again not particularly relevant in our problem domain.

3. TECHNIQUE FRAMEWORK

This section elaborates the details of our method. We assume the test reports only consist of two parts: text description and screenshots, which we will handle separately and finally generate the balanced distance. Figure 1 shows the framework of calculating the distance between the test reports, which mainly contains three steps: (1) screenshot-set distance calculation, (2) test-description distance calculation, and (3) distance balancing. After we compute the distance matrix from the test-report set, we apply various strategies to prioritize test reports.

3.1 Preliminary

Even though, in practice, there could be other multimedia information that exists in the mobile test reports, such as the short operation videos and voice messages, our experience indicates that text descriptions and screenshots are the most widely used types of information. In this paper, we focus on the processing of mobile screenshots to assist the test-report prioritization procedure. We assume each of the test reports only consists of two parts: a text description and a set of screenshots, *i.e.,* the test report set $R(r) = \{r(S_i, T_i)|i = 0...n\}$, in which, S denotes the screenshots (*i.e.,* images) containing the views that may capture symptoms of the bug being reported, and T denotes the text describing the buggy behavior.

3.2 Test-Description Processing

The processing of text consists of two steps: (1) keywords set building and (2) distance calculation. Because natural-language-processing (NLP) techniques have been widely used to assist various software engineering tasks (*e.g.,* [9, 10, 30, 38]), we focus our description below on the distinguishing features and implementation choices of our approach.

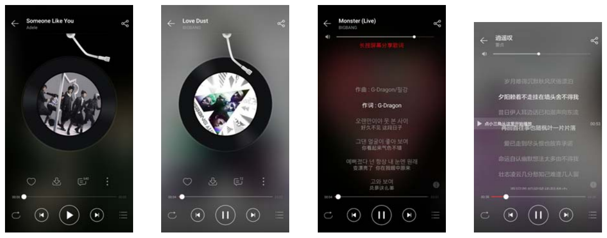
Keywords Set Building. In order to extract the keywords from the natural-language description, we first need to segment the text. Fortunately, word segmentation is a basic NLP task, and as such many efficient tools for word segmentation for different natural languages have been implemented [13]. In our method, we adopted the Language Technology Platform (LTP)[2] [4], which is the most widely used Chinese NLP cloud platform, to process the Chinese text descriptions. LTP segments the *Text* parts of test reports and marks each word with its Part-of-Speech (POS) for its context. In this procedure, LTP used the Conditional Random Fields (CRF) [15] model to segment Chinese words and adopted the Support Vector Machine (SVM) approach for tagging the POS. After we compute the segmentation results with the POS tags, we filter out relatively meaningless words that could negatively impact the distance calculation. According to prior works (*e.g.,* [27,33]), verbs and nouns can reveal the main information of a document. So, to simplify the technique, we extract only the nouns and verbs to build the keywords sets.

It is worth noting that our technique should not be limited to only the Chinese language. By applying other NLP tools, such as the Stanford NLP toolkit,[3] similar text models can be built for text descriptions written in other languages, such as English, French, or German. However, different natural languages have different characteristics, and may need special accommodations. For example, languages with relatively more prevalent polysemy (*i.e.,* many possible meanings for a word or phrase) and synonyms may require special processing, such as synonym detection and replacement, to avoid negative impacts on analyses.

Distance Calculation. Our method focuses on processing mobile test reports. Compared with the test reports of desktop or web applications, one characteristic of typical mobile test reports, and based on our experience, is that their text descriptions are shorter and contain more screenshots. As such, we treat all of the words in the text description equally, and we adopted the Jaccard Distance to measure the difference between the text descriptions T_i in the test-report set $R(r)$. The definition of Jaccard Distance used in our technique is presented in the following equation, in which, K_i denotes the *keyword set* of test report T_i, and $DT(r_i, r_j)$ denotes the distance between the text portion of the test reports r_i and report r_j.

$$DT(r_i, r_j) = 1 - \frac{|K_i \cap K_j|}{|K_i \cup K_j|}$$

[2]http://www.ltp-cloud.com/
[3]http://nlp.stanford.edu/software/



(a) Playing-1 (b) Playing-2 (c) Lyrics-1 (d) Lyrics-2

Figure 2: Four example screenshots from the test reports of the CloudMusic application. (a) and (b) are screenshots of the playing view, and (c) and (d) are screenshots of the lyrics view of two different songs.

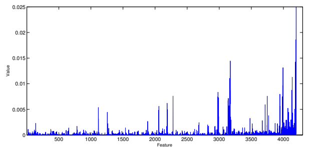
3.3 Screenshot Processing

Compared with NLP techniques, image understanding techniques are relatively less studied and used in the software-engineering domain. One of our motivations of conducting this research is to proposed a method to extract the information from images to assist software-engineering tasks. The workflow of processing screenshots S is presented in the top branch of Figure. 1. The process is composed of three key steps to build up the distance between screenshot sets: (1) building feature histograms, (2) calculating distance between individual screenshots, and (3) computing the distance between screenshot sets.

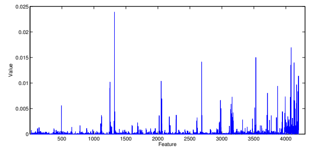
Feature Histogram Building. In order to compute the difference between the screenshots, we convert the screenshots into feature vectors. Bug screenshots provide not only views of buggy symptoms, but also app-specific visual appearances. We hope to automatically identify application behaviors based on their visual appearance in the screenshots. However, the screenshots often have variable resolution and complex backgrounds. Therefore, modeling the similarity between the screenshots merely based on RGB is not an approach that is well suited for our task. To address the challenges, we apply the Spatial Pyramid Matching (SPM) [17] to build a global representation of screenshots. Since the details of SPM are beyond this paper's topic, we only briefly introduce it here.

