UNIVERSITY OF CALIFORNIA,
IRVINE

Leveraging the Power of Crowds: Automated Test Report Processing for The Maintenance of Mobile Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Yang Feng

Dissertation Committee:
Professor James A. Jones, Chair
Professor Sam Malek
Professor David F. Redmiles

2019

© 2019 Yang Feng

To my great motherland, and to the ones who are dedicated to expanding the frontiers of human knowledge.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

CURRICULUM VITAE **x**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**

2 Background **7**
2.1 Test and Bug Report Resolution 7
2.2 Image Understanding 8
2.3 Mobile Crowdsourced Test Reports 9
2.4 Motivation 11

3 Literature Review **15**
3.1 Crowdsourced Software Testing 15
3.1.1 Crowdsourcing for Software Testing Tasks 16
3.1.2 Crowdsourcing for Testing Process 18
3.1.3 Optimization of Crowdsourced Testing 20
3.1.4 Platforms 21
3.2 Test Report Processing 22
3.2.1 Duplicate Report Detection 22
3.2.2 Bug Triaging 24
3.2.3 Bug Report Summarization 25

4 Text-Analysis-Based Crowdsourced Test Report Prioritization **27**
4.1 Preliminary 28
4.2 Technique Design 31
4.2.1 Test Report Collection 33
4.2.2 Test Report Processing 33
4.2.3 Keyword Vector Modeling 35
4.2.4 Prioritization Strategy 37

4.3 Experiment 39
4.3.1 Comparison Baselines 40
4.3.2 Datasets 41
4.3.3 Research Questions 41
4.3.4 Evaluation Metric 42
4.4 Result Analysis 44
4.4.1 Addressing RQ1 44
4.4.2 Addressing RQ2 45
4.4.3 Discussion 47
4.5 Threats to Validity 49
4.6 Conclusion 50

5 Image-Understanding-Based Crowdsoured Test Report Prioritization 51
5.1 Preliminary 52
5.2 Technique Design 53
5.2.1 Test-Description Processing 53
5.2.2 Screenshot Processing 55
5.2.3 Balanced Formula 59
5.2.4 Diversity-Based Prioritization 60
5.3 Experiment 61
5.3.1 Software Subject Programs 62
5.3.2 Prioritization Strategies 63
5.3.3 Evaluation Metrics 64
5.3.4 Experimental Setup 65
5.4 Result Analysis 66
5.4.1 Answering Research Question 1 66
5.4.2 Answering Research Question 2 67
5.4.3 Answering Research Question 3 68
5.4.4 Discussion 69
5.5 Threats to Validity 72
5.6 Conclusion 74

6 Crowdsourced Test Reports Aggregation and Summarization 75
6.1 Preliminary 77
6.2 Technique Design 78
6.2.1 Distance Calculator 79
6.2.2 Aggregator 81
6.2.3 Summarizer 82
6.3 Implementation 89
6.3.1 Architect and Design 89
6.3.2 Interface and Usage 93
6.4 Experiment 96
6.4.1 Research Questions 96
6.4.2 Dataset Description 97
6.4.3 Parameter Settings 98

6.4.4 RQ1. Effectiveness of Duplicate Aggregator. 99
6.4.5 RQ2. Effectiveness of Summarizer 101
6.4.6 RQ3. Effectiveness of **CTRAS** 103
6.5 Results and Analysis 106
6.5.1 Answering RQ1: Effectiveness of the Duplicate Aggregator 106
6.5.2 Answering RQ2: Effectiveness of the Summarizer 107
6.5.3 Answering RQ3: Effectiveness of **CTRAS** 108
6.6 Threats to Validity 110
6.7 Conclusion 111

7 An Empirical Study on Clustering Crowdsourced Test Reports of Mobile Applications Using Image Understanding 112
7.1 Study Design 112
7.1.1 Research Questions 113
7.1.2 Data Collection 114
7.1.3 Experiment Setup 114
7.1.4 RQ1. Effectiveness 115
7.1.5 RQ2. Usefulness 118
7.1.6 RQ3. Parameter Sensitivity 119
7.2 Results and Discussion 120
7.2.1 Answering Research Question 1 120
7.2.2 Answering Research Question 2 121
7.2.3 Answering Research Question 3 128
7.3 Threats to Validity 132
7.4 Conclusion 134

8 Conclusion 135
8.1 Contributions 136
8.2 Future Work 137

Bibliography 141

LIST OF FIGURES

Page

2.1 The duplicate bug report ratio of software applications 10
2.2 Procedure of Crowdsourced Testing 12

4.1 The framework of test report prioritization 31
4.2 Test report prioritization experiment results 43

5.1 Test-report processing framework 53
5.2 Four example screenshots from the CloudMusic application. 55
5.3 The corresponding feature histograms of the screenshots in Figure 5.2. 56
5.4 Test report prioritization experiment results 71

6.1 Process flow of **CTRAS** 79
6.2 A Running Example of **CTRAS** 83
6.3 Components and features of **CTRAS** 90
6.4 The test-report-list view of **CTRAS** 94
6.5 The summary view of **CTRAS** 95
6.6 Evaluation results for the Summarizer 109

7.1 APFD of Experimental Subjects (averaged over 30 runs) 123
7.2 Average Fault Detection Rates (AFDR) on Experimental Subjects (averaged over 30 runs) 125
7.3 The sensitivity of clustering results to the parameter β 130
7.4 The sensitivity of APFD to the parameter ρ 131

LIST OF TABLES

Page

2.1 Popular Crowdsourcing Platforms 13

4.1 Seven test reports from P2 32
4.2 Keywords from 7 test reports 34
4.3 Keyword Dictionary 35
4.4 Keyword Vector Model 35
4.5 Distance Matrix DM and Risk-ssessment Vector RV 36
4.6 Summary of Test Reports 41
4.7 Bonferroni Means Separation Tests 46
4.8 Linear interpolation of the average number of inspected test reports 48

5.1 Distance between screenshots of Figure 5.2 58
5.2 Experimental Software Subjects 63
5.3 One-way ANOVA Tests 70
5.4 Linear interpolation of the average number of inspected test reports 72

6.1 Statistical Information of Testing Applications 98
6.2 Details of the summarization result in RQ3 105
6.3 Interview Questions in RQ3 106
6.4 Evaluation Results for the Duplicate Aggregator: RQ1 107
6.5 Task Evaluation Results: RQ3 110

7.1 Summary of Experimental Subjects 115
7.2 Experiment Reuslts of Test Report Clustering with Different Distance Metrics 121
7.3 One-way ANOVA Tests 126
7.4 The comparison of mean clustering results under different settings of β . . . 127
7.5 The comparison of APFD mean value under different settings of ε 132
7.6 The comparison of APFD mean value under different settings of ρ 132

ACKNOWLEDGMENTS

When I was a kid, I read the books about the ones who have the remarkable ability to expand the frontiers of human knowledge and push up the technology ladder. I hold great respect for their dedication and contribution from that time, and this guides me to pursue the degree of Doctor of Philosophy through these years.

Today, when writing these words to complete my dissertation, I finally have the chance to help the education and research of the software engineering discipline. In this journey, with greatest fortunate, I have been helped, supported and guided by many people. I am eternally and profoundly grateful to them.

I would like to thank my advisor, James A. Jones, for the priceless guidance and help throughout these years. In the whole journey, over five years, his supports were unconditional and endless. When I was just admitted by UCI and met Jim for the first time in 2014, he believed in my potential and gave me strong encouragements and guidances. From that time, Jim worked for my successes patiently and tirelessly. It is his patience and commitment that changes me from a coward to a qualified Ph.D. I am very proud of working with Jim and take him as my role model for my career. I want to thank him from the bottom of my heart for giving me the chance to achieve my dream, guiding me to get out of confusion, and providing his wisdom to strengthen me.

I would like to thank my parents, Xiaoping Yu and Shuangyu Feng, for their supports of my whole education and growth. They may have no idea on my research and work, but their undying love and support have been a true inspiration and encouragement that pull me out of frustration.

I would like to thank Dr. David Redmiles and Dr. Sam Malek, who have provided me with great support and advice in the past years. They not only give me insightful and constructive feedback on my research but also wholeheartedly helps my Ph.D. studies at UC Irvine.

I would like to thank Dr. Oliver Yi Wang and Zhendong Wang, who always acts as my brother and have been helping me in and beyond research for the past years. They expanded my horizon and nurtured my interest in researching software engineering from social-technical aspects. We always fight together to overcome the obstacles in this journey. The days and nights we spent together always makes me smile and never let me down. Although situations, times and life may change over the years, our friendship is the same as before.

I would like to thank Dr. Qi Lou and Dr. Wen Shen. We were admitted by UCI in the same year and knew each other even before arrived here. We depend on and encourage each other in the whole journey. Our discussion on our dreams, mission, vision, and core values are still vivid in my mind.

Over my journey of obtaining the Ph.D. degree, I have fortunately received substantial supports and guidance from the great faculties, staffs, and affiliates of the University of California, Irvine. I would like to thank Dr. Andre van der Hoek for helping me to improve

my presentation and research; I would like to thank Dr. Cristina Videira Lopes for advising me in academic writing; I would like to thank Dr. Sameer Singh for teaching me the natural language processing and model interpretation knowledge; I would like to thank Dr. Joshua Garcia and Dr. Alfred Qi Chen for helping me to boost my research; I would like to thank Debra Brodbeck for countless help and encouragement.

I am grateful for the help and friendship with which all friends and colleagues provide me. They have been my family in University of California, Irvine, in no particular order: Mengyao Zhao, Yuanzhe Dong, Tao Wang, Yao Li, Lu An, Cilan Cai, Chang Tian, Yunfei Zhang, Xu Gao, Yang Shi, Hao Ni, Junjie Shen, Zhe Wang, Shu Kong, Peiyun Hu, Zhengli Zhao, Vijay Krishna Palepu, Kaj Dreef, Felix Lee, Junghun Kim, Jungyoun Ku, Ethan Wessel, Armin Balalaie, Francisco Servant, Nicholas DiGiuseppe, Yuan Xia, Junwei Lin, and Sumaya Almanee.

Finally, I would finally like to acknowledge that my work is supported by the National Science Foundation, under grants awards CCF-1350837, CCF-1116943 and IIS-1850067.

CURRICULUM VITAE

Yang Feng

EDUCATION

Doctor of Philosophy in Software Engineering **2019**
University of California, Irvine *Irvine, CA*

Master of Engineering in Software Engineering **2013**
Nanjing University *Nanjing, Jiangsu*

Bachelor of Engineering in Software Engineering **2011**
Nanjing University *Nanjing, Jiangsu*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2019**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2014–2015**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Rui Hao, **Yang Feng**, James A. Jones, Yuying Li, and Zhenyu Chen. Ctras: Crowdsourced test report aggregation and summarization. *The 41st International Conference on Software Engineering (ICSE).* May 2019

Wen Shen, **Yang Feng**, and Cristina V. Lopes. Multi-winner contests for strategic diffusion in social networks. *The 33rd AAAI Conference on Artificial Intelligence (AAAI). AAAI Press.* 2019

Yang Feng, Kaj Dreef, James A. Jones, and Arie van Deursen. Hierarchical Abstraction of Execution Traces for Program Comprehension. *The 26th Conference on Program Comprehension (ICPC).* May 2018

Yang Feng, James Jones, Zhenyu Chen, and Chunrong Fang. An Empirical Study on Software Failure Classification with Multi-label and Problem-Transformation Techniques. *The 11th International Conference on Software Testing, Verification and Validation (ICST).* Apr 2018

Di Liu, Xiaofang Zhang, **Yang Feng**, and James A. Jones. Generating descriptions for screenshots to assist crowdsourced testing. *The 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* March 2018

Yang Feng, James A. Jones, Zhenyu Chen, and Chunrong Fang. Multi-objective test report prioritization using image understanding. *The 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* Sep 2016

Yang Feng, Zhenyu Chen, James A. Jones, Chunrong Fang, and Baowen Xu. Test report prioritization to assist crowdsourced testing. *The 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* Aug 2015

REFEREED JOURNAL PUBLICATIONS

Ruizhi Gao, Yabin Wang, **Yang Feng**, Zhenyu Chen, and W. Eric Wong. Successes, challenges, and rethinking–an industrial investigation on crowdsourced mobile application testing. *Empirical Software Engineering (EMSE) 24, no. 2: 537-561* 2019

Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, **Yang Feng**, and Baowen Xu. How practitioners perceive automated bug report management techniques.. *IEEE Transactions on Software Engineering (TSE)* 2018

Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, **Yang Feng**, Zhenyu Chen, Baowen Xu. Smart Contract Development: Challenges and Opportunities.. *IEEE Transactions on Software Engineering (TSE)* 2019

Qingkai Shi, Zhenyu Chen, Chunrong Fang, **Yang Feng**, and Baowen Xu. Measuring the diversity of a test set with distance entropy. *IEEE Transactions on Reliability 65, no. 1: 19-27* 2015

WORKING PAPERS

Zhendong Wang, **Yang Feng**, Yi Wang, James A. Jones, David Redmiles. Unveiling Elite Developers' Activities in Open Source Projects.

Zhendong Wang, **Yang Feng**, Yi Wang, James A. Jones, David Redmiles. The Tale of Elite Developers' Activities in Five Open Source Ecosystems.

Zhendong Wang, **Yang Feng**, Yi Wang, James A. Jones, David Redmiles. Are Good Commit Messages All Alike.

Joshua Garcia, **Yang Feng**, Junjie Shen, Sumaya Almanee, Yuan Xia, Qi Alfred Chen. A Comprehensive Study of Autonomous Vehicle Bugs.

ABSTRACT OF THE DISSERTATION

Leveraging the Power of Crowds: Automated Test Report Processing for
The Maintenance of Mobile Applications

By

Yang Feng

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2019

Professor James A. Jones, Chair

Crowdsourcing is an emerging distributed problem-solving model combining human and machine computation. It collects intelligence and knowledge from a large and diverse workforce to complete complex tasks. In the software engineering domain, crowdsourced techniques have been adopted to facilitate various tasks, such as design, testing, debugging, development, and so on. Specifically, in crowdsourced testing, crowdsourced workers are given testing tasks to perform and submit their feedback in the form of *test reports*. One of the key advantages of crowdsourced testing is that it is capable of providing engineers software engineers with domain knowledge and feedback from a large number of real users. Based on diverse software and hardware settings of these users, engineers can bugs that are not caught by traditional quality assurance techniques. Such benefits are particularly ideal for mobile application testing, which needs rapid development-and-deployment iterations and support diverse execution environments. However, crowdsourced testing naturally generates an overwhelming number of crowdsourced test reports, and inspecting such a large number of reports becomes a time-consuming yet inevitable task.

This dissertation presents a series of techniques, tools and experiments to assist in crowdsourced report processing. These techniques are designed for improving this task in multiple

aspects: 1. prioritizing crowdsourced report to assist engineers in finding as many unique bugs as possible, and as quickly as possible; 2. grouping crowdsourced report to assist engineers in identifying the representative ones in a short time; 3. summarizing the duplicate reports to provide engineers with a concise and accurate understanding of a group of reports;

In the first step, I present a text-analysis-based technique to prioritize test reports for manual inspection. This technique leverages two key strategies: (1) a diversity strategy to help developers inspect a wide variety of test reports and to avoid duplicates and wasted effort on falsely classified faulty behavior, and (2) a risk-assessment strategy to help developers identify test reports that may be more likely to be fault-revealing based on past observations. Together, these two strategies form our technique to prioritize test reports in crowdsourced testing.

Moreover, in the mobile testing domain, test reports often consist of more screenshots and shorter descriptive text, and thus text-analysis-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 me to propose a novel technique based on using image understanding for multi-objective test-report prioritization. This technique employs 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.

Next, I design and implement **CTRAS**: a novel approach to leveraging duplicates to enrich the content of bug descriptions and improve the efficiency of inspecting these reports. **CTRAS** is capable of automatically aggregating duplicates based on both textual information and screenshots, and further summarizes the duplicate test reports into a comprehensive and comprehensible report.

I validate all of these techniques on industrial data by collaborating with several compa-

nies. The results show my techniques can improve both the efficiency and effectiveness of crowdsourced test report processing. Also, I suggest settings for different usage scenarios and discuss future research directions.

Chapter 1

Introduction

Crowdsourced techniques have recently gained wide popularity in the software-engineering research domain [81]. The power of crowds has been widely employed to assist various software engineering tasks, such as design [65, 89], development [67, 66], testing [143, 74] and so on. By employing a large population of crowd workers in a short period, crowdsourced techniques are capable of simulating various real usage scenarios and providing feedback of real users [81].

Such benefits are particularly ideal for software testing. Crowdsourced testing (*e.g.,* beta testing) provides validation data for a large population of varying users, hardware, and operating systems and versions. 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 [132], Testin [125], Baidu Crowd Test [8], Alibaba Crowd Test [1], MoocTest [86] and TestIO [126]) have emerged in the past five years.

Typically, in crowdsourced testing, crowdsourced workers provide testing results for devel-

opers in the form of *test reports*, which may consist of screenshots and textual content. Due to the inherent nature of crowdsourced techniques, 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 processing crowdsourced test reports automatically.

In the past decades, to improve the efficiency of processing reports, software-engineering researchers have presented many techniques. These work can be classified into two categories based on information they employed to measure the similarity between reports. The first category focuses on natural language information and leverages text-analysis techniques, such as language modeling [109, 120], text mining [39], topic modeling [98], and information retrieval [91, 27], to identify similar reports. Many later works in this direction tried to enhance the accuracy of similar report detection based on metrics on textual similarity [118, 69], identification strategies [119, 128, 46], or extra information [135]. On the other hand, as many software applications, *e.g.,* Microsoft Windows, Firefox, and Internet Explorer, have provided features to automatically record execution traces for field bugs and send reports to their producers, it is natural to use such execution information to identify similar reports. Execution traces mainly contain dynamic behaviors of the program, like stack trace, branch, or statement coverage. Similar failing traces imply the same bug [73]. Based on execution traces, researchers have designed a number of models with supervised or unsupervised learning techniques *e.g.,* [26, 31, 32, 137, 30, 99]. These models can identify failure reports with similar causes, and the classification results can be helpful for diagnosing the frequency and severity of these reports.

Even though these works have significantly improved the efficiency of dealing with test reports, they are often difficult to be applied to the specific domain of crowdsourced mobile application testing. Crowdsourced workers often prefer to take a screenshot of the problematic activity view, which is relatively easy on most mobile devices, rather than type a

long bug description, which is more difficult using the restricted small keyboard of most mobile devices. Due to these factors, the textual information of crowdsourced mobile test reports often lack sufficient details and accuracy, and execution traces are also difficult to collect due to the limited processing capability, storage, battery power, and communication of many mobile devices. While text-analysis-based methods become less effective because of short and inaccurate text descriptions, automatically identifying information from screenshots becomes critical for developers to understand reports. Images are considered to be one of the most essential and convenient communication carriers on the mobile platform [123]. Moreover, in comparison with desktop applications, screenshots of the mobile application are often well-defined and describe the activity views. The resolutions, layouts, and even features of these images are limited in a given scope, unlike desktop applications, where windows can often be reshaped and occluded.

In this thesis, I present the following statement:

> **Thesis Statement:** *The thesis of this research is that the crowdsourced test report processing technique can be partially automated in a way that is capable of saving bug identificaiton cost, improving the duplicate detection accuracy, and supporting the test report comprehension.*

Specifically, in this thesis, I evaluate the efficiency of bug identification based on the Average Percentage of Faults Detected (APFD) metric, measure the duplicate detection accuracy based on the V-measure scores, and assess the test report comprehension by conducting user study. To validate my dissertation statement, I have been investigating the following questions:

- How to organize test reports into proper order for software engineers to diagnose when

these reports are rich in textual descriptions?

- How to organize test reports into proper order for software engineers to diagnose when these reports are short of textual descriptions and rich in screenshots?
- How to summarize the critical information from a group of test reports for software engineers to diagnose, when these reports contain both textual descriptions and screenshots?
- How to leverage duplicates to enrich the content of bug descriptions and improve the efficiency of inspecting test report?

To answer the questions mentioned above, I have designed a systematical solution that consists of fully and partially automated techniques. First, to dealing with the overwhelming size of reports, I present a prioritization technique for crowdsourced mobile test reports. This method can help testers to reveal as many faults as possible as early as possible; however, it requires testers to inspect all reports. Because I observe that it is often impossible to manually inspect all test reports in a limited time, I design an approach to aggregating duplicates and summarizing them to enrich the content of bug descriptions. This aggregation and summarization approach improves the comprehensibility of reports and saves the time cost of testers in dealing with duplicates.

The primary techniques and studies presented in my dissertation are summarized as follows:

- **Text-Analysis-Based Prioritization:** I present a text-analysis-based technique to prioritize crowdsourced test reports for use in crowdsourced testing. This technique is built upon two prioritization strategies: the risk-assessment strategy (**Risk**) and the diversity strategy (**Div**). The risk-assessment strategy is designed to dynamically select the test report that has the highest probability of showing the new bug for inspection in each iteration. The diversity strategy is designed to select the diversified test reports

for inspection by maximizing the distances to already inspected test reports. For the projects that contain rich texts, these techniques can reveal as many unique bugs as possible and as quickly as possible.

- **Image-Understanding-Based Prioritization:** I 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. I design a hybrid distance computation method to measure the distance between test reports. Based on the hybrid distance, I 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 and as quickly as possible.

- **Aggregation and Summarization:** I propose a technique, named CTRAS, which is capable of leveraging the information of duplicate test reports to assist developers in comprehending test reports. Different from the conventional bug/test-report-processing techniques, instead of discouraging developers from submitting duplicates and filtering them out, my technique aims at leveraging the additional information provided by them, and summarizing both the textual and image information from the grouped duplicates to a comprehensive and comprehensible report.

- **Empirical Validation:** I conduct an empirical study to investigate the performance of crowdsourced test report clustering technique that leverages features of both the textual descriptions and screenshots. This study is performed based on six industrial projects that contain more than 1600 tests reports and 1400 screenshots. In this study, I comprehensively analyze its performance regarding different parameter settings. Based on the empirical results, I present guidance and suggestions for applying the crowdsourced test report clustering technique under different scenarios.

The organization of this thesis. Chapter 2 presents the background of this disserta-

tion. Chapter 3 presents a literature review on crowdsourced software testing and the recent advancement in the research of test report processing techniques. Chapter 4 presents the text-analysis-based prioritization technique; I validate this technique on three industrial projects. Chapter 5 presents the image-understanding-based prioritization technique; I validate this technique on six industrial projects. Chapter 6 introduces the crowdsourced test report aggregation and summarization technique, namely CTRAS, and also includes the design and implementation of the tool; I conduct Chapter 7 empirically investigates the effectiveness of image features in clustering crowdsourced test reports; Chapter 8 summarizes the conclusion and contribution of this dissertation. In this chapter, I also explore future research directions.

Chapter 2

Background

2.1 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[17], 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, I overcome the difficulties in understanding the screenshots by applying advanced image matching techniques.

2.2 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 I compare in this work are screen captures of application views rendered by different devices by different

workers for different apps. In 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 [13]. 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.

2.3 Mobile Crowdsourced Test Reports

In contrast to conventional bug repositories of desktop software applications, bug-report repositories of mobile crowdsourced testing often have higher duplicate ratios, briefer text descriptions, and richer screenshots [142].

High duplicate ratio. Crowdsourced testing is popular in mobile application testing because it enables developers to evaluate the performance of their software products under real usage scenarios. However, in practice, crowd workers are often required to finish crowdsourcing tasks in a given short time, and the number of completed tasks influences the rewards for the crowd workers. As such, crowd workers are less apt to actively filter out duplicates, and they are incentivized to submit as many reports as possible. These factors contribute to crowdsourced testing to contain a higher duplicate ratio than conventional testing.

To illustrate this difference, I present the comparison of duplicate report ratio between 12 crowdsourced testing applications and reported data of Bhattacharya *et al.*'s[12] study for

conventional bug reporting in Fig. 2.1. In this figure, the leftmost 25 blue bars represent the duplicate ratio for conventional bug reporting systems, whereas the rightmost 12 orange bars represent the duplicate ratio for crowdsourced testing. I notice that the duplicate ratio of these 12 crowdsourced testing systems ranges from 27.1% to 65.8%, and it reaches 41.7% on average. Compared with the test reports from conventional testing methods, which ranges from 4.5% to 23.1% and the average stays around 12.0%.

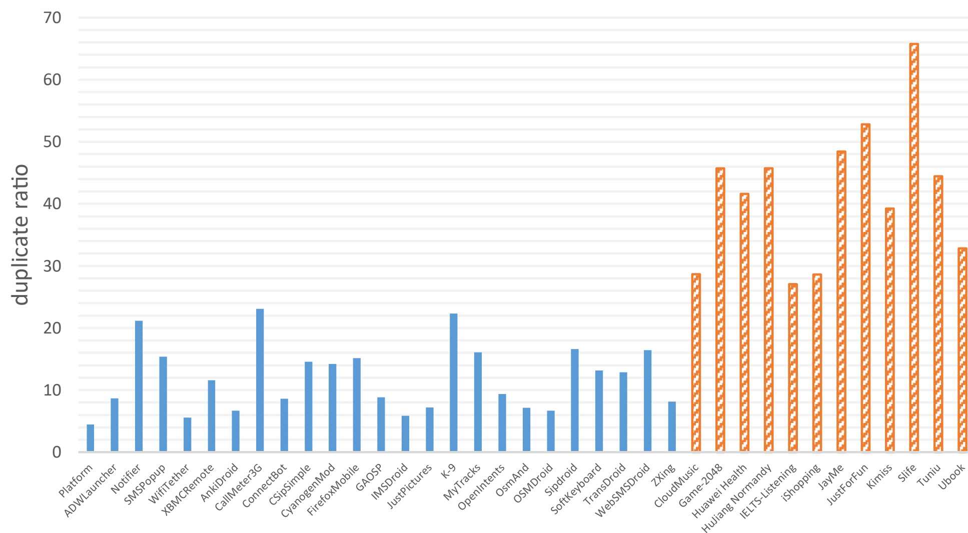


Figure 2.1: The duplicate bug report ratio of software applications

Short Text and Rich Screenshots. In addition, on almost all mobile devices, images have played a crucial role in sharing, expressing, and exchanging information. End users can easily take a screenshot and the crowd workers tend to describe bugs with a direct screenshot and short descriptions rather than verbose and complex text descriptions, largely due to the ease of taking screenshots and the relative difficulty in typing longer descriptions on mobile virtual keyboards [142]. On mobile platforms, screenshots usually capture well-defined application views, and do not suffer as many of the difficulties of desktop-application screenshots, such as varying resolutions, scaling, occlusion, and window sizes. In this context, the above issues are ameliorated, and the main problems are the prompting of the error-message dialogs, shifting of GUI elements or the fact that some elements are not displayed at all.

2.4 Motivation

The aforementioned features of mobile crowdsourced test reports, *i.e.,* high duplicate ratio, short text descriptions and rich screenshots, motivate us to propose an approach to leveraging both the text and image information from duplicate reports to enhance developers' understanding of bugs.

Moreover, a common and conventional belief in software-development practice is that the reporting of duplicate test reports is a bad practice and therefore considered harmful. The long and frequent arguments against duplicates are that they strain issue-tracking systems and waste efforts of software maintainers. Thus, based on this argument, prior researchers have proposed many techniques to assist developers in avoiding wasting time on duplicates. However, there are also arguments to the contrary. Zimmerman *et al.* [144] claim that the missing information, such as reproduction steps and environment settings, is one of the most serious problems of test reports of open-source projects. They find that developers often need to spend extra time to interact with reporters to identify the missing information and gain enough understanding of the bug. Bettenburg *et al.* [11] present empirical evidence to show that duplicates provide additional information for describing bugs and this information is helpful for fault localization and fixing. These findings fit the situation of mobile crowdsourced testing, which has been widely adopted in the quality assurance of modern mobile applications.

Crowdsourced testing involves three different related stakeholders [81]. Figure 2.2 shows the practical procedure of crowdsourced testing.

Companies and organizations often play the role of requesters. They release testing tasks and software under test on the crowdsourced testing platform. Also, they set up constraints regarding testing resources, environments, and technical requirements. Based on these constraints, platforms can match proper crowd workers for these tasks, and conversely, crowd

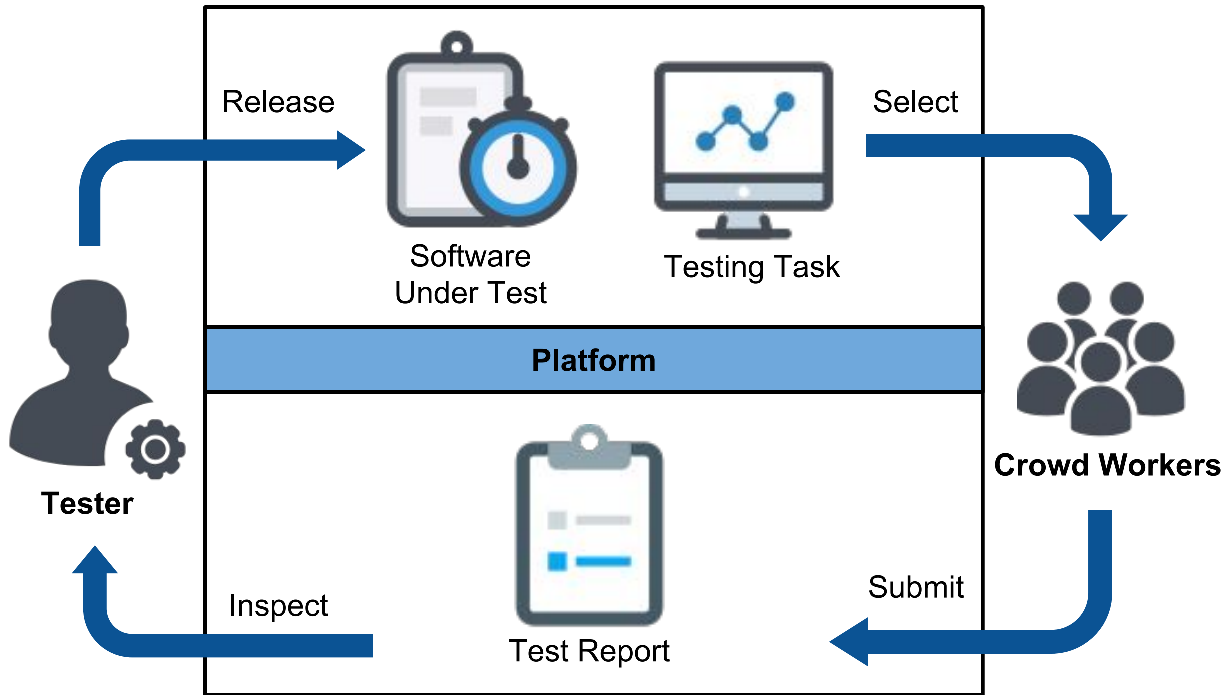


Figure 2.2: Procedure of Crowdsourced Testing

workers can also find tasks of interests. In recent years, many crowdsourcing platforms have been built for software testing tasks. Table 2.1 gives an overview of some popular industrial crowdsourced software testing platforms. By inspecting this table, I can observe the testing targets of these platforms vary widely, including performance testing, security testing, functional testing and usability testing, and some of these platforms provide the functionality to facilitate the testing process and manage the test reports. They not only enable the communication between requesters and crowd workers but also significantly improve the efficiency of each part of the whole process.