Given an image, SPM partitions it into sub-regions in a pyramid fashion. At each pyramid level, it computes an orderless histogram of low-level features in each sub-region. After decomposition, it concatenates statistics of local features over sub-regions from all levels. After building the "Spatial Pyramid" representation, we apply kernel-based pyramid matching scheme to compute feature correspondence in two images.

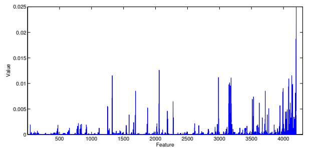
Figure 2 presents four original and actual screenshots from four test reports of a popular Chinese music-playing app, CloudMusic. Figures 2a and 2b show the music-playing view of the application, and Figures 2c and 2d show the lyrics view. Note that in each screenshot, the details of the view differ: *e.g.,* different music is playing, different background images appear, different lyrics are shown, and even the screen size is different for the last image. The layout of screenshots and background colors differ and provide challenges for correct matching: although Figures 2a and 2b have the same view layout, Figures 2b and 2d share a similar background color. If we were to directly calculate distance based on the RGB histograms, we would incorrectly



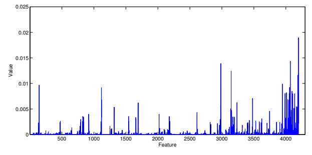
(a) Playing-1



(b) Playing-2



(c) Lyric-1



(d) Lyric-2

Figure 3: The corresponding feature histograms of the screenshots in Figure 2.

Table 1: Distance between screenshots of Figure 2

	Playing-1	Playing-2	Lyrics-1	Lyrics-2
Playing-1	0	**0.38957**	0.40255	0.45109
Playing-2	**0.38957**	0	0.51161	0.51873
Lyrics-1	0.40255	0.51161	0	**0.32029**
Lyrics-2	0.45109	0.51873	**0.32029**	0

get a closer distance between Figures 2b and 2d. Nevertheless, the image-understanding technique should be able to capture the similarities of the the similar views. Intuitively, Figures 2a and 2b should be identified as similar views, and Figures 2c and 2d should be identified as similar views.

Based on the four images, SPM first builds the histograms of features for each of image. The resulting histograms for these images are shown in Figure 3.

Screenshot Distance Calculation. Using the screenshot feature histograms, a distance is computed for each pair of images. To compute such distances between feature histograms, we adopt the chi-square distance metric [29]. The chi-square metric is generally used to compute the distance between two normalized histogram vectors, *i.e.,* their elements sum to 1. Also, both of the pairwise histograms being compared should contain the same number of bins (*i.e.,* the vectors should have the same number of dimensions).

We use $H_i(x_1, x_2, ..., x_n)$ to denote the feature histogram of screenshot s_i, and $H_i(x_k)$ to denote the value of kth feature of s_i. The formula used to calculate chi-square distance $Ds(s_i, s_j)$ between screenshot s_i and s_j is defined as follows:

$$\begin{aligned} Ds(s_i, s_j) &= \chi^2(H_i, H_j) \\ &= \frac{1}{2}\sum_{k=1}^{d}\frac{(H_i(x_k) - H_j(x_k))^2}{H_i(x_k) + H_j(x_k)} \end{aligned} \quad (1)$$

Based on Equation 1, we obtain the distance matrix shown in Table 1 from the feature histograms of Figure 3.

These results show that the calculated distance between the same views (Playing-1 and Playing-2, and Lyrics-1 and Lyrics-2) have relatively shorter (*i.e.,* smaller) distances (0.389 between playing screenshots and 0.320 between lyrics screenshots) than the across-view distances.

Screenshot Sets Distance Calculation. The previous step uses the chi-square distance metric to compute distances between pairs of screenshots. However, in practice, each test report may contain more than one screenshot. So, in this step, we compute the distance between screenshot sets. To account for the diversity of display resolutions of mobile devices and user content (*e.g.,* songs, backgrounds), we set a threshold γ to assess screenshots that match. The γ threshold is first used to find representative members from within the same screenshot set (*i.e.,* from the same test report). Screenshot subsets whose histograms produce chi-square distances that are below the distance threshold (*i.e.,* assessed as representing the same situation) are first represented as an aggregated, summary histogram which is computed as the mean of the feature histograms from the constituent members.

Once the representative set of screenshots are selected from each test report, the chi-squared metric with the γ metric is again used to compute the across-test-report screenshot similarity between the representative screenshots. Again, for screenshots (*i.e.,* their representative histograms) whose distance is less than γ, they are assessed as representing the same view, and as such, the similar and non-similar screenshots from each test report can be used to calculate the inter-test-report screenshot set distance for a pair of reports. For this calculation, we use the Jaccard distance metric. For the test reports r_i and r_j and their respective screenshot sets S_i and S_j, the distance metric is defined as:

$$DS(r_i, r_j) = 1 - \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$$

Note that in the special case where both S_i and S_j are the empty set (*i.e.,* no screenshots were included for either test report), we assess DS to be zero.