Even though these platforms lay the infrastructure for flourishing crowdsourced software testing, some features of test reports bring challenges into the inspection procedure. Crowdsourced testing is widespread in mobile application testing because it enables developers to evaluate the performance of their software products under real usage scenarios, which includes a diverse set of mobile devices and OS versions. However, in practice, crowd workers are often required to finish crowdsourcing tasks in a given short time, and the number of

Table 2.1: Popular Crowdsourcing Platforms

Name	Site	Testing Domain
uTest	utest.com	performance testing, security testing, test case management, test case recording
Testin	testin.cn	functional testing, performance testing, intelligent hardware testing
TestIO	test.io	functional testing, exploratory testing, wearables testing, IoT testing
Testflight	testflight.apple.com	functional testing, usability testing, performance testing
Bugcrowd	bugcrowd.com	security testing
Baidu Crowd Test	test.baidu.com	functional testing, usability testing, performance testing, text&image annotation
Alibaba Crowd Test	mqc.aliyun.com	performance testing, compatibility testing, test case recording
Tencent Crowd Test	task.qq.com	functional testing, usability testing, performance testing

completed tasks influences the rewards for the crowd workers. As such, crowd workers are less apt to filter out duplicates actively, and they are incentivized to submit as many reports as possible. These factors contribute to crowdsourced testing to contain a higher duplicate ratio than conventional testing [142].

Further, on almost all mobile devices, images have played a crucial role in sharing, expressing, and exchanging information. Zhang *et al.*'s research [142] finds test reports of mobile applications contain much shorter text descriptions and more screenshots in comparison with the test reports of desktop applications. For testing mobile applications, testers tend to describe bugs with a direct screenshot and short descriptions rather than tedious and complicated text descriptions, mainly due to the ease of taking screenshots and the relative difficulty in typing longer descriptions on mobile virtual keyboards [142]. The shortage of the textual description hinders applying the text-analysis-based test-report-processing techniques for mobile crowdsourced testing. While the different knowledge background and software configurations of crowd workers make it difficult for developers to fully understand the textual content of test reports, compared with the text description, screenshots can objectively

describe the operation steps and GUI exceptions, which makes the test reports more readable. These screenshots of mobile applications are usually well-defined application views, and do not suffer as many of the difficulties of desktop application screenshots, such as varying resolutions, scaling, occlusion, and window sizes. In this context, the above issues are ameliorated, and the main problems are the prompting of the error-message dialogs, shifting of GUI elements or the fact that some elements are not displayed at all. These facts motivate us to propose an automated technique to extract the information from the screenshot of crowdsourced test reports of mobile applications to assist the procedure of test report inspection for developers.

Chapter 3

Literature Review

3.1 Crowdsourced Software Testing

Crowdsourcing is a distributed problem-solving and production-organizing model brought by the Internet, which solves the problems through the integration of unknown people and machines on the Internet. Since the concept of crowdsourcing was firstly introduced in 2006 [49], it has been successfully applied in the fields of human-computer interaction, database, natural language processing, machine learning, artificial intelligence, information retrieval, and computer theory science. In software engineering, crowdsourcing techniques also have been gradually applied in almost all aspects. Particularly in software testing, a large number of online workers work together to complete test tasks, which can provide the efficient simulation of real application scenarios, shortens test life cycles and relatively reduce the cost. For these advantages, crowdsourced testing has received extensive attention from academia and industry. Many researchers focus on developing methods and techniques for improving crowdsourced testing. And meanwhile, many commercial crowdsourced testing platforms have emerged.

Some researchers have already summarized relevant studies of crowdsourcing from different perspectives. This research includes (1) optimization for specific types of testing, such as quality of experience (QoE) testing, usability testing, GUI testing; and (2) optimization for the testing process, such as test case generation, program debugging and fixing. In this chapter, I present a literature review for the research on crowdsourced testing in recent years.

3.1.1 Crowdsourcing for Software Testing Tasks

Quality of Experience Testing. Quality of experience (QoE) is used to measure the delight or annoyance of using a product or service. Because the quality of experience is difficult to be evaluated automatically, conventional QoE testing often needs to involve many users and thus become costly and time-consuming. Therefore, crowdsourcing techniques are firstly applied to QoE testing field and lead to many extensive and lasting research, such as [23, 22, 136, 48, 47, 38, 36, 37].

To ease the crowdsourced QoE testing, several research groups have designed and implemented frameworks and web-based platforms [23, 22, 136]. Chen et al. present a crowdsourceable QoE evaluation framework for multimedia content [23, 22]. Their framework derives interval-scale scores that enable subsequent quantitative analysis and QoE provisioning. Wu et al. present an evaluation framework to evaluate crowdsourcing projects regarding software quality, costs, diversity of solutions, and competitive nature in crowdsourcing [136]. This framework is designed based on a game theory model, which analyze the primary factors in the crowdsourcing process.

On the other hand, because the performance of QoE testing is often influenced by many factors, such as user habits, geographical distribution, user expectations, device environments and so on, plenty of research have been conducted to analyze these impacts and improve the QoE testing process. To solve the inherent problems in crowdsourced QoE testing, Hossfeld

et al. analyzed existing solutions. They have made a comprehensive design of the crowdsourcing competition and presented many suggestions on practices [48, 47]. Gardlo et al. compared the MOS (mean opinion score) scores of the two identical HD videos performed on the payment crowdsourcing platform (Microworkers.com) and non-payment crowdsourcing platform to examine the impact of different crowdsourcing platforms on QoE testing results [38]. The experimental results preliminarily proved the importance of distinguishing different platforms. Based on this finding, Gardlo et al. developed an exemplary application which is closely linked to the social network environment of Facebook [36, 37].

Usability Testing. Crowdsourced usability testing is also widely applied in mobile application testing. Vast crowd workers are often required to complete usability testing tasks on their own mobile devices.

To record more runtime information and user behaviors, several research groups have designed and implemented toolkits. Nebeling et al. present a toolkit, namely CrowdStudy, to provide full supports for crowdsourced usability testing of web sites [89]. This toolkit can recruit crowd workers effectively. It is also capable of evaluating the usability of websites under different conditions.

On the other hand, several early studies are conducted to compare the performance of conventional lab testing and crowdsourced testing [74, 84]. Liu et al.'s study shows that crowdsourced usability testing makes it easier to obtain data from different backgrounds globally, and these crowdsourcing testing tasks can be conducted in parallel and thus significantly reduce the cost [74]. Their study also claims both the quantity and quality of feedback for crowdsourced usability testing are slightly fewer than the conventional methods. However, Meier et al.'s research presents an opposite result to Liu and his colleagues' study [84]. Meier et al. conducted crowdsourced usability testing on MTurk to evaluate web-based tools. They find the testing results similar to the conventional lab testing methods.

GUI Testing. Modern software applications often provide user-friendly graphical user interfaces(GUIs). GUIs ease the interaction between users and software applications, and developers contribute plenty of code to implement them. The quality assurance of GUIs is critical for the entire system's safety, robustness, and usability. However, current GUI testing are strongly depend on software developers to design and maintain scripts.

To address this problem, researchers design and implement crowdsourcing tools to improve the GUI testing. Vliegendhart et al. deploy the under test software system into virtual machines and embed them in the HIT (Human Intelligence Task) of Mechanical Turk platform [133]. Crowd workers can access these virtual machines through browsers and complete the GUI testing tasks online. Dolstra et al. discuss the bias of employing virtual machines to assist crowdsourced software testing [28]. They notice that this virtual-machine-based method may make the crowdsourced GUI testing results inaccurate. To ameliorate this problem, they present a method to improve the design of crowdsourcing tasks and filter crowd workers. Similarly, to investigate the quality of data collected online, Komarov et al. conduct an empirical study on crowdsourced GUI testing on MTurk [62]. They compare the data collected on MTurk with the lab testing results. Their study shows there is no significant or substantial difference in the data collected under the two different settings. These results prove that crowdsourced GUI testing is an effective GUI testing method.

3.1.2 Crowdsourcing for Testing Process

Test case generation. Test case generation is one of the essential steps in software testing.

To harvest test cases that can reflect the real usage scenarios, several research groups gamify the testing tasks and outsourcing them to crowd workers. Gamification breaks the problems like object mutations and complex constraint into smaller puzzles and then distributes them to crowd workers. It can enhance the understandability of crowdsourced testing tasks, and

improve both the number of crowd workers and the quality of the crowdsourced work [87] Chen et al. [24] conducted a study that shows these small puzzles can be efficiently solved by crowd workers in a short time while obtaining a higher code coverage rate than the current automatic test case generation techniques, such as jCUTE [112], Randoop [92], and Pex [129].

Also, crowdsourced data can help developers generate test cases to reproduce defects. Gómez et al. propose a crowdsourcing-based method to support application developers to reproduce context-related crashes faced by end-users automatically [41]. The method retrieves crowdsourced data from the users' mobile devices, then identifies the crash mode in the execution of the application, and finally generates a test case that can reproduce the crash.

On the other hand, the knowledge of crowds is also employed to solve oracle problems in the test case generation process [10, 5]. Pastore et al. proposed CrowdOracles to organize the behaviors of current programs into assertions and publish these assertions as testing tasks on crowdsourced platforms, then crowd workers will evaluate the correctness of these assertions [94]. The experiments prove that CrowdOracles can effectively alleviate Oracle problems and advance the automation of software testing. But this method is highly dependent on the clear and understandable design of crowdsourced testing tasks. Meanwhile, this method also faces the problem of crowdsourced feedback quality [94].

Debugging. The knowledge of crowds is often employed to assist debugging open-source projects. However, this practice requires crowds to devote plenty of time on comprehending the legacy code and fixing the bug. To assist crowds in debugging code, Chen et al. proposed the crowd debugging method, which is based on the QA information in the Stack Overflow community [21]. This technique employs clone detection and matching technique to identify potential locations of bugs in the source code. And then it recommends the suspicious files and locations to crowds and collect their feedbacks online. Eventually, a bug report that

contains information like the name of the source file, the potential range of bugs, and error correction instructions are generated to assist the debugging.

According to the "foraging" behavior in program debugging [68], Petrillo et al. proposed the SDI framework (swarm debug infrastructure) and utilize the real-time information to assist program debugging [96, 97]. This framework supports the collection, storage, sharing and visualization of program debugging information, which allows multiple developers to participate in the same program debugging task, using group intelligence to accomplish the program understanding, error positioning and fixing more effectively.

3.1.3 Optimization of Crowdsourced Testing

In addition to applying crowdsourcing methods to specific types of testing or to solving specific testing problems, some researchers focus on optimizing the process of crowdsourced testing. These studies focus on improving the following two aspects: 1. the management of crowd workers, 2. task design and assignment.

Recruiting and effectively managing high-quality crowd workers is a prerequisite for conducting crowdsourced testing. Mantyla et al. have studied the impact of the scale of testers with time constraints on testing effects in crowdsourced testing [80]. Their study shows that individuals under time pressure can obtain better performance in detecting defects in comparison with individuals without time pressure. In software testing tasks, the scale of crowd workers should be should be determined based on the share of duplicate and invalid reports produced by the crowd and by the effectiveness of the duplicate handling mechanisms.

Task design and decomposition is critical for improving the efficiency and effectiveness of crowdsourced testing. Tung et al. [131] theoretically model the task assignment of collaborative testing in a crowdsourcing environment as an NP-complete problem. To improve the

work assignment, they optimize the process and transform the problem into an integer linear programming problem. Guo et al. [42] propose a distributed testing method that leverages multi-task matching technique to solve large-scale collaborative testing. This method is composed of three phases: task partitioning algorithm, greedy-based task matching algorithm, and crowdsourced testing results integration. It can select task sets from test cases dynamically and assign test cases or task sets to appropriate crowd workers. The method is validated from quality, efficiency, reliability, and scalability as well as discussed the balance between the number of testers and the quality of testing results.

3.1.4 Platforms

To optimize crowdsourced testing, many platforms and tools have been presented in the past decades. Teinum et al. present an open source user testing tool to facilitate crowdsourced testing by introducing the automated testing tools to assist the crowd workers in the testing process [124]. They adopts the client-server architecture that reduces the requirement for bandwidth and server capacity during testing activities. Nebeling et al. develop CrowdStudy that adopts a light weight client-server architecture, which can be embedded in the website through a single line of code [90]. CrowdStudy deploys a user activity tracking component on client-side and a data record and crowdsourcing component on server-side. It is capable recording behaviors of crowd workers, and automatically sending the results back to server side. Starov et al. implement a cloud testing framework for mobile systems, named CTOMS, to support crowd workers on conducting functional and UI testing of Android system applications [117]. Similarly, Liang et al. implement Caiipa, a cloud service platform, for extensible mobile application testing. Caiipa employs a fuzzy approach to extend the context of mobile application refer to crowd workers' input data and multiple measures (eg, multiple network conditions, multiple operator networks and different geographical locations, etc). iTest, designed by Yan et al., employs crowdsourcing techniques to complete testing

tasks for mobile applications and web services [139]. Xue et al present CrowdBlaze which firstly uses static analysis and automated testing to explore the application. Based on the analysis results, CrowdBlaze can provide guidance to crowd workers in designing complex cases [138]. By combining both automated testing and crowdsourced testing, CrowdBlaze can effectively improve testing coverage.

3.2 Test Report Processing

In crowdsourced software testing, requesters need to analyze and understand the feedback from crowd workers to fix bugs. Due crowdsourcing techniques inherently depend on the large workforce, the number of test reports is often overwhelming for manual inspection. Thus, software engineering researchers have proposed many techniques to assist in test report processing tasks. In this section, I mainly discuss three strands of test report processing techniques.

3.2.1 Duplicate Report Detection

Duplicate report detection is the technique of verifying whether a new test report is a duplicate of existing reports, which has been well studied by many researchers.

Previous duplicate detecting approaches are mainly based on natural language processing techniques [109], machine learning techniques [119, 118, 2, 69, 26], and information retrieval techniques [135, 120, 91, 46]. Runeson *et al.* [109] first studied the duplicate report detection problem, they investigated the natural language processing techniques to support the duplicate identification by developing and evaluating the prototype tool, the evaluation result showed that about 2/3 of the duplicates could be detected. Sureka and Jalote [120] proposed an approach to utilize the character-level N-grams model, instead of word-based model, for

the report text similarity matching and duplicate detection. In their work, they investigated the advantages of character-level features over word-based features, such as robustness to noisy text and language independence, the evaluation on a large bug report corpus from Eclipse indicated the effectiveness of their approach. To detect duplicate bug reports more accurately, Sun *et al.* [119] proposed a discriminative model, trained by Support Vector Machine, to contrast duplicate bug reports from non-duplicate duplicate bug reports, the approach improved the accuracy of state-of-the-art methods by up to 43% based on the evaluation on three large software bug repositories from Firefox, Eclipse, and OpenOffice. Nguyen *et al.* [91] introduced DBTM, a duplicate bug report detection approach based on topic model, for the duplicate detection problem. The DBTM treated bug report as textual document describing one or more technical topics and extended Latent Dirichlet Allocation to represent the topic structure of the bug reports.

For the information that can be exploited to identify duplicate bug reports, dominant works [109, 119, 120, 91] made use of the natural language information such as title, description in report. Wang *et al.* [135] first involved execution information to calculate the similarity between reports, they extracted the natural language features and execution features respectively, then retrieved potential target reports using the two kinds of features. Furthermore, contextual information, *e.g.,* product, component, priority, type, were used in some research works [118, 2, 69, 46] for accurate duplicate detection.

Lately, Deshmukh *et al.* [27] sought to apply deep learning on detecting duplicate reports, the result indicated their method can outperform the state-in-art approaches. The duplicates also exist in crash reports besides test reports, to facilitate managing crashes efficiently, Dang *et al.* [26] considered a novel bucketing method called ReBucket that clusters the crash reports based on call stack matching using the Position Dependent Model (PDM). Jiang *et al.* [56] applied the clustering technique on crowdsourced test reports. They proposed TERFUR to aggregate multiple redundant test reports into clusters to reduce the number

of reports that need to be inspected manually.

3.2.2 Bug Triaging

Bug report triage is a process that includes: prioritizing bug reports, filtering out duplicate reports, and assigning reports to the proper bug fixer. Numerous bug reports are submitted everyday for large and popular projects, and as such, manually assigning these reports to the appropriate developer is a time consuming task. Thus various automatic bug triage approaches have been proposed to assist this process, which are mainly ML and IR methods [19].

Murphy and Cubranic [88] first propose a computer-assisted bug report triage method. They utilize supervised Bayesian learning to predict suitable developers based on bug text description. Anvik *et al.* [3] present a semi-automated approach that recommends a small number of developers suitable for the triager to choose as the fixer. They apply a machine learning technique to learn what kinds of reports have been assigned to each developer and use this information to suggest potential developers.

Yu *et al.* [140] 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.* [61] used SVM and Naive Bayes classifiers to assist bug priority recommendation. Tian *et al.* [127] 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.* [64] conducted case studies on three large-scale open source projects, and based on the result, concluded that the technique is able to predict the severity with a reasonable accuracy.

Based on the investigation on the assignment and bug-tossing activities of Eclipse and Mozilla, Jeong *et al.* [53] introduce a tossing graph model based on Markov chains. It captures tossing probabilities between developers from historical tossing data. The experiment with 445,000 bug reports shows that the model can reduce tossing events by up to 72%.

Tamrawi *et al.* [122] propose Bugzie, a fuzzy set and cache-based approach for bug triaging, to reduce time and effort for a triager. The key idea of Bugzie is modeling the fixing association of developers to multiple technical aspects through fuzzy sets. Thus when a new report is submitted, Bugzie combines the fuzzy sets that are related with its content terms and prioritizes the developers based on their membership scores toward that fuzzy set.

Lately, Mani *et al.* [78] explore the feasibility of applying deep learning on the bug triaging problem. They propose a novel bug report representation approach which is based on the Deep Bidirectional Recurrent Neural Network with Attention (DBRNN-A) mechanism, and the experiment results show that this method outperforms the bag-of-words model for rank-10 average accuracy. All above approaches are supervised methods, they need to build and train models based the developers' historical bug reports before triaging bugs. In order to escape the training cost and potential problems caused by the noisy data in training dataset, some unsupervised approaches [83, 102, 72, 134] are introduced in bug triaging.

3.2.3 Bug Report Summarization

There are several works discussing the problem of bug report summarization, which are resolved in either a supervised or unsupervised way. Rastkar *et al.* [105, 104] developed a supervised learning classifier to judge whether a sentence should be included in the summary, and found that the classifier trained specifically on bug reports outperformed existing conversation-based classifiers. Jiang *et al.* [58] leveraged the authorship characteristics to

assist bug report summarization. The intuition is that, given a new bug report created by contributor A, the classifier trained over annotated bug reports by A is highly likely to generate better summaries than classifiers trained over annotated bug reports by others. Jiang *et al.* [57] proposed a PageRank-Based Summarization Technique (PRST), which utilized the information in a master report and associated duplicates to summarize the master report.

To address the problem that supervised approaches require manually annotated corpora and generated summaries may be biased towards training data, Mani *et al.* [77] applied four well known unsupervised summarization algorithms to bug report summarization. Lotufo *et al.* [75] proposed a hypothetical model of users' bug-report-reading processes, utilized it to rank sentences by their probability of being read, and include sentences with the highest probabilities into the summary.

Chapter 4

Text-Analysis-Based Crowdsourced Test Report Prioritization

In this chapter, I propose a text-analysis-based technique to prioritize test reports for use in crowdsourced testing. I adopt natural language processing (NLP) techniques, including word segmentation, and synonym replacement, to extract keywords of test reports. These keywords are used to predict failure risks of tests and calculate distances of test reports. I design two single prioritization strategies: the risk-assessment strategy (**Risk**) and the diversity-based strategy (**Div**). While the risk-assessment strategy is designed to dynamically select the test report having the highest risk-assessment score for inspection in each iteration, the diversity-based strategy is designed to select the diversified test reports for inspection by maximizing the distances to already inspected test reports. Finally, in order to reveal as many faults as possible and as early as possible, I combine the two strategies to a hybrid prioritization strategy (**DivRisk**).

In 2013, my collegues and I conducted crowdsourced testing on three projects with our industry partner, Baidu. The three projects were used to evaluate the effectiveness of test

report prioritization methods. The average percentage of faults detected (APFD) [107] and the fault-detection rate were introduced to compare four test-report prioritization methods: **Random**, **Risk**, **Div** and **DivRisk**. The **Best** and the **Worst** theoretical results of test-report prioritization were computed to discover the room for improvement of our prioritization methods. The results of empirical study indicate that: (1) **DivRisk** can outperform the random prioritization technique significantly (14.29%–34.52% improvement in terms of the APFD metric); (2) **DivRisk** can approximate the **Best** theoretical result (the gap is only 7.07% in terms of APFD) of crowdsourced testing for the mobile application testing.

4.1 Preliminary

Crowdsourced Testing Study. In 2013, my collegues and I conducted crowdsourced testing on three projects with Baidu. Testers in Baidu prepared packages for crowdsourced testing: software under test and testing tasks. Testing tasks were divided into some subtasks. The packages were distributed online, and workers bid on testing tasks. Workers were required to complete tasks in a limited time (3–5 days in our projects). Then workers submitted test reports online. Workers submitted thousands of test reports due to financial incentive and other motivations. These test reports had many false positive results (32%–80% in our projects), *i.e.,* a test report marked as "failed" that actually described correct behavior. Test reports also contained many redundant behaviors, because workers preferred to reveal simple faults instead of complex faults. Testers manually inspected all test reports to judge the workers' performance, *i.e.,* their values for revealing faults. This was a time-consuming and tedious process (nearly 12 days in our projects). Hence, it motivated me to prioritize test reports to improve the effectiveness of inspection in crowdsourced testing.

The three software systems on which my collegues and I conducted crowdsourced testing are as follows:

P1: The first project is Baidu-Input[1] on Android, which can support several input methods. Testers in Baidu provide 10 functionality sets. One crowd worker can select one functionality set, and each functionality set can be selected by at most two crowd workers, who use different mobile phones and different versions of Android.

P2: The second project is Baidu-Browser[2], which is a web browser. Testers in Baidu provide seven functionality sets for regression testing. One crowd worker can select three functionality sets.

P3: The third project is Baidu-Player[3], which is a multimedia player. Testers in Baidu provide three performance testing scenarios. One crowd worker should cover all of these three scenarios.

Workers can report other problems, such as usability and compatibility problems, in test reports.

Experience and Lessons. In total, the crowdsourced workers submitted over 2000 test reports. Of these submitted test reports, 757 were labeled as "failed" and as such were gathered for manual inspection. Upon manual inspection of all test reports that were labeled as failed, 462 of the 757 failed test reports were false positives. In other words, 462 out of 757 test reports described behavior that was either correct behavior or behavior that was considered outside the behavior of the studied software system (*e.g.,* external problems such as advertisements).

Through informal and extensive discussions with professional test engineers at Baidu, a number of observations and lessons were learned, which is summarized as follows:

1. The number of test reports submitted by crowdsourced workers quickly became chal-

[1] http://shurufa.baidu.com/
[2] http://liulanqi.baidu.com/
[3] http://player.baidu.com/

lenging to manually inspect. A larger crowdsourced testing session would have produced prohibitively many reports to manually inspect.

2. The number of false positives were more numerous than would have been expected, and presented challenges for inspection.

3. Many of the true positives and false positives were duplicates of the same underlying behavior.

4. Many crowdsourced workers performed many easy tasks and reported shallow bugs, presumably due to the incentive structures that reward quantity of submitted reports.

5. The word choice among the true positive and false positive test reports were sufficiently consistent, when accounting for word variations and synonyms.

Based on these observations by test engineers at Baidu and informed by our discussions with them, I attempted to assist with the processing and inspection of test reports, particularly for the scenario of crowdsourced testing for which the plethora of reports would be even greater. Lessons 1 and 2 simply motivate the need for some automated assistance. Lessons 3 and 4 motivate the need for looking for *diversity* in test reports — test reports that are duplicate (whether true positives or false positives) present the opportunity for wasted inspection effort and delayed identification of new true faults. Lesson 5 motivates the use of natural-language techniques to categorize test reports in an effort to automatically infer duplicate test reports.

As such, our experiences and interactions with our industrial partners motivate us to use natural language techniques (*i.e.,* NLP) to cluster test reports. Lessons 3 and 4 have motivated the need to prioritize these clusters to account for diversity (*i.e.,* our **Div** strategy).

However, because the goal of such prioritization is to reveal as many faults as early as possible, I have incorporated an additional strategy that I are calling **Risk**. The **Risk** strategy learns from already inspected test reports that were manually assessed as *true*

positive, *i.e.,* true failures that revealed true faults in the software system under test. As such, the **Risk** strategy guides the prioritization order toward other test report clusters that include similar words.

Finally, I note and recognize that the motivations for **Div** and **Risk** are, in a way, at odds — **Div** seeks to find the next test-report cluster most dissimilar from the already inspected ones, whereas **Risk** seeks to find the next test-report cluster most similar to already-inspected true positives. To account for these contrasting motivations, I created a hybrid strategy, **DivRisk**, that incorporates both **Div** and **Risk** to both maximize the distance from inspected test reports (and thus reduce inspection of duplicates and false positives) and guide the search toward the software behavior having higher likelihood to detect errors(and thus increase discovery of new true positives).

4.2 Technique Design

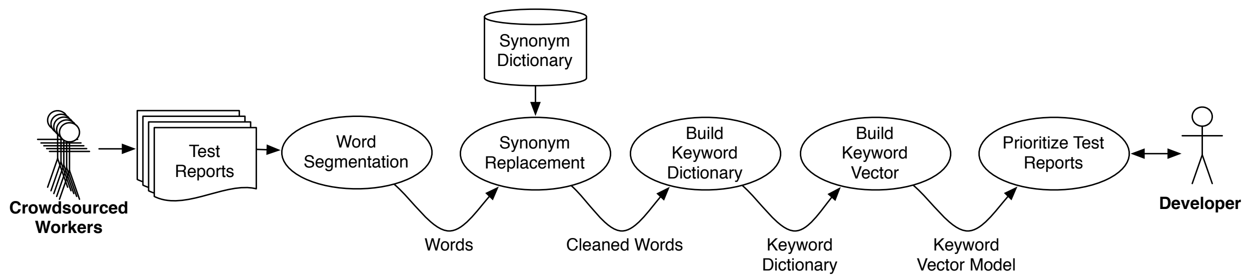


Figure 4.1: The framework of test report prioritization

In this section, I present the text-analysis-based test report prioritization methods in detail. Figure 4.1 shows the process of text analysis, which mainly contains four steps: (1) test report collection, (2) test report processing, (3) keyword vector modeling, and (4) prioritizing test reports.

Table 4.1: Seven test reports from P2

No.	*Operation*	*Description*	Result
TR1	Login renren.com in compatibility mode, click on "Personal Homepage" or "Send a Gift"(to friends), then after clicking the back button, click the forward button.	The page content is not consistent before clicking the back button with the content after clicking the forward button.	Non-fault
TR2	Enter compatibility mode, login renren.com, click one of the friend links, then click the "Personal Homepage" button , click the back button after loading.	It can go back to friend page only after clicking the back button twice.	Fault1
TR3	Open the browser, select tools→options→security, set "ad block" enhanced , input "http://soudu.org/" in the address bar.	Ads on the lower right of the page are not blocked successfully.	Fault2
TR4	In the input box in Baidu homepage , search "group buying" in compatibility mode. Next, search "ice cream" and the "red bull", double click the back button and then click the forward button once.	The page content is not consistent before clicking the back button with the content after clicking the forward button .	Fault1
TR5	Open the browser, in maximized mode, wait for the program to load and then switch the program, which means rapid and continual full-screen switch.	Sometimes the bug appears when taskbar at the bottom of the system do not disappear, especially when open other browser simultaneously . When the system is busy, the bug is more likely to occur. Move the cursor onto some task and after the appearance of the task, the system operates correctly.	Non-Fault
TR6	Select menu→options in the browser, set "ad block" closed in the security page, open the link "http://www.narutom.com/" and pop-up ads are found while loading; select menu→options in the browser, set "ad block" enhanced in the Security page, open the link "http://www.narutom.com/" to check again.	Ads blocking failed.	Non-Fault
TR7	Select menu→options in the browser , set "ad block" closed in the Security page, open the link "http://www.qidian.com/Default.aspx", and floating ads or ads around the edge of the web page are found; select menu→options in the browser, set "ad block" enhanced in the security page, open the link to check; switch the browser mode, refresh the page to check again.	When the blocking mode is switched, the number of blocked ads is not consistent with the previous one.	Non-Fault

Running Example. In order to demonstrate our methods, I sample seven test reports in P_2, as shown in Table 4.1. TR1, TR5, TR6 and TR7 are false positive test reports. That is, workers mark them as failed test reports, but testers inspect them and judge that they are not. TR2 and TR4 reveal the same fault, denoted by "Fault1". TR3 reveals another fault, denoted by "Fault2". TR7 describes the problem of some inconsistent advertisements in different modes. The test report is false positive, because it is not a fault of Baidu-Browser, but instead of the advertisement host site. Please note that all test reports are written in Chinese, and our implementation is written to handle Chinese test reports. In order to facilitate understanding, my team translates them into English in the paper, as shown in Table 4.1.

4.2.1 Test Report Collection

In our crowdsourcing projects, all test reports were committed online by workers in *Excel* files. I predefined the format of *Excel* files, such that these test reports strictly contained the fields of *operations* and *descriptions*. Note that *operations* and *descriptions* were used to inform keywords for use by the NLP techniques.

4.2.2 Test Report Processing

As shown in Figure 4.1, test report processing contains two steps: word segmentation and synonym replacement.

Word Segmentation. Word segmentation is a basic NLP task. There are many efficient tools of word segmentation for different languages [61, 35]. I adopted ICTCLAS[4] for word segmentation, which is a widely used Chinese NLP platform. *Operations* and *descriptions* of test reports were segmented into words marked with their Part-Of-Speech (POS) in the context, and then the POS tagging was applied. Hidden Markov models were used in the POS tagging [4]. Finally, the bi-gram model [14] was introduced to count the classes of words.

Synonym Replacement. In crowdsourced testing, test reports are committed by part-time workers or self-identified volunteers, who are often from different workplaces. Workers have different preferences of words and different habits of expression. Some words in test reports are meaningless for revealing faults. Hence, I filtered out these useless words (often referred to as "stop words" in the NLP literature). Prior studies show that verbs and nouns are most important to reflect the content of a document [100, 141]. Hence, I retained only verbs and nouns as candidate keywords of test reports and filtered out other words. Also, workers

[4]http://ictclas.org/

often use different words to express the same concept. For example, “thumb keyboard” and “nine-grids keyboard” refer to the same layout of keyboard in Chinese. I introduced the synonym replacement technique in NLP to alleviate this problem. In this method, I adopted the synonym library of Language Technology Platform (LTP) [20], which is largely considered as one of the best cloud-based Chinese NLP platforms.