3.4 Balanced Formula

Based on above distance computations for both the textual descriptions and the screenshot sets, we combine these distances to produce a hybrid distance. We present Equation 2 to combine these differing distance values. Equation 2 is a step-wise formula, where the first condition holds for when the textual descriptions are assessed to be identical by way of the text distance formula DT. In this case, we assess the balanced distance metric to be similarly identical. In the next step, where $DS = 0$, where typically no screenshots were included for either test report, the textual difference is used and scaled to make them more similar, and thus less diverse. This diversity adjustment will make these less descriptive test reports less likely to be highly prioritized in the next prioritization step. In the final step, which holds in all other cases, the harmonic mean is calculated between the textual distance DT and screenshot set distance DS. The resulting balanced distance BD is used to represent the pairwise distance of the corresponding test reports.

$$BD(r_i, r_j) = \begin{cases} 0, & \text{if } DT(r_i, r_j) = 0 \\ \alpha \times DT(r_i, r_j), & \text{if } DS(r_i, r_j) = 0 \\ (1+\beta^2)\frac{DS(r_i,r_j) \times DT(r_i,r_j)}{\beta^2 DS(r_i,r_j) + DT(r_i,r_j)}, & \text{otherwise} \end{cases} \tag{2}$$

3.5 Diversity-Based Prioritization

Using the computed balanced distance measures for all test reports, we can prioritize the test reports for inspection by developers. The guiding principle of our prioritization approach is to promote diversity of test reports that get inspected. In other words, when a developer inspects one test report, the next test reports that she inspects should be as different as possible to allow her to witness as many diverse behaviors (and bugs) as possible in the shortest order. This diversity-based prioritization strategy has been used by other software-engineering researchers for test prioritization (*e.g.,* [5, 12, 32]). The goal is for software engineers to find as many bugs as possible in a limited time budget.

Given Q denotes the result queue, the distance between a test report r and Q, denoted by $\mathcal{D}(r, Q)$, is defined by the minimal distance between r and each r_i in Q, *i.e.,* $\mathcal{D}(r, Q) = Min_{r_i \in Q}\{\mathcal{D}(r, r_i)\}$. The algorithm of **BDDiv** is shown in Algorithm 1. In the beginning, Q is empty, we first initialize the algorithm by randomly choosing one report from R and append it to Q. The second step is to calculate the distance between each test report $r_i \in R$ and Q. As soon as we get the distance values, we choose the largest one to append to Q. The whole procedure completes when $|R| = 0$.

Algorithm 1: BDDiv(BD, R)

```
 1: $Q = \varnothing$
 2: Randomly choose a test report $r_k$ from $R$, append $r_c$ to $Q$
 3: $R := R - \{r_k\}$
 4: while $|R| \neq 0$ do
 5:   $maxDis := -1, r_c = NULL$
 6:   for all $r_i \in R$ do
 7:     $minDis := 2$
 8:     for all $r_j \in Q$ do
 9:       if $BD(r_i, r_j) < minDis$ then
10:         $minDis = BD(r_i, r_j)$
11:       end if
12:     end for
13:     if $minDis > maxDis$ then
14:       $maxDis = minDis$
15:       $r_c = r_i$
16:     end if
17:   end for
18:   Append $r_c$ to $Q$
19:   $R := R - \{r_c\}$
20: end while
21: return $Q$
```

4. EXPERIMENT

In our experiment, we propose the following three research questions:

RQ1: Can test-report prioritization substantially improve test-report inspection to find more unique buggy reports earlier?
RQ2: To what extent can the image-based approaches improve the effectiveness of the text-only-based approach?
RQ3: How much improvement is further possible, compared to a best-case ideal prioritization?

If the software engineers have no test report prioritization technique, they may randomly inspect test reports, in a non-systematic order. $RQ1$ is designed to inform whether prioritization of test reports is, in fact, advantageous. To address the $RQ1$, we conduct the experiment to evaluate the effectiveness of our prioritization techniques alongside a Random-based strategy. $RQ2$ is designed to investigate whether image-understanding techniques can assist the inspection procedure compared with the text-only-based technique. $RQ3$ is designed to investigate the gap between the performance of our techniques and the theoretical Ideal prioritization technique, which could be helpful to engineers in

Table 2: Experimental Software Subjects

Name	Version	$\|R\|$	$\|F\|$	$\|S\|$	$\|R_s\|$	$\|R_f\|$
SE-1800	2.5.1	192	7	856	164	99
CloudMusic	2.5.1	96	16	272	70	40
Ubook	2.1.0	99	22	719	90	99
iShopping	1.3.0	209	73	581	160	130
JustForFun	1.8.5	90	9	109	69	90
Totals		686	127	2537	553	458

selecting proper techniques in practice and inform the future research in this field.

4.1 Software Subject Programs

From November 2015 to January 2016, we collaborated with three companies and more than 300 students to simulate a crowdsourced testing process. The five applications on which we simulated crowdsourced testing are as follows:

JustForFun: A picture editing and sharing application, produced by Dynamic Digit.

iShopping: A shopping application for Taobao, produced by Alibaba.[4]

CloudMusic: An application for free-sharing music as well as a music player, produced by NetEase.[5]

SE-1800: A monitoring application for a power supply company, produced by Panneng.

Ubook: An application for online education, produced by New Oriental.[6]

Testing for all of these applications was crowdsourced to workers on Kikbug.net. For these five apps, more than 300 students were involved. To perform crowdsourced testing, each student installed a Kikbug-Android app, chose testing tasks, and completed testing tasks on their own phone. During the testing process, workers performed testing tasks according to some guidelines, specified by the app developers. During task performance, the workers could take screenshots if necessary, such as experiencing some unexpected behavior. After the testing task was completed, the worker could provide a brief description on bug phenomenon on his own phone. Finally, the student submitted a test report, including the short descriptions and possible screenshots.

Then all the test reports are submitted to app developers, and the developers can inspect the reports and begin the debugging process. With the help of the developers' inspection, Kikbug obtained ground truth assessments for the students' reports. The detailed information of the applications is shown in Table 2, in which, the $|R|$ denotes the number of reports, $|F|$ denotes the number of faults revealed in the test reports, $|S|$ denotes the number of screenshots contained in the test reports, $|R_s|$ denotes the number of test reports containing at least one screenshot, and $|R_f|$ denotes the number of test reports that revealed faults.

4.2 Prioritization Strategies

Technique 1: Ideal. The best result in theory to inspect test reports in such a way as to demonstrate the most unique bugs as early as possible. Represented as IDEAL.

Technique 2: TextDiv. The prioritization strategy based only on the distance between test reports' text descriptions, *i.e.,* in this strategy DT will replace BD as the first parameter of Algorithm 1. Represented as TEXTDIV.

Technique 3: ImageDiv. The prioritization strategy based only on the distance between test reports' screenshots, *i.e.,* in this strategy DS will replace BD as the first parameter of Algorithm 1. Represented as IMAGEDIV.

Technique 4: Random. The random prioritization strategy, which is used to simulate the situation without any prioritization technique. Represented as RANDOM.

Technique 5: Text&ImageDiv. Our prioritization strategy that balances the distance of screenshot sets and text descriptions. Represented as TEXT&IMAGEDIV.

4.3 Evaluation Metrics

We employed the APFD (Average Percentage of Fault Detected) metric [28], which is the most widely-used evaluation metric for test-case prioritization techniques, to measure the effectiveness of our techniques. For each fault, APFD marks the index of the first test report revealing it. We present the formula to compute the AFPD value in Equation 3, in which, n denotes the number of test reports, M denotes the total number of faults revealed by all test reports, T_{fi} is the index of the first test report that reveals fault i.

$$APFD = 1 - \frac{T_{f1} + T_{f2} + ... + T_{fM}}{n \times M} + \frac{1}{2 \times n} \quad (3)$$

In our experiment, a higher APFD value implies a better prioritization result. That is, it can reveal more faults earlier than the other methods do.

Although the APFD values reflect the global performance of prioritization techniques, in practice developers often cannot inspect all reports in a limited time budget. Thus, we also provide a metric to reveal the percentage of bugs that would be found at certain milestones of inspection. For this, we use linear interpolation [18] to evaluate the partial performance of each prioritization technique. We define linear interpolation as following:

- $Q_p = M \times p$, which is the number of faults corresponding to a percentage p. Let $int(Q)$ and $frac(Q)$ be the integer part and fractional part of Q, respectively. If $frac(Q) \neq 0$, the linear interpolation is needed.
- i, j are the indexes of reports that reveal at least Q and $Q{+}1$ faults respectively. The linear interpolation value V_p is calculated as $V_p = i + (j - i) \times frac(Q)$

In our experiment, we set the $p \in \{25\%, 50\%, 75\%, 100\%\}$.

4.4 Experimental Setup

In order to ensure the correctness of the implementation of SPM, we directly used the MATLAB code provided by the inventors of SPM. There are some key parameters affecting the performance of SPM, which are the size of the descriptor dictionary $DictSize$, number of levels of the pyramid L, and number of images to be used to create the histogram bins $HistBin$. In our experiment, as the recommended values of the SPM inventor, we set $DictSize = 200$, $L = 3$, and $HistBin = 100$. For the NLP technique, because all of test reports in our experiment are in Chinese, we employed the LTP platform to assist the text description analysis.

Moreover, the size of screenshots (*i.e.,* image resolution) submitted by the crowd workers was not fixed; in fact, they varied widely. In order to apply the SPM technique, we resize all screenshots to 480 × 480 pixels. Given the way

[4]https://guang.taobao.com
[5]http://music.163.com
[6]http://www.pgyer.com/y44v

that the SPM technique focuses on detecting features within images, resizing the images should not produce a substantial impact to the distance calculation.

In this experiment, we implemented all of the strategies presented in Section 4.2. Particularly for the TEXT&IMAGEDIV strategy, we set the threshold of determining the identity of screenshots γ to 0.1, the factor α that is used to weaken the weight of test reports without any screenshots to 0.75, and the parameter β used to balanced the text-based distance and screenshot-set distance to 1, which means, we weigh the two kinds of distance equally.

5. RESULTS, ANALYSIS AND DISCUSSION

In this section we present the results of our experiment, then interpret those results to attempt answers to our research questions, and finally discuss the overall results. In order to reduce the bias that was introduced by the random initialization of the algorithm and the tie-breaking, we conducted the experiment 30 times and present the result in Figure 4. Figures 4 (a, c, e, g, and i) show the boxplots of the APFD results for the five projects, respectively, each aggregated over the 30 experimental runs. Figures 4 (b, d, f, h, and j) show the average fault detection rate curves. The exact mean value of APFD is shown in Table 3, which also includes the result of one-way ANOVA tests of all strategies: the improved extent over RANDOM, and the gap between our strategies and IDEAL. Furthermore, we present the mean linear interpolation value over the 30 experiment runs in Table 4 to demonstrate the performance of our techniques in limited time budgets.

5.1 Answering Research Question RQ1

RQ1: Can test-report prioritization substantially improve test-report inspection to find more unique bugs earlier?

Based on the results shown in Figure 4 (a, c, e, g, i) and in the third column of Table 3, we find, to different extents, all of the three diversity-based prioritization strategies outperform RANDOM. Furthermore, in Table 3, all *F-values* are relatively large and the *p-values* ≤ 0.001, which means the APFD values of the four strategies are significantly different. Compared with the RANDOM strategy, the percentage of improvement of TEXT&IMAGEDIV ranges 9.93% – 24.95%.

Summary: All of the diversity-based prioritization methods can improve the effectiveness of test report inspection over RANDOM, and thus test-report prioritization can substantially, and significantly, find more unique buggy reports earlier in the prioritized order.

5.2 Answering Research Question 2

RQ2: To what extent can the image-based approaches improve the effectiveness of the text-only-based approach?