Example. In our example, keywords are extracted from test reports, shown in Table 4.2. For example, “compatibility” indicates that TR1, TR2 and TR4 may report some compatibility problems; “menu” indicates that TR6 and TR7 may report some problems related to menu options.

Table 4.2: Keywords from 7 test reports

No.	Keywords
TR1	compatibility/n, mode/n, login/v, click/v, person/n, homepage/n, friend/n, gift/n, back/v, button/n, forwad/v, page/n, content/n
TR2	enter/v, compatibility/n, mode/n, login/v, click/v, friend/n, link/n, person/n, homepage/n, button/n, load/v, back/v, page/n
TR3	open/v, browser/n, tool/n, options/n, security/n, ads/n, block/v, select/v, address/n, input/v, page/n, corner/n, not/v
TR4	compatibility/n, mode/n, input/v, groupon/v, click/n, search/v, button/n, icecream/n, redbull/n, back/v, forward/v, result/n
TR5	open/v, browser/n, maximize/v, condition/n, wait/v, program/n, load/v, finish/v, do/v, switch/v, fullscreen/n, appear/v, system/n, task/n, miss/v, possibility/n, mouse/n, thumbnail/n, restore/v
TR6	browser/n, click/n, menu/n, options/n, security/n, page/n, ads/n, block/v, close/v, open/v, link/n, load/v, find/v, strength/n, check/v, fail/v
TR7	browser/n, click/n, menu/n, options/n, security/n, page/n, ads/n, block/v, closed/v, open/v, link/n, appear/v, floating/v, strength/n, check/v, switch/v, mode/n, refresh/v, button/n, select/v, change/v, number/n

4.2.3 Keyword Vector Modeling

The next step is to build the keyword vector model KV. I then create the risk-assessment vector RV and the distance matrix DM based on KV.

Table 4.3: Keyword Dictionary

No.	Word	Freq.	No.	Word	Freq.
K1	button	4	K2	strength	2
K3	homepage	2	K4	input	2
K5	person	2	K6	switch	2
K7	browser	4	K8	friend	2
K9	options	3	K10	login	2
K11	check	2	K12	back	3
K13	mode	4	K14	block	3
K17	click	5	K18	ads	3
K19	load	3	K20	menu	2
K21	security	3	K22	select	2
K23	link	3	K24	page	5
K25	forward	2	K26	compatibility	3

Keyword Dictionary. Keywords extracted from test reports play an important role in test report prioritization. In order to summarize the information contained within the keywords, I count the frequencies (*i.e.,* the number of occurrence) of keywords. In practice, I set a threshold ε to remove some keywords with low frequency to improve the effectiveness. As a result, a keyword dictionary is built.

Example. Table 4.3 shows the keyword dictionary of the 7 test reports. In the example, $\varepsilon = 2$, *i.e.,* all keywords with frequency < 2 in Table 4.2 are removed to produce Table 4.3.

Table 4.4: Keyword Vector Model

No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
TR1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	0	0	1	0	0	0	0	0	0	1	1	1
TR2	1	0	1	0	1	0	0	1	0	1	0	1	1	0	0	0	1	0	1	0	0	0	1	1	0	1
TR3	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0	0	1	0	0	1	1	0	1	0	0
TR4	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1
TR5	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
TR6	0	1	0	0	0	0	1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	0	1	1	0	0
TR7	1	1	0	0	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	0	0

Keyword Vector. Based on the keyword dictionary, I construct a keyword vector for each test report $tr_i = (e_{i,1}, e_{i,2}, \cdots, e_{i,m})$, in which m is the number of keywords in keyword dictionary. I compute that $e_{i,j} = 1$ if the ith test report contains the jth keyword in keyword dictionary; and $e_{i,j} = 0$ otherwise.

Example. Table 4.4 shows the keyword vector model KV of the seven test reports, in which the ith row is the keyword vector of TRi, *i.e.*, $KV(i, *) = tr_i$. KV is an $n \times m$ matrix for n test reports and m keywords in keyword dictionary.

Table 4.5: Distance Matrix DM and Risk-ssessment Vector RV

DM	TR1	TR2	TR3	TR4	TR5	TR6	TR7	RV
TR1	0	3	18	6	15	21	20	11
TR2	3	0	19	9	14	18	19	12
TR3	18	19	0	14	9	9	10	9
TR4	6	9	14	0	11	19	18	7
TR5	15	14	9	11	0	12	15	4
TR6	21	18	9	19	12	0	5	14
TR7	20	19	10	18	15	5	0	17

Risk-assessment Vector. Keywords in a test report reflect their values of revealing faults to some extent. For example, the most frequent word is "click" in Table 4.3. However, I cannot claim that "click" is the most important one for revealing faults, because "click" is a common operation in a browser. I can simply count the number of "1"s in the keyword vector as the risk-assessment value of test report, denoted by $RV(i) = \sum_{j=1}^{m} e_{i,j}$. RV is an $n \times 1$ vector for n test reports, as shown in Table 4.5.

Distance Matrix. Based on the keyword vector matrix KV, I can calculate the distances of each pair of test reports. In this work, I adopt the Hamming distance. That is, for two keyword vectors tr_i and tr_k, I count the number of different $e_{i,j}$ and $e_{k,j}$ in the corresponding position j, as the distance $\mathcal{D}(tr_i, tr_k)$. The inverse distance indicates the similarity of test reports.

Example. As a result, I construct an $n \times n$ distance matrix for n test reports. For example, the distance matrix of the seven test reports is shown in Table 4.5. $\mathcal{D}(tr_1, tr_2) = 3$, for TR1 and TR2 have 3 different keywords; $\mathcal{D}(tr_1, tr_7) = 20$, for TR1 and TR7 have 20 different keywords in the keyword dictionary.

4.2.4 Prioritization Strategy

In this subsection, I present three prioritization strategies: **Risk**, **Div** and **DivRisk**, based on the risk-assessment vector RV and the distance matrix DM, which are calculated based on the keyword vector model KV.

Risk-assessment. In order to reveal faults as early as possible, it is natural to give the top priority to inspect the test report having the highest probability to reveal bugs, *i.e.,* the test report TRi with the highest risk-assessment score computed from $RV(i)$. If multiple test reports share the highest risk-assessment score, one of them is selected for inspection. Let QTR be the *ordered set* of already inspected test reports.

Example. Based on the risk-assessment scores in Table 4.5, TR7 ($RV(7) = 17$) is first selected for inspection. Then TR6 ($RV(6) = 14$) is selected for inspection, followed by TR2 ($RV(2) = 12$). At this point of processing, $QTR = \{$TR7, TR6 TR2$\}$.

I adopt a dynamic prioritization strategy based on the risk-assessment scores and the inspection results. That is, if TRk is inspected and determined to be a true failure, all keywords of TRk in KV are increased by δ ($\delta = 0.2$ in our projects). The algorithm of updating KV is shown in Algorithm 3. Based on the new KV, the risk-assessment scores in RV are updated. That is, for each i, $RV(i) = \sum_{j=1}^{m} KV(i,j)$.

Algorithm 1: updateKV(KV, δ, k)

```
1: for all j do
2:   if KV(k, j) > 0 then
3:     for all i do
4:       KV(i, j) := KV(i, j) + δ
5:     end for
6:   end if
7: end for
8: return KV
```

Example. Because TR2 is determined to be a true failure, the risk-assessment scores of TR1, TR3, TR4 and TR5 are updated to 13(11+0.2*10), 9.2(9+0.2*1), 8.0(7+0.2*5) and 4.2(4+0.2*1), respectively. That is, for TR1, TR3, TR4 and TR5, there are 10, 1, 5 and 1 same keywords as TR2, respectively. In this way, I can get the final prioritization result of test reports: $QTR = \{$TR7, TR6, TR2, TR1, TR3, TR4, TR5$\}$.

Div. The **Div** strategy is based on the diversity principle of test selection. I prefer to select the test report tr_i with the maximal distance to QTR. Without confusion, QTR is also used to denote the set of keyword vectors $\{tr_i\}$ of already inspected test reports. The distance of tr and QTR, denoted by $\mathcal{D}(tr, QTR)$, is defined by the maximum distance between tr and each tr_i in QTR, i.e. $\mathcal{D}(tr, QTR) = Max_{tr_i \in QTR}\{\mathcal{D}(tr, tr_i)\}$.

Example. I use the seven test reports to demonstrate **Div** based on the distance matrix in Table 4.5. Initially, the test report TR7 having the highest risk-assessment score is selected, thus $QTR = \{$TR7$\}$. For the next test report, since the maximum distance is $\mathcal{D}(tr_1, QTR) = 20$, TR1 is selected. Thus, $QTR = \{$TR7, TR1$\}$. And then TR5 is selected, because $\mathcal{D}(tr_5, QTR) = 15$ is the maximum distance for the remained test reports. In this way, I can get the final prioritization result of test reports: $QTR = \{$TR7, TR1, TR5, TR3, TR4, TR6, TR2$\}$.

DivRisk. In order to reveal faults as early as possible and as many as possible, **Risk** and

Div are combined to a hybrid strategy **DivRisk**. The algorithm of **DivRisk** is shown in Algorithm 2. The risk-assessment vector RV and distance matrix DM can be calculated based on KV (Line 1–2). Initially, the test report having the highest risk-assessment score is selected for inspection (Line 4–6). Then, a candidate set CTR is constructed by selecting n_c test reports with maximum distance(s) $\mathcal{D}(tr_i, QTR)$ (Line 8). The test report having the highest risk-assessment score in CTR is selected for inspection (Line 9–11). If the inspected test report is a failed one and $\delta > 0$, the keyword vector KV will be updated by Algorithm 3 and the risk-assessment value vector RV will also be updated (Line 12–15). Finally, the prioritization result QTR is returned.

Example. I use the seven test reports to demonstrate **DivRisk**. Initially, TR7 is selected for inspection, for it has the highest risk-assessment score. $QTR = \{\text{TR7}\}$. Since the number of test reports is small in this example, I set $n_c = 2$ to facilitate demonstration. The candidate set $CTR = \{\text{TR2, TR1}\}$, for $\mathcal{D}(tr_2, QTR) = 20$ and $\mathcal{D}(tr_1, QTR) = 19$ are the two largest ones. TR2 is selected for inspection, for TR2 has a higher risk-assessment score than TR1, i.e. $RV(2) = 12 > RV(1) = 11$. In this way, I can get the final prioritization result $QTR = \{\text{TR7, TR2, TR3, TR4, TR6, TR1, TR5}\}$.

The hybrid strategy **DivRisk** will be reduced to the **Risk** if $n_c \geq |TR|$, and it will be reduced to the **Div** if $n_c = 1$. Hence, I need to set a modest number to n_c ($n_c = 8$ in our projects) for a reasonable hybrid result.

4.3 Experiment

In this study, I evaluated the test report prioritization methods: **Risk**, **Div** and **DivRisk** with three crowdsourced testing projects. In our projects, $\delta = 0.2$ and $n_c = 8$ as described above.

Algorithm 2: DivRisk(KV, n_c, δ)

1: For each i, j, $DM(i, j) := \mathcal{D}(KV(i, :), KV(j, :))$
2: For each i, $RV(i) := \sum_{j=1}^{m} KV(i, j)$
3: $TR\{1, 2 \cdots, n\}$: n is the number of rows in KV
4: $QTR := \{\text{TR}k\}$: TRk with the highest risk-asssessment score $RV(k)$ in TR
5: $QTR := QTR \cup \{\text{TR}k\}$
6: $TR := TR - \{\text{TR}k\}$
7: **while** $|TR| \neq 0$ **do**
8: CTR:= Select n_c reports TRi with the largest distances $\mathcal{D}(tr_i, QTR)$
9: Select the test report TRk with the highest risk-asssessment score in CTR for inspection
10: $QTR := QTR \cup \{\text{TR}k\}$
11: $TR := TR - \{\text{TR}k\}$
12: **if** TRk is a failed test report by inspection AND $\delta > 0$ **then**
13: $KV := updateKV(KV, \delta, k)$
14: For each i, $RV(i) := \sum_{j=1}^{m} KV(i, j)$
15: **end if**
16: **end while**
17: **return** QTR

4.3.1 Comparison Baselines

In order to verify the effectiveness of our prioritization methods, three baselines for comparison are selected. The first baseline of comparison was the **Random** strategy, which is widely used in software testing. Given a set of finite number of test reports, all possible orderings of test reports could be enumerated in theory. Supposing that I know which test reports are truly fault revealing in advance, the **Best** and the **Worst** prioritization results could be determined. For example, {TR2, TR3, TR4, TR1, TR5, TR6, TR7} is one of the best prioritization results and {TR7, TR1, TR5, TR6, TR2, TR4, TR3} is one of the worst prioritization results. In order to fairly compare these prioritization methods, the experiment was repeated 50 times to collect experimental data.

Table 4.6: Summary of Test Reports

Project	P1	P2	P3
# Report	274	231	252
# F-Report	186	47	62
% F-Report	67.88%	20.35%	24.60%
# Fault	27	22	18

4.3.2 Datasets

In our projects, all test reports were manually inspected by testers without any prioritization method. I carefully checked the inspection results again and get the final inspection results, as summarized in Table 4.6.

In Table 4.6, "# Report" is the number of test reports marked as failed by workers. These test reports were collected in the test report bucket. Testers inspected these test reports to judge whether they could reveal faults. "# F-Report" and "% F-Report" are the number and the percentage of test reports judged as failed ones by testers, respectively. In practice, some tests may reveal same faults. "# Fault" is the number of faults revealed by these test reports.

4.3.3 Research Questions

In the experiment, I investigated the following research questions.

- **RQ1**: Can our prioritization methods improve the effectiveness of test report inspection?

If the crowdsourcing requesters have no prioritization method on-hand, testers will inspect test reports in a random order. That means, testers would be motivated to adopt a prioritization method only if it can outperform the **Random** strategy. RQ1 evaluates the

effectiveness of our prioritization methods **Risk**, **Div**, and **DivRisk**.

- **RQ2**: How large is the gap between our prioritization methods and **Best**?

In practice, it is difficult to design one method that can work well in all cases. Hence, it is valuable to know the gap between the on-hand methods and the best one in theory. RQ2 evaluates the room for improvement of our prioritization methods.

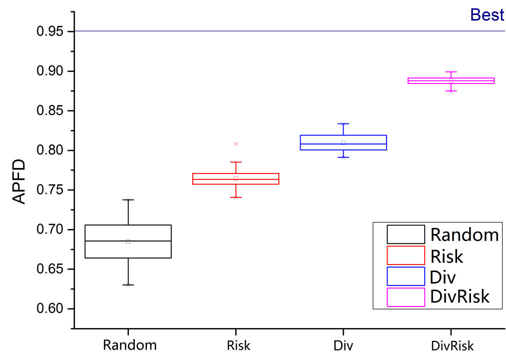
4.3.4 Evaluation Metric

In order to measure the effectiveness of prioritization methods, I adopt the APFD (Average Percentage of Fault Detected) [107], which is a widely used evaluation metric in test case prioritization [71]. For each fault, I mark the index of the first test report which reveals it. Based on the order of the test reports and information about which test reports revealed which faults, I can calculate the APFD values to measure the effectiveness of the prioritization methods. A higher APFD value indicates a better prioritization result. That is, it can reveal more faults earlier than the other methods do. APFD is formalized as follows.