Figure 4 reveals that, except on the "JustForFun" project, the TEXT&IMAGEDIV outperforms the TEXTDIV, IMAGEDIV and RANDOM strategies, which means, the image-understanding technique improves the performance of the text-only-based technique. We did a deeper investigation on this problem and found what we speculate to be the reason for the different result for the "JustForFun" project. JustForFun is an image editing and sharing application, and as such, the inherent functionality is to process various user-provided photos. The screenshots for this app largely consist of user content, with relatively few app-specific features in those screenshot images. Thus, the various screenshots of "JustForFun" make the screenshot sets distance calculating procedure generate large distances, even between the same activity views, which leads to a negative impact on the image-based strategies. In contrast, based on Table 4, TEXT&IMAGEDIV outperformed the single text-based prioritization techniques on inspecting different percentage of test report of "SE-1800", "CloudMusic" and "Ubook."

Summary: Generally, compared with the text-only-based prioritization strategy, the image-understanding technique is able to improve the performance of prioritizing test reports, both globally (*i.e.,* APFD) and partially (*i.e.,* linear interpolation at many level). However, we found that some classes of apps are naturally less suited for image-understanding techniques — namely apps where the bulk of the views are composed of user contect.

5.3 Answering Research Question 3

RQ3: How much improvement is further possible, compared to a best-case ideal prioritization?

The fourth column of Table 3 shows the gap between our strategies and the theoretical IDEAL. We found the gap between TEXT&IMAGEDIV and IDEAL vary from 15.21% to 31.98%. For more details, we can observe the growth curves in Figure 4. The curve of IDEAL grows at a fast rate. The best situation reached top while the TEXT&IMAGEDIV only stayed around 35%.

Summary: We find that our prioritization methods can provide a reasonable small gaps for the theoretical IDEAL result, particularly for some subjects. However, there is room for future work to continue to improve the prioritization ordering of test reports.

5.4 Discussion

Method Selection. Reflecting on all of our experimental results, we find that image-understanding techniques can provide benefits to test-report prioritization, and that the area of such hybrid text-and-image approaches demonstrates promise. That said, we also observed that different techniques may be more or less applicable for different types of applications. Specifically, we observed that the image editor app produced the worst results for the image-based and hybrid techniques, compared to text-only. In such cases, where the screenshots mainly represent user content, image-based techniques may be less applicable. However, in applications in which little user or external content is displayed, image-based or hybrid techniques may be more applicable.

One noteworthy point is that both the TEXTDIV and TEXT&IMAGEDIV are full-automated, which we believe are more applicable in practice than the semi-automated DivRisk and Risk techniques [9] that require the users to input the inspection result to prioritize the crowdsourced test reports dynamically.

Mobile Application Testing. All of our experimentation was conducted on mobile applications, and thus we cannot state with certainty that such results would hold for other types of GUI software, such as desktop or web applications. However, we speculate that while there will likely be new and unique challenges in these domains, the basic concepts would likely hold, at least for the class of applications with relatively less user content. Desktop and web applications have the potential for even more differing screen and window sizes, as well as multiple windows and pop-up dialog windows, and each of these unique aspects would likely need

Table 3: One-way ANOVA Tests

Method	APFD Means	Improvement $\frac{X-Random}{Random}$	Gap $\frac{Best-X}{X}$
SE-1800: $F(3,119) = 54.966$, *p-value* ≤ 0.001			
Ideal	0.982	37.47%	—
Text&ImageDiv	0.852	19.32%	15.21%
TextDiv	0.817	14.46%	20.10%
ImageDiv	0.836	17.04%	17.45%
Random	0.714	—	37.47%
CloudMusic: $F(3,119) = 73.170$, *p-value* ≤ 0.001			
Ideal	0.917	58.65%	—
Text&ImageDiv	0.722	24.95%	26.97%
TextDiv	0.664	14.98%	37.98%
ImageDiv	0.641	10.99%	42.94%
Random	0.578	—	58.65%
Ubook: $F(3,119) = 84.167$, *p-value* ≤ 0.001			
Ideal	0.889	40.92%	—
Text&ImageDiv	0.750	18.95%	18.47%
TextDiv	0.735	16.57%	20.88%
ImageDiv	0.686	8.69%	29.65%
Random	0.631	—	40.92%
iShopping: $F(3,119) = 73.178$, *p-value* ≤ 0.001			
Ideal	0.825	45.08%	—
Text&ImageDiv	0.625	9.93%	31.98%
TextDiv	0.614	7.88%	34.48%
ImageDiv	0.586	2.98%	40.89%
Random	0.569	—	45.08%
JustForFun: $F(3,119) = 94.482$, *p-value* ≤ 0.001			
Ideal	0.950	45.89%	—
Text&ImageDiv	0.784	20.41%	21.16%
TextDiv	0.842	29.28%	12.85%
ImageDiv	0.681	4.54%	39.55%
Random	0.651	—	45.89%

Table 4: Linear Interpolation (average number of inspected test reports)

Program	Strategy	25%	50%	75%	100%
SE-1800	Ideal	1.75	3.50	5.25	7.00
	Text&Image	**3.51**	**12.32**	**31.30**	91.70
	TextDiv	6.98	23.27	43.27	112.67
	ImageDiv	4.21	16.38	50.38	**86.47**
	Random	4.79	31.05	79.36	145.57
Cloud-Music	Ideal	4.00	8.00	12.00	16.00
	Text&Image	13.10	**24.30**	**39.33**	62.00
	TextDiv	16.10	34.57	44.57	**59.00**
	ImageDiv	**11.07**	26.53	49.77	85.83
	Random	14.10	33.97	59.20	88.83
Ubook	Ideal	5.50	11.00	16.50	22.00
	Text&Image	**7.33**	**18.67**	44.17	**64.43**
	TextDiv	10.52	23.67	**35.17**	78.03
	ImageDiv	8.05	20.20	50.73	95.40
	Random	9.35	29.03	57.82	93.13
iShop-ping	Ideal	18.25	36.50	54.75	73.00
	Text&Image	37.16	**66.42**	119.27	201.23
	TextDiv	52.89	82.82	**111.30**	**160.07**
	ImageDiv	**32.60**	75.88	134.59	206.30
	Random	37.20	83.72	144.13	207.13
Just-ForFun	Ideal	2.25	4.50	6.75	9.00
	Text&Image	2.94	9.32	18.13	64.83
	TextDiv	**2.88**	**8.07**	**17.28**	**45.23**
	ImageDiv	3.16	18.12	39.01	79.47
	Random	**2.88**	22.25	49.88	80.17