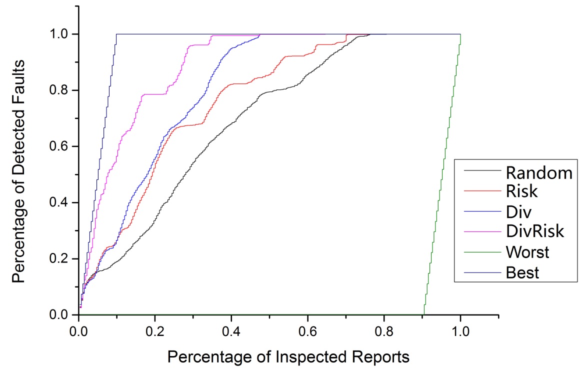
$$APFD = 1 - \frac{T_{f1} + T_{f2} + ... + T_{fM}}{n \times M} + \frac{1}{2 \times n} \tag{4.1}$$

in which, n denotes the number of test reports and 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.

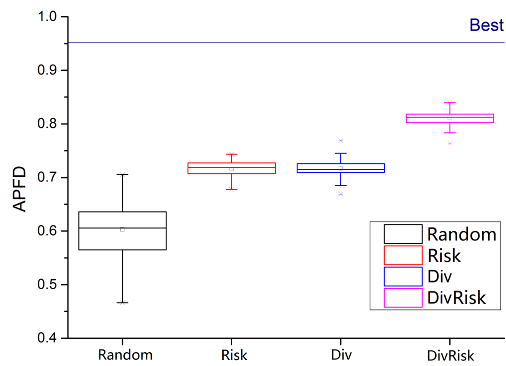
APFD indicates the fault detection rate of all test reports. However, testers cannot inspect a large number of test reports in limited time. In practice, testers will stop inspecting test reports when the limited resource is used up. At that time, testers may only inspect 25% or 50% test reports. Therefore, I evaluate how APFD varies for permutations of the same set of test reports. I use the linear interpolation [71] as follows.



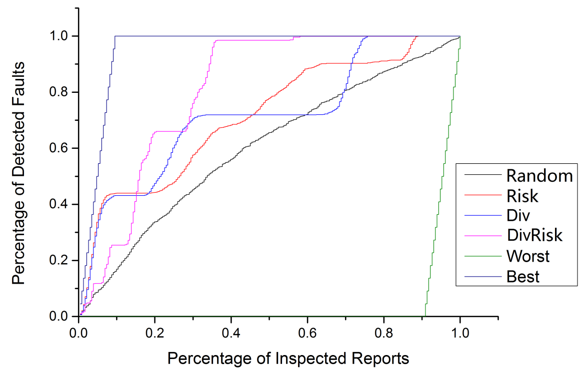
(a) APFD on P1



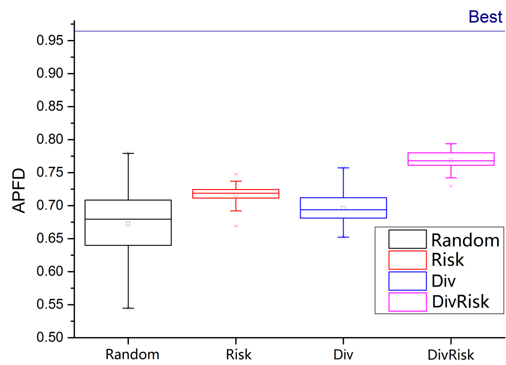
(b) Average Fault Detection Rates on P1



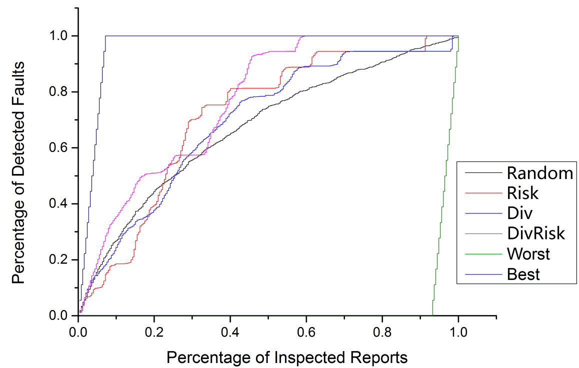
(c) APFD on P2



(d) Average Fault Detection Rates on P2



(e) APFD on P3



(f) Average Fault Detection Rates on P3

Figure 4.2: Test report prioritization experiment results

(averaged over 50 runs)

- M denotes the total number of faults revealed by all test reports.

- $p \in \{25\%, 50\%, 75\%\}$, the percentage used in our experiment.

- $Q = M \times p$, which is the number of faults corresponding to a percentage. Let $int(Q)$ and $frac(Q)$ be the integer part and fractional part of Q, respectively. If $frac(Q) = 0$, 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 is calculated as $i + (j - i) \times frac(Q)$

The linear interpolation value indicates the cost of testing to detect the given number of faults. Hence, a lower value of linear interpolation indicates a better prioritization result.

4.4 Result Analysis

In this section, I analyze the experimental results to answer RQ1 and RQ2. The results of all prioritization methods are shown in Figure 4.2. Figure 4.2 (a, c, and e) shows the box-plots of APFD results of the three projects (P1–P3) for the 50 experimental runs. The prioritization methods are shown on the horizontal axis, and the APFD values are shown on the vertical axis. The blue horizontal line in Figure 4.2 (a, c, and e) denotes the **Best** APFD value, in theory, for that subject. Figure 4.2 (b, d, and f) shows the average growth curves of the three projects (P1–P3). The percentage of the inspected test reports is shown on the horizontal axis, the the percentage of revealed faults is shown on the vertical axis.

4.4.1 Addressing RQ1

RQ1: Can our prioritization methods improve the effectiveness of test report inspection?

Based on the results shown in Figure 4.2 (a, c, and e), I can find that all of our prioritization methods outperform **Random**. In particular, **DivRisk** can outperform **Random** significantly. The hybrid strategy **DivRisk** can also improve the single strategies **Risk** and **Div**. Moreover, the box-plots show that our methods are substantially more stable than **Random**. Figure 4.2 (b, d, and f) show the average growth curves. The line charts in Figure 4.2 (b and d) show that **DivRisk** presents smooth curves to the top (**Best**).

In order to further investigate our test report prioritization methods, I do Bonferroni means separation tests for all results in Table 4.7. All F-values are very large and the all p-values are much smaller than 0.001 in Table 4.7. Compared with the **Random** strategy, the percentage of improvement of **DivRisk** ranges 14.29%–34.52%. In summary, the experimental results are encouraging for the use of the hybrid **DivRisk** strategy in practice.

In summary, I find that our prioritization methods can improve the effectiveness of test report inspection.

4.4.2 Addressing RQ2

RQ2: How large is the gap between our prioritization methods and **Best**?

Figure 4.2 shows that the hybrid strategy **DivRisk** provides the best approximation of the **Best** result in P1 and P2. For P3, **DivRisk** provides one of the best results, but there is a larger gap between its results and the **Best** result than I found for P1 and P2. For more details, I can observe the growth curves in Figure 4.2. The curves of **Best** grow very fast. The curves of **DivRisk** reach the curves of **Best** when I have inspected nearly 30% test reports in P1–P2 and nearly 60% test reports in P3.

Table 4.7 shows the gaps between our prioritization methods and **Best**. The gap between **DivRisk** and **Best** on P1 is small (7.07%). Please recall that the results of **Best** are purely

Table 4.7: Bonferroni Means Separation Tests

Method	APFD Means	Improvement $\frac{X-Random}{Random}$	Gap $\frac{Best-X}{X}$
P1: $F(3, 200) = 1549.27$, p-value ≤ 0.0001			
DivRisk	0.8879	29.66%	7.07%
Div	0.8094	18.20%	17.46%
Risk	0.7639	11.55%	24.45%
Random	0.6848	—	38.83%
Best	0.9507	38.83%	—
P2: $F(3, 200) = 474.15$, p-value ≤ 0.0001			
DivRisk	0.8113	34.52%	17.39%
Div	0.7167	18.84%	32.89%
Risk	0.7158	18.69%	33.05%
Random	0.6031	—	57.92%
Best	0.9524	57.92%	—
P3: $F(3, 200) = 90.42$, p-value ≤ 0.0001			
DivRisk	0.7686	14.29%	25.46%
Risk	0.7165	6.54%	34.58%
Div	0.6962	3.52%	38.51%
Random	0.6725	—	43.39%
Best	0.9643	43.39%	—

hypothetical and based on an unrealistically omniscient best-case analysis. Hence the result of **DivRisk** may be, or at least approximate, the best one in practice. The gaps on P2 and P3 may be, thus, acceptable (17.79% and 25.49%) in practice, and moreover, do improve the ordering of unordered or random ordering.

In order to explain the results more clearly, I calculate the linear interpolations shown in Table 4.8. Table 4.8 shows the average numbers of inspected test reports in the cases of detecting 25%, 50%, 75% and 100% faults. If the users need to reveal 25% or 50% faults, **DivRisk** is near to **Best**. However, if the users need to reveal more faults, there may be room for additional improvement. A strange phenomenon is worthy of attention: **Risk** outperforms **DivRisk** for the 25% level of inspected faults faults for P2 and the 75% level of inspected faults for P3. This result may be due to the heuristic nature of these methods and will be a subject of additional investigation in the future.

In summary, I find that our prioritization methods can provide a reasonable approximation for the theoretical Best result for some software subjects, and for other subjects provide some of the smallest gaps. In all cases that I studied, it provided better than the unordered or random ordered test reports.

4.4.3 Discussion

Method Selection. The idea of prioritization is widely used in software engineering, especially in software testing. Crowdsourced testing is usually conducted in rapidly iterative software development. In this situation, I can only inspect a subset of test reports for revealing and fixing faults before software release. Hence, test report prioritization plays a key role for a cost-effective result of crowdsourced testing. Our prioritization methods contain two key parts: the risk-assessment strategy (**Risk**) and the diversity strategy (**Div**). In software development, I need to reveal as many faults as possible, *i.e.,* **Div**. In contrast, I need to inspect the most probable "true failure" test reports early, *i.e.,* **Risk**. These two requirements of crowdsourced testing drive us to design a hybrid prioritization method **DivRisk** by combining **Risk** and **Div**. Therefore, it is not surprising that **DivRisk** can outperform the random prioritization technique significantly.

Mobile Application Testing. DivRisk shows different effectiveness in different crowdsourcing projects. The P1 project involves mobile application testing. The effectiveness of **DivRisk** in P1 was very encouraging and approximated **Best**. I reviewed the test reports in P1 and discussed with testers in Baidu. Since workers used different mobile phones and different versions of Android, they reported many compatibility problems of the application under test. The compatibility problems were easier to identified than other problems for mobile applications. Moreover, part-time workers (crowd workers here) preferred to select testing tasks of mobile applications, because it could be done anywhere and any time. There-

Table 4.8: Linear interpolation of the average number of inspected test reports

Pro.	Tech.	25%	50%	75%	100%
P1	Random	35.34	75.83	116.96	196.6
	Risk	21.12	51.64	94.19	190.7
	Div	22.39	46.80	81.27	123.5
	DivRisk	**8.885**	**20.38**	**43.09**	**99.20**
	Best	6.750	13.50	20.25	27.00
P2	Random	33.37	74.86	138.3	217.5
	Risk	**8.780**	56.22	106.4	201.2
	Div	9.200	47.46	121.2	170.1
	DivRisk	21.97	**36.24**	**66.93**	**98.30**
	Best	5.500	11.00	16.50	22.00
P3	Random	22.25	61.72	122.2	226.8
	Risk	32.90	57.14	**83.01**	230.2
	Div	23.88	61.16	104.4	246.4
	DivRisk	**14.90**	**42.44**	95.94	**145.3**
	Best	4.500	9.000	13.50	18.00

fore, it is not surprising that the prioritization results of P1, as shown in Table 4.6, were more effective than on P2 and P3. Workers committed more test reports and revealed more faults on P1 than on P2 and P3. The percentage of useful test reports (*i.e.,* F-report) is 67.88%, which was better than P2 (20.35%) and P3 (24.60%). The high quality test reports can help our test report prioritization methods, because our methods rely on keywords from test reports. As such, such crowdsourced testing and prioritization methods may be a good fit for mobile application testing.

Cost and Scalability. The total cost of test report processing in our projects is less than 10 minutes. Please note that our prioritization algorithms only involve numerical calculation on KV, RV and DM. Hence, the cost of test report prioritization methods may be negligible. The **DivRisk** algorithm is flexible. For example, I can set $\delta = 0$ in Algorithm 2, and as a result, the dynamic prioritization strategy are reduced to a static prioritization strategy. The static prioritization strategy does not rely on inspection. Hence, it can be fully automated and be more efficient, although the results may be worse. Moreover, **DivRisk** does not rely on the languages of test reports. **DivRisk** can also be used for test reports written in other

languages by using other NLP tools for other natural languages. For example, I can adopt Stanford CoreNLP[5] for word segmentation [60] and WordNet[6] for synonym replacement [85] to process English test reports, and build keyword vector model KV. Based on KV, I can use the **DivRisk** algorithm for English test report prioritization.

4.5 Threats to Validity

There are some general threats to validity in our empirical study. For example, I need more projects and different parameter values to reduce the threat to validity.

Subject selection bias:. These three crowdsourced testing projects are from industry. The software products are widely used on the Internet, and were not especially designed for our study. Due to the limited cost, I only required the industry partner to provide crowdsourced testing tasks that could be finished in 5 days. The cost of conducting our empirical study was very expensive (involving more than 200 people), so I have only three projects in our empirical study. This may threaten the generalization of our conclusions. However, the software products used in our crowdsourced testing projects are diverse. This may reduce the threat to some extent.

Crowd worker relation:. "Crowdsourcing" often requires workers from a large pool of individuals that one has no direct relationship with the others. In our experiment, the students play the roles of crowd worker, which means our crowd workers have certain social relations, and my collegues and I have only nearly 230 crowd workers. The results may be different if the crowd workers are from Internet with open calls. However, Salman *et al.* [110] found that that if a technique is new to both students and professionals, similar performance can be expected to be observed. As such, I believe that this may not be a key point for our

[5]http://nlp.stanford.edu/software/corenlp.shtml
[6]http://wordnet.princeton.edu/wordnet/

test report prioritization techniques.

Data quality: The materials of crowdsourced testing are prepared and distributed by the industrial testers. All test reports are committed by workers online directly. I checked all data and participated in the discussions of the final inspection results. In summary, all data used in this paper are from industry and the results were checked carefully by professional testers of Baidu. This may reduce the threat to the validity of data quality.

4.6 Conclusion

In this chapter, I proposed a novel test report prioritization method **DivRisk** to reduce the cost of inspection in crowdsourced testing. The keywords are extracted from test reports by using NLP techniques. These keywords construct a keyword vector model KV. I calculate the risk-assessment vector RV based on KV to predict failure risk of tests. I construct the distance matrix DM based on KV to design the diversity strategy for prioritization. The risk-assessment strategy and the diversity strategy are combined to a hybrid strategy **DivRisk** to fulfill effective test report prioritization. Three crowdsourced testing projects from industry have been used to evaluate the effectiveness of test report prioritization methods. The results of empirical study encourage us to use **DivRisk** for test report prioritization in practice, especially for mobile application testing. I also provide guidelines to extend our prioritization methods to deal with test reports written in other languages.

Chapter 5

Image-Understanding-Based Crowdsoured Test Report Prioritization

In this chapter, I proposed an technique to test-report prioritization that utilizes a hybrid analysis technique, which is both text-based and image-based. This technique is a fully automatic diversity-based prioritization technique to assist the inspection of crowdsourced mobile application test reports. To facilitate this, I capture textual and image information and measure the similarity among these artifacts. For the image analyses, I employed the Spatial Pyramid Matching (SPM) [70] technique to measure the similarity of screenshots. For the textual analyses, I used natural-language textual analysis techniques to measure the similarity of textual descriptions within test reports. Finally, I combine these similarity results using a multi-objective optimization algorithm to produce a hybrid distance matrix among all test reports. Based on these results, I 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, I 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, I received and analyzed 686 crowdsourced test reports from the crowd workers. I assessed effectiveness of our technique using the Average Percentage of Faults Detected (APFD) [107] metric and the fault detection rate. To serve as our baseline effectiveness results, I 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.