to be addressed. Overall, we speculate that the success of such image-understanding-assisted test-report prioritization techniques would likely depend on the visual complexity of the application views.

5.5 Threats to Validity

There are some general threats to validity in our experimental results. For example, we need more projects and different parameter values combinations to reduce the threat to external validity and to better generalize our results.

Crowd Workers. Due to a monetary limitation, we "simulated" the crowdsourced mobile testing procedure to validate our techniques, in which, we invited the students to work as crowd workers. Such a choice means that our population of workers may be less diverse than the population of crowdsourced workers from the general populace. Theoretically, "crowdsourcing" requires workers come from a large pool of individuals that one has no direct relationship with the others [20], which implies that our result may be different if the crowd workers were from the internet with open calls. However, according to the study of Salman *et al.* [31], if a technique or task is new to both students and professionals, similar performance can be expected to be observed. Based on this study, we believe this threat may not be the key problem for our validation procedure.

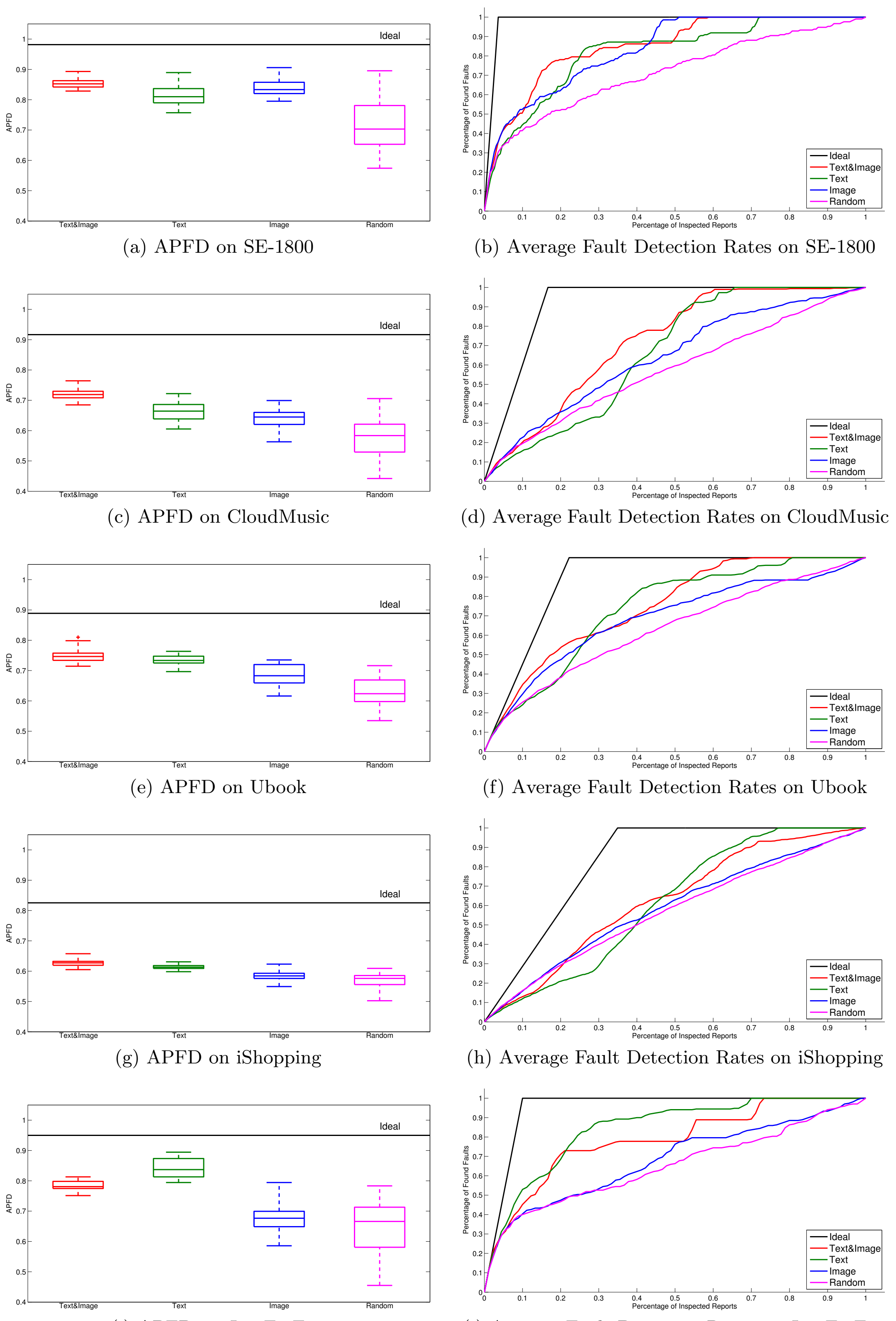
Subject Program Selection. The cost of conducting this kind of experiment is quite expensive (involved more than 300 people), the monetary budget is limited, so we conducted the experiment on only five applications. However, these five applications are widely used and publicly accessible. The functionalities of our subject applications vary widely, including music player, video player, picture editor, power monitor, and online shopping assistant. Thus, we believe these applications can be used to validate the our methods, at least to give initial indications of effectiveness and applicability.

Natural Language Selection. Admittedly, in our experiment, all of the test reports were written in Chinese, which could threaten the generalizability to other natural languages. However, the NLP techniques and text-based prioritization technique are not the focus of this work. Even though we used text-based techniques as one of our baselines, what matters to the performance of these technique is the distance built from keywords set but not the languages. As for the keyword-extraction technique, different languages have their own inherent characteristics, and thus NLP researchers have proposed keywords-extraction techniques for different languages. In future research, we will validate our technique with test reports written in English. Moreover, the focus of this work is to study the potential for image-understanding techniques to augment text-only-based techniques.

6. RELATED WORK

Bug Report Triage. As a large number of bug reports will be submitted in the software testing phase, manually triaging each of these reports will become an effort-consuming task. Bug report triage is a process that includes: prioritizing bug reports, filtering out duplicate reports, and assigning reports to the proper bug fixer. Among the various bug report triaging techniques, we address two highly relevant research areas: bug report prioritization and duplicate identification techniques.

Yu *et al.* [40] used neural networks to predict the priority of bug reports. Their technique also employs the reused data set from similar systems to accelerate the evolutionary training phase. Kanwal *et al.* [13] used SVM and Naive Bayes classifiers to assist bug priority recommendation. Tian *et al.* [36] predicted the priority of bug reports by presenting a machine learning framework that takes multiple factors including temporal, textual, author, related-report, severity, and product into consideration. By analyzing the textual description from bug reports and using text mining algorithms, Lamkanfi *et al.* [16] conducted case studies on three large-scale open source projects, and based on the result,



(a) APFD on SE-1800 (b) Average Fault Detection Rates on SE-1800

(c) APFD on CloudMusic (d) Average Fault Detection Rates on CloudMusic

(e) APFD on Ubook (f) Average Fault Detection Rates on Ubook

(g) APFD on iShopping (h) Average Fault Detection Rates on iShopping

(i) APFD on JustForFun (j) Average Fault Detection Rates on JustForFun

Figure 4: Experiment Results (averaged over 30 runs)

concluded that the technique is able to predict the severity with a reasonable accuracy.

By applying natural language processing techniques, Runeson *et al.* [30] took more textual features, including software versions, tester information, and submission date into consideration to detect duplicate bug reports. They validated this technique by conducting a large-scale experiment on industrial projects. Jalbert *et al.* [11] used the surface features, textual semantics, and graph clustering to identify the duplicate status. Besides duplicate detection, their technique is also able to rank the existing reports that are more similar with the new one. By measuring the textual semantic similarity between the test reports, Nyugen *et al.* [24] applied the topic model to detect duplicate bug reports. Podgurski *et al.* proposed the first approach to categorizing software failure reports by applying the undirected clustering on execution traces. Feng and Xin *et al.* [8, 39] adopted the multi-label classification technique to assign the bug reports into more than one classes based on the execution traces.

To assist test report prioritization in crowdsourced software testing, Feng *et al.* [9] proposed two strategies: Div and Risk. Both of the two strategies are text-based. Div is a fully automated technique, which aims at assisting the developers inspect a wide variety of test reports and to avoid duplicates and wasted effort on falsely classified faulty behavior. In this paper, we denote Div technique as TEXTDIV, and treat it as a baseline. Risk is designed to assist developers to identify test reports that may be more likely to be fault-revealing based on past observations. Because Risk requires the users to input the inspection result, *i.e.,* it is a semi-automated technique (and not fully automatic), it is a distinct category of technique, and thus we did not employ it as a baseline for evaluation.

Crowdsourced Software Testing. Mao *et al.* conducted a comprehensive survey on the crowdsourced software engineering [20], in which, they defined the crowdsourced software engineering as "the act of undertaking any external software engineering tasks by an undefined, potentially large group of online workers in an open call format." In fact, crowdsourced techniques have been widely used in industrial software testing and gained popularity in usability testing, localization testing, GUI testing, user-experience testing, and stress&performance testing. However, it is a fairly new research topic in the software-engineering research community. Liu *et al.* [19] investigated both methodological differences and empirical contrasts of the crowdsourced usability testing and traditional face-to-face usability testing. To solve the oracle problem, Pastore *et al.* [25] applied the crowdsourcing technique to generating test inputs depending on a test oracle that requires human input in one form or another. Dolstra *et al.* [7] used virtual machines to run the system under test and enable the crowd workers to accomplish expensive and semi-automatic GUI testing tasks. By introducing crowdsourced testing, Nebeling *et al.* [23] conducted an experiment to evaluate the usability of web sites and web-based services, the result of which showed that crowdsourcing testing is an effective method to validate the web interfaces.

Application of Image Understanding on Testing. In [21], Michail *et al.* proposed a static approach, GUISearch, to guide search and browsing of its source code by using the GUI of an application. They further proposed a dynamic approach to obtain an explicit mapping from high-level actions to low-level implementation by identifying execution triggered by user actions and visually describing actions from a fragment of the application displayed [3]. Kurlander *et al.* [14] introduced the notion of an editable graphical history that can allow the user to review and modify the actions performed with a graphical interface. Similarly, Michail and Xie [22] used before/after screenshots to visually describe application state at a very high level of abstraction to help users avoid bugs in GUI applications. However, images in these work are provided to developers or users directly without machine understanding.