5.1 Preliminary

Even though, in practice, there could be other multi-media information that exists in the mobile test reports, such as the short operation videos and voice messages, my experience indicates that text descriptions and screenshots are the most widely used types of information. In this paper, I focus on the processing of mobile screenshots to assist the test-report prioritization procedure. I 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.

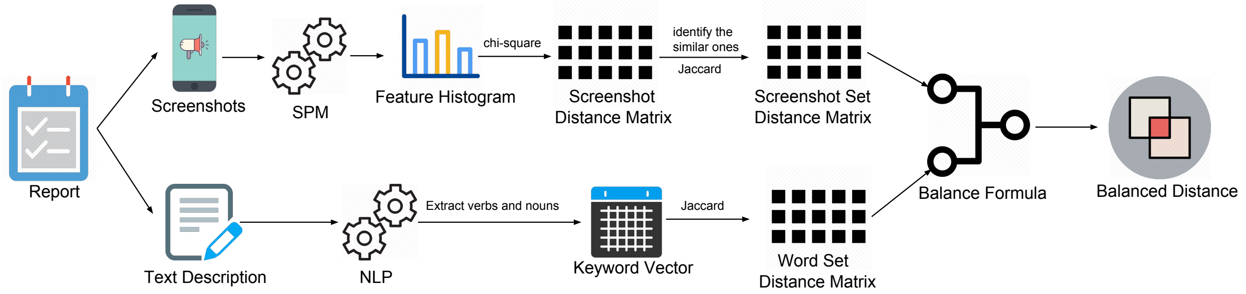


Figure 5.1: Test-report processing framework

5.2 Technique Design

This section elaborates the details of my technique. I assume the test reports only consist of two parts: text description and screenshots, which I will handle separately and finally generate the balanced distance. Figure 5.1 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 I compute the distance matrix from the test-report set, I apply various strategies to prioritize test reports.

5.2.1 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.,* [51, 109, 135, 32]), I 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, I 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 [61]. In this technique, I adopted the Language Technology Platform

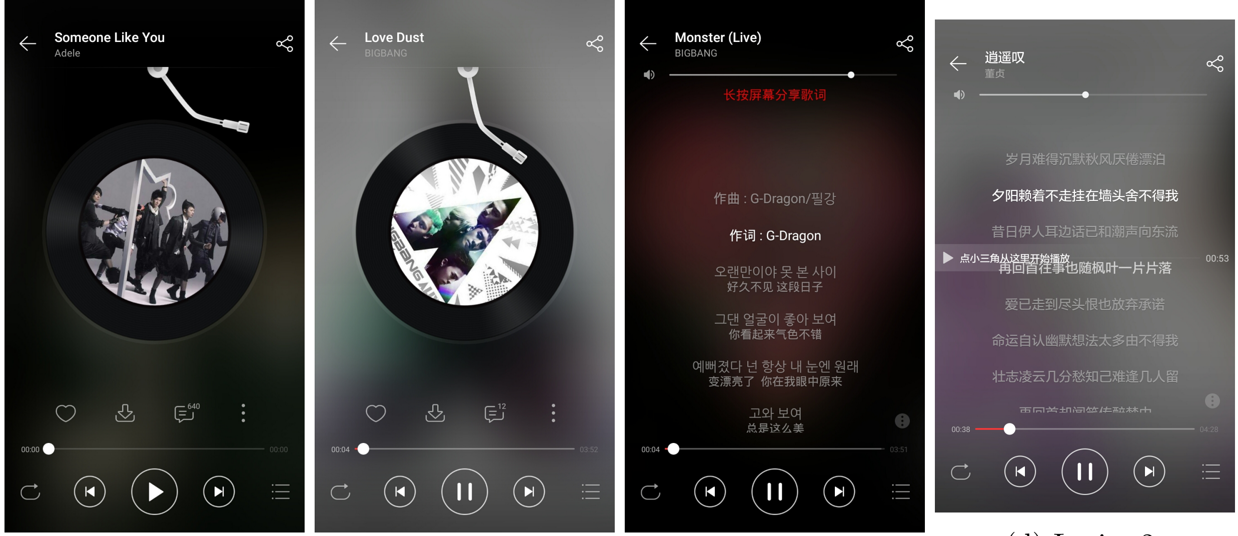
(LTP)[1] [20], which is the most widely used Chinese NLP cloud platform, to process the Chinese text descriptions. LTP segments the *Text* parts of test reports and marks each word with its Part-of-Speech (POS) for its context. In this procedure, LTP used the Conditional Random Fields (CRF) [63] model to segment Chinese words and adopted the Support Vector Machine (SVM) approach for tagging the POS. After I compute the segmentation results with the POS tags, I filter out relatively meaningless words that could negatively impact the distance calculation. According to prior works (*e.g.,* [100, 115]), verbs and nouns can reveal the main information of a document. So, to simplify the technique, I 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,[2] 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. My technique 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, I treat all of the words in the text description equally, and I 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.

[1] `http://www.ltp-cloud.com/`

[2] `http://nlp.stanford.edu/software/`



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

Figure 5.2: Four example screenshots from 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.

$$DT(r_i, r_j) = 1 - \frac{|K_i \cap K_j|}{|K_i \cup K_j|} \tag{5.1}$$

5.2.2 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. 5.1. 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.

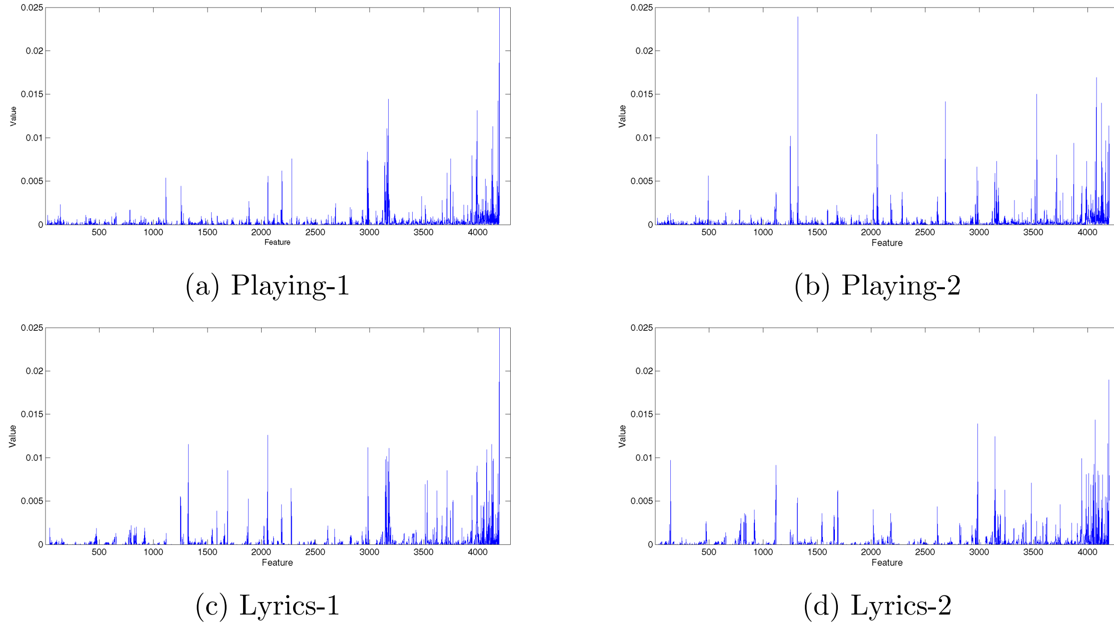


Figure 5.3: The corresponding feature histograms of the screenshots in Figure 5.2.

Feature Histogram Building. In order to compute the difference between the screenshots, I convert the screenshots into feature vectors. Bug screenshots provide not only views of buggy symptoms, but also app-specific visual appearances. I 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, I apply the Spatial Pyramid Matching (SPM) [70] to build a global representation of screenshots. Since the details of SPM are beyond this paper's topic, I 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, I apply kernel-based pyramid matching scheme to compute feature correspondence in two images.

Figure 5.2 presents four original and actual screenshots from four test reports of a popular Chinese music-playing app, CloudMusic. Figures 5.2a and 5.2b show the music-playing view of the application, and Figures 5.2c and 5.2d 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 5.2a and 5.3b have the same view layout, Figures 5.3b and 5.3d share a similar background color. If I were to directly calculate distance based on the RGB histograms, I would incorrectly get a closer distance between Figures 5.3b and 5.3d. Nevertheless, the image-understanding technique should be able to capture the similarities of the the similar views. Intuitively, Figures 5.2a and 5.3b should be identified as similar views, and Figures 5.3c and 5.3d 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 5.3.

Screenshot Distance Calculation. Using the screenshot feature histograms, a distance is computed for each pair of images. To compute such distances between feature histograms, I adopt the chi-square distance metric [108]. 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).

I 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 $Ds(s_i, s_j)$

Table 5.1: Distance between screenshots of Figure 5.2

	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

between screenshot s_i and s_j is defined as follows:

$$\begin{aligned} Ds(s_i, s_j) &= \chi^2(H_i, H_j) \\ &= \frac{1}{2}\sum_{k=1}^{d} \frac{(H_i(x_k) - H_j(x_k))^2}{H_i(x_k) + H_j(x_k)} \end{aligned} \tag{5.2}$$

Based on Equation 5.2, I obtain the distance matrix shown in Table 5.1 from the feature histograms of Figure 5.3.

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, I compute the distance between screenshot sets. To account for the diversity of display resolutions of mobile devices and user content (*e.g.,* songs, backgrounds), I 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, I 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:

$$DS(r_i, r_j) = 1 - \frac{|S_i \cap S_j|}{|S_i \cup S_j|} \tag{5.3}$$

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), I assess DS to be zero.

5.2.3 Balanced Formula

Based on above distance computations for both the textual descriptions and the screenshot sets, I combine these distances to produce a hybrid distance. I present Equation 5.4 to combine these differing distance values. Equation 5.4 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, I 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.

$$BD(r_i, r_j) = \begin{cases} 0, & \text{if } DT(r_i, r_j) = 0 \\ \alpha \times DT(r_i, r_j), & \text{if } DS(r_i, r_j) = 0 \\ (1+\beta^2)\frac{DS(r_i,r_j) \times DT(r_i,r_j)}{\beta^2 DS(r_i,r_j) + DT(r_i,r_j)}, & \text{otherwise} \end{cases} \quad (5.4)$$

5.2.4 Diversity-Based Prioritization

Using the computed balanced distance measures for all test reports, I 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.,* [25, 55]). 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 3. In the beginning, Q is empty, I 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 I get the distance values, I choose the largest one to append to Q. The whole procedure completes when $|R| = 0$.

5.3 Experiment

In this experiment, I 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?

Algorithm 3: 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

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 $RQ1$, I 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 selecting proper techniques in practice and inform the future research in this field.

5.3.1 Software Subject Programs

From November 2015 to January 2016, my collegues and I collaborated with three companies and more than 300 students to simulate a crowdsourced testing process. The five applications on which I 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.[3]
- **CloudMusic:** An application for free-sharing music as well as a music player, produced by NetEase.[4]
- **SE-1800:** A monitoring application for a power supply company, produced by Panneng.
- **Ubook:** An application for online education, produced by New Oriental.[5]

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

Table 5.2: 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

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 5.2, 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.

5.3.2 Prioritization Strategies

In my experiment, I investigate the performance of following 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 3. 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 3. 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**.

5.3.3 Evaluation Metrics

I employed the APFD (Average Percentage of Fault Detected) metric [107], 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. I present the formula to compute the AFPD value in Equation 5.5, 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.

$$APFD = 1 - \frac{T_{f1} + T_{f2} + ... + T_{fM}}{n \times M} + \frac{1}{2 \times n} \tag{5.5}$$

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, I also provide a metric to reveal the percentage of bugs that would be found at certain milestones of inspection. For this, I use linear interpolation [71] to evaluate the partial performance of each prioritization technique. I 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, I set the $p \in \{25\%, 50\%, 75\%, 100\%\}$.

5.3.4 Experimental Setup

In order to ensure the correctness of the implementation of SPM, I 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, I set $DictSize = 200$, $L = 3$, and $HistBin = 100$. For the NLP technique, because all of test reports in our experiment are in Chinese, I 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, I resize all screenshots to 480×480 pixels. Given the way 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, I implemented all of the strategies presented in Section 5.2. Particularly for the **Text&ImageDiv** strategy, I 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, I weigh the two kinds of distance equally.

5.4 Result Analysis

In this section I 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, I conducted the experiment 30 times and present the result in Figure 5.4. Figures 5.4 (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 5.4 (b, d, f, h, and j) show the average fault detection rate curves. The exact mean value of APFD is shown in Table 5.3, 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, I present the mean linear interpolation value over the 30 experiment runs in Table 5.4 to demonstrate the performance of our techniques in limited time budgets.

5.4.1 Answering Research Question 1

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

Based on the results shown in Figure 5.4 (a, c, e, g, i) and in the third column of Table 5.3, I find, to different extents, all of the three diversity-based prioritization strategies outperform **Random**. Furthermore, in Table 5.3, all *F-values* are relatively large and the *p-values* $\leq$ 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%.

In 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.

5.4.2 Answering Research Question 2

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

Figure 5.4 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. I did a deeper investigation on this problem and found what I 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 5.4, **Text&ImageDiv** outperformed the single text-

based prioritization techniques on inspecting different percentage of test report of "SE-1800", "CloudMusic" and "Ubook."

In 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, I 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.

5.4.3 Answering Research Question 3

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

The fourth column of Table 5.3 shows the gap between our strategies and the theoretical **Ideal**. I found the gap between **Text&ImageDiv** and **Ideal** vary from 15.21% to 31.98%. For more details, I can observe the growth curves in Figure 5.4. The curve of **Ideal** grows at a fast rate. The best situation reached top while the **Text&ImageDiv** only stayed around 35%.

In Summary, I find that my 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.

5.4.4 Discussion

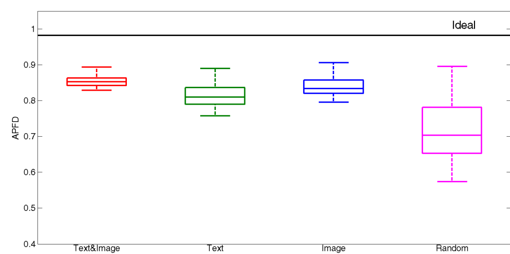
Method Selection. Reflecting on all of our experimental results, I find that image-understanding techniques can provide benefits to test-report prioritization, and that the area of such hybrid text-and-image approaches demonstrates promise. That said, I also observed that different techniques may be more or less applicable for different types of applications. Specifically, I 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 I believe are more applicable in practice than the semi-automated DivRisk and Risk techniques [32] that require the users to input the inspection result to prioritize the crowd-sourced test reports dynamically.

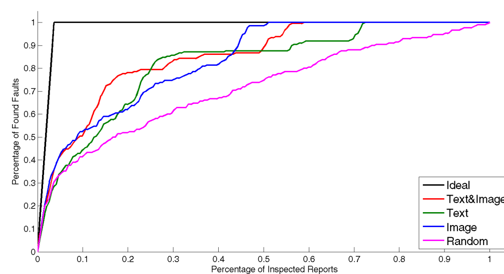
Mobile Application Testing. All of our experimentation was conducted on mobile applications, and thus I cannot state with certainty that such results would hold for other types of GUI software, such as desktop or web applications. However, I speculate that while there will likely be new and unique challenges 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 to be addressed. Overall, I speculate that the success of such image-understanding-assisted test-report prioritization techniques would likely depend on the visual complexity of the application views.