Image-understanding techniques have been used in cross-browser issues for web applications. Cai *et al.* propose the VIPS algorithm [2], which segments a web page's screenshot into visual blocks to infer the hierarchy from the visual layout, rather than from the DOM. Choudhary *et al.* proposed a tool called WEBDIFF to automatically identify cross-browser issues in web applications. Given a page to be analyzed, the comparison is performed by combining a structural analysis of the information in the page's DOM and a visual analysis of the page's appearance, obtained through screen captures.

7. CONCLUSION

In this work, we proposed a novel technique to prioritize test reports for inspection by software developers by using image-understanding techniques to assist traditional text-based techniques, particularly in the domain of crowdsourced testing of mobile applications. We proposed approaches for prioritizing based on text descriptions, based on screenshot images, and based on a hybrid of both sources of information. To our knowledge, this is the first work to propose using image-understanding techniques to assist in test-report prioritization. In order to evaluate the promise of using image understanding of screenshots to augment text-based prioritization, we implemented our hybrid approach, as well as a text-only- and image-only-based approaches, and two baselines: an ideal best-case and a random average-case baseline. We found that prioritization, in almost all cases, is advantageous as compared to test-report inspection based on an unordered process. We also found that for most software applications that we studied, there was a benefit to using the screenshot images to assist prioritization. However, we also found that there exist a class of applications for which image-understanding may not be as applicable, and found room for improvement to narrow the gap to the hypothetical best-case ideal result.

As such, in future work, we will investigate ways to help prioritize for those classes of applications, and also identify application classes that are best suited for each type of technique. Finally, in future work we will extend the set of software systems that we use and the natural language used to write the test reports.

8. ACKNOWLEDGEMENTS

We would like to express gratitude to Peiyun Hu for valuable suggestions for this paper. This work is supported by the National Science Foundation under award CAREER CCF-1350837, and is partly supported by National Basic Research Program of China (973 Program 2014CB340702), National Natural Science Foundation of China (No. 61373013).

9. REFERENCES

[1] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[2] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. Vips: a visionbased page segmentation algorithm. Technical report, Microsoft technical report, MSR-TR-2003-79, 2003.

[3] K. Chan, Z. C. L. Liang, and A. Michail. Design recovery of interactive graphical applications. In *Proceedings of the 25th international conference on Software engineering*, pages 114–124. IEEE Computer Society, 2003.

[4] W. Che, Z. Li, and T. Liu. Ltp: A chinese language technology platform. In *Proceedings of the 23rd International Conference on Computational Linguistics: Demonstrations*, pages 13–16. Association for Computational Linguistics, 2010.

[5] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[6] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1084–1093. IEEE Press, 2012.

[7] E. Dolstra, R. Vliegendhart, and J. Pouwelse. Crowdsourcing gui tests. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 332–341. IEEE, 2013.

[8] Y. Feng and Z. Chen. Multi-label software behavior learning. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1305–1308. IEEE Press, 2012.

[9] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu. Test report prioritization to assist crowdsourced testing. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. New York: ACM*, pages 225–236, 2015.

[10] M. Ilieva and O. Ormandjieva. Automatic transition of natural language software requirements specification into formal presentation. In *Natural Language Processing and Information Systems*, pages 392–397. Springer, 2005.

[11] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.

[12] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 233–244. IEEE, 2009.

[13] A. Kao and S. R. Poteet. *Natural language processing and text mining.* Springer Science & Business Media, 2007.

[14] D. Kurlander and S. Feiner. Editable graphical histories. In *IEEE Workshop on Visual Languages*, pages 127–134. Citeseer, 1988.

[15] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[16] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 1–10. IEEE, 2010.

[17] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2169–2178. IEEE, 2006.

[18] Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 510–514. IEEE, 2009.

[19] D. Liu, R. G. Bias, M. Lease, and R. Kuipers. Crowdsourcing for usability testing. *Proceedings of the American Society for Information Science and Technology*, 49(1):1–10, 2012.

[20] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *RN*, 15:01, 2015.

[21] A. Michail. Browsing and searching source code of applications written using a gui framework. In *Proceedings of the 24th International Conference on Software Engineering*, pages 327–337. ACM, 2002.

[22] A. Michail and T. Xie. Helping users avoid bugs in gui applications. In *Proceedings of the 27th international conference on Software engineering*, pages 107–116. ACM, 2005.

[23] M. Nebeling, M. Speicher, M. Grossniklaus, and M. C. Norrie. *Crowdsourced web site evaluation with crowdstudy.* Springer, 2012.

[24] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 70–79. IEEE, 2012.

[25] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 342–351. IEEE, 2013.

[26] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE, 2003.

[27] A.-M. Popescu and O. Etzioni. Extracting product features and opinions from reviews. In *Natural language processing and text mining*, pages 9–28. Springer, 2007.

[28] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[29] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval.

International journal of computer vision, 40(2):99–121, 2000.

[30] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 499–510. IEEE, 2007.

[31] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 666–676. IEEE Press, 2015.

[32] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu. Measuring the diversity of a test set with distance entropy.

[33] E. Shutova, L. Sun, and A. Korhonen. Metaphor identification using verb and noun clustering. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 1002–1010. Association for Computational Linguistics, 2010.

[34] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society, 2011.

[35] 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*, pages 45–54. ACM, 2010.

[36] Y. Tian, D. Lo, and C. Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *2013 IEEE International Conference on Software Maintenance*, pages 200–209. IEEE, 2013.

[37] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 385–390. IEEE, 2012.

[38] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, pages 461–470. ACM, 2008.

[39] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang. Towards more accurate multi-label software behavior learning. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 134–143. IEEE, 2014.

[40] L. Yu, W.-T. Tsai, W. Zhao, and F. Wu. Predicting defect priority based on neural networks. In *Advanced Data Mining and Applications*, pages 356–367. Springer, 2010.

[41] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *Software Engineering, IEEE Transactions on*, 36(5):618–643, 2010.