Table 5.3: One-way ANOVA Tests

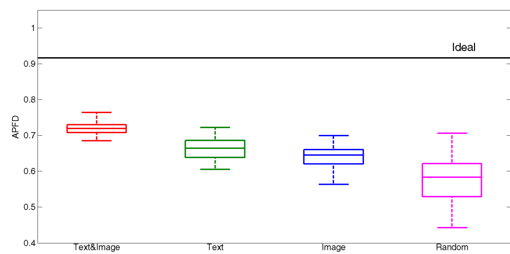
Method	APFD Means	Improvement $\frac{X-Random}{Random}$	Gap $\frac{Best-X}{X}$
SE-1800: $F(3,119) = 54.966$, *p-value* ≤ 0.001			
Ideal	0.982	37.47%	—
Text&ImageDiv	0.852	19.32%	15.21%
TextDiv	0.817	14.46%	20.10%
ImageDiv	0.836	17.04%	17.45%
Random	0.714	—	37.47%
CloudMusic: $F(3,119) = 73.170$, *p-value* ≤ 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%



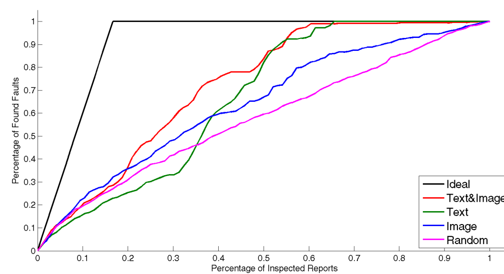
(a) APFD on SE-1800



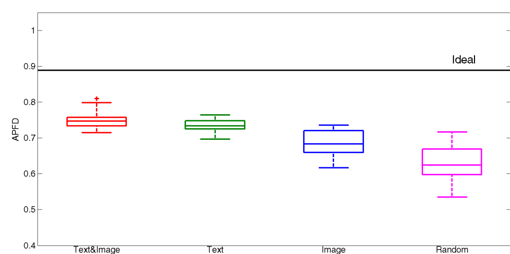
(b) Fault Detection Rates on SE-1800



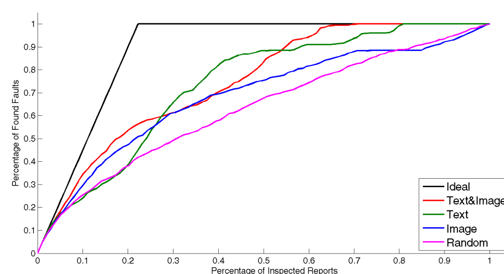
(c) APFD on CloudMusic



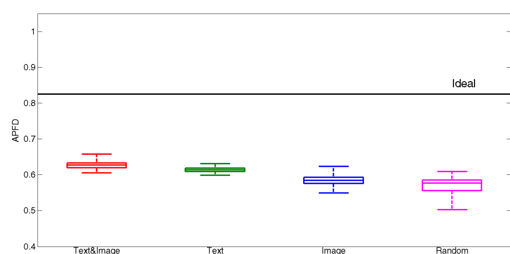
(d) Fault Detection Rates on CloudMusic



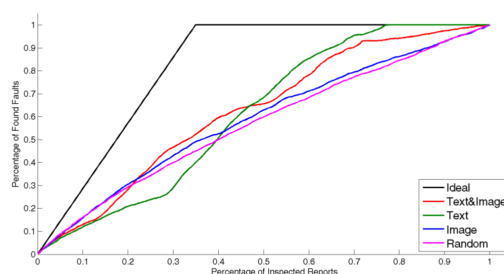
(e) APFD on Ubook



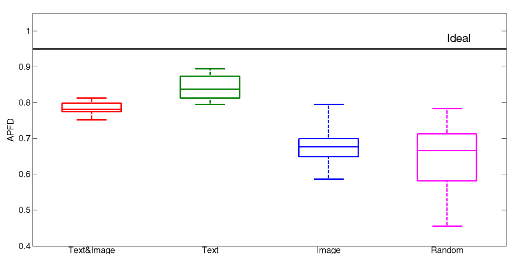
(f) Fault Detection Rates on Ubook



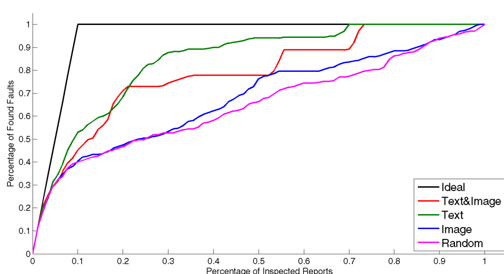
(g) APFD on iShopping



(h) Fault Detection Rates on iShopping



(i) APFD on JustForFun



(j) Fault Detection Rates on JustForFun

Figure 5.4: Test report prioritization experiment results
(averaged over 30 runs)

Table 5.4: Linear interpolation of the 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

5.5 Threats to Validity

There are some general threats to validity in our experimental results. For example, I 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, I simulated the crowdsourced mobile testing procedure to validate our techniques, in which, I 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 [81], 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.* [110], if a technique or task is new to both students and professionals, similar performance can be expected to be observed. Based on this study, I 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 I 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, I 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 I 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, I 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.

5.6 Conclusion

In this chapter, I 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. I 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, I 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. I found that prioritization, in almost all cases, is advantageous as compared to test-report inspection based on an unordered process. I also found that for most software applications that I studied, there was a benefit to using the screenshot images to assist prioritization. However, I 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, I 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 I will extend the set of software systems that I use and the natural language used to write the test reports.

Chapter 6

Crowdsourced Test Reports Aggregation and Summarization

To improve the efficiency of processing test reports, software-engineering researchers have presented many techniques to detect and summarize duplicates. Duplicate-detection techniques aim at assisting developers in identifying duplicate submissions from the repository [109, 119, 118, 2, 69, 135, 120, 91, 46, 27]. Prior research primarily focuses on two kinds of information: text descriptions [109, 119, 118, 2, 69, 120, 91, 46, 27] and execution traces [135] to reach this goal.

Further, conventional and widely used issue-tracking systems, such as Bugzilla [15], Jira [59], and Mantis [79], have provided keyword-search-based features for reporters to query similar reports to reduce duplicates [45, 103]. Also, Rastkar *et al.* presented a test-report summarization technique to assist developers to identify key sentences from test reports to reduce inspection efforts [105].

However, the settings and inherent features of *mobile* crowdsourced testing bring challenges into applying these techniques. Zhang *et al.* found that mobile test reports often contain

insufficient text descriptions and rich screenshots in comparison with desktop software [142]. Under this situation, while text-analysis-based methods become less effective because of short and inaccurate text descriptions, automatically identifying information from screenshots becomes critical for developers to understand reports. Further, while all of these techniques are built on the assumption that duplicate reports are harmful to software maintenance and aim at filtering out this information, Zimmermann *et al.* and Bettenburg *et al.* empirically found that duplicate reports are helpful for report comprehension and debugging [144, 11].

Thus, in this chapter, I propose an approach, named **CTRAS**, which is capable of leveraging the information of duplicate test reports to assist developers in comprehending test reports. Different from the conventional bug/test-report-processing techniques, instead of discouraging developers from submitting duplicates and filtering them out, our technique aims at leveraging the additional information provided by them, and summarizing both the *textual and image information* from the grouped duplicates to a comprehensive and comprehensible report.

CTRAS automatically detects and aggregates the duplicate reports by measuring the similarity of both the text description and screenshots. Based on the aggregation results, for each duplicate report cluster, it identifies the most informative report, which I call the *master report*, and summarizes the supplementary text and screenshots. These supplementaries are sorted by their weight, and **CTRAS** generates the final summarized report by combining the master report and supplementaries to provide the developer with a comprehensible overview of each test-report duplicate group.

To validate **CTRAS**, I conducted both quantitative and qualitative experiments using more than 5000 test reports collected from 12 mobile applications. The results show that **CTRAS** can accurately detect and aggregate 87% duplicate reports, by utilizing both text description and screenshot information. It improves the duplicate report detection and aggregation accuracy by 6%, and by 44% when compared to only text-based, screenshot-based methods.

Meanwhile, based on our evaluation result, **CTRAS** generates more descriptive summaries when compared to classic MCB [34, 121] and MMR [18] summarization methods. Furthermore, I conducted a task-based evaluation involving 30 participants, whose result indicates that **CTRAS** can save 30% time costs on average without losing correctness.

6.1 Preliminary

In this section, I highlight the main goals of **CTRAS** and provide definitions and notations that will be used subsequently. The overall goal is to aggregate duplicate test reports and provide a comprehensible overview of the group. Specifically for mobile software, I seek to provide a technique that is robust and effective in clustering test reports that may contain short text descriptions, but may include screenshots that exhibit failure symptoms.

For a software project with submitted crowdsourced test reports $\mathcal{R}(r) = \{r(S_i, T_i)|i = 0...n\}$, 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. Note that each the text description consists of multiple sentences, thus I have $T_i = \{t_{ij}|j = 0...m\}$, in which, t_{ij} denotes the jth sentence in the test report r_i. Similarly, I employ s_{ij} to denote the jth screenshot in the test report r_i.

Here, I list fundamental definitions as follows to ease my introduction.

Summary: The goal of **CTRAS** is to cluster duplicate reports into groups, each group of duplicate reports is a subset of $\mathcal{R}$, and then to generate a summary $\mathcal{S}$ for each group in $\mathcal{G}$. In my formulation, I define a *summary* for a group of duplicate test reports as a *master report* and a list of *supplementaries.*

Master Report: In contrast to traditional testing, crowd workers are often inexperienced

and unfamiliar with software testing. They may describe a bug from different aspects and in various ways. This fact leads the quality, writing style, and content of these reports to varying widely. Hence, I seek to find one report that provides a relatively comprehensive description of the issue for the group of duplicates. A *master report* is defined as an individual test report $r^\star \in \mathcal{G}$ that is identified as providing the most information.

Supplementary: Even though presenting the most informative one or its summary is helpful for developers to build a high-level understanding of the bug, the supplementary information, such as different software and hardware settings, diverse inputs, and various triggers, is critical for developers to gain enough knowledge of debugging and fixing. These supplementary details from other test reports in the duplicate group can provide additional insights to developers in understanding the varying conditions the lead to the issue. Hence, I seek to identify the useful information from the redundant information and summarize them into the comprehensible supplementaries. A *supplementary* is defined as the representative information item, *i.e.,* either text or screenshot, which is taken from $(\mathcal{G} - \{r^\star\})$.

Textual supplementaries are small snippets of text that present information that is not included in the master report, and thus can provide more information and greater context for developers during debugging and triaging. Likewise, image supplementaries are screenshots that differ from those included in the master report.

6.2 Technique Design

I present the process flow of **CTRAS** in the Fig. 6.1. **CTRAS** is composed of three main components: *distance calculator*, *aggregator* and *summarizer*. The *distance calculator* measures the distances among reports based on their text description and screenshots. The *aggregator* is designed for computing the distance between test reports and aggregating the

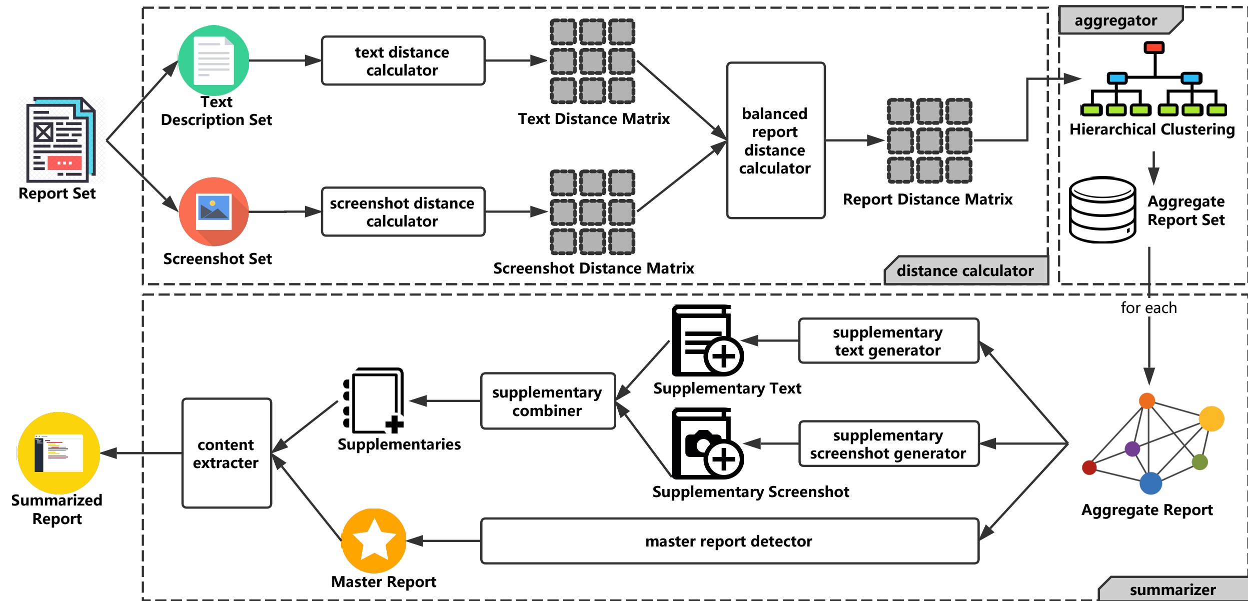


Figure 6.1: Process flow of **CTRAS**

duplicates into group $\mathcal{G}$. I use hierarchical clustering to accomplish this task. To assist developers understanding the content of the duplicate groups quickly, the *summarizer* first picks a single test report that best exemplifies the group of aggregated test reports, then it supplements information by gradually extracting *supplementary* information, which contains topics or features uncovered by the *master report*.

6.2.1 Distance Calculator

The *distance calculator* is a fundamental part of **CTRAS**. Its output, the distance matrix, is employed by both the *aggregator* and *summarizer*. **CTRAS** first calculates the text-level and screenshot-level distances separately by applying natural-language processing and image processing techniques, then combine these distances to generate whole-report-level distances.

The text-level distance calculation is composed of the following sub-steps: (1) analyzing reports' textual descriptions, and (2) measuring distances among reports under a specific strategy.

Text Preprocessing. Similar to I discussed in Chapter 4, I preprocess the text from each test report by spliting text into sentences and segmenting each sentence into words. **CTRAS** applies Language Technology Platform (LTP)[1], which is the most popular Chinese processing platform. Then, I use LTP to mark each word with its Part-of-Speech (POS), and then filter the words to retain only nouns and verbs, which can reflect the main content of a document [100, 50]. Finally, I remove the stop words based on the stop-word-list[2].

Text Distance Calculation. Considering that the words for each test report are generally short for mobile crowdsourced testing. In the implementation of **CTRAS**, I treat all words (after preprocessing) equally and apply the Jaccard Distance to calculate the distance between reports. The formula of Jaccard Distance is represented in Equation 6.1, in which $d_T(r_i, r_j)$ means the textual distance between report r_i and r_j, and T_i and T_j denotes the set of words of report r_i, r_j respectively.

$$d_T(r_i, r_j) = 1 - |T_i \cap T_j|/|T_i \cup T_j| \tag{6.1}$$

The processing of screenshots consists of two steps: (1) feature extraction and (2) distance calculation.

Feature Extraction. To get the representative features of the screenshots, I utilize the Spatial Pyramid Matching (SPM) [70] technique, whose basic approach is subdividing the image iteratively at different levels of resolution and computing the orderless histogram of local features at each sub-region, and then concatenating these spatial histograms with different weights. The final concatenated histogram represents the visual feature of the image.

Distance Calculation. To perform distance calculations of screenshot sets for test reports,

[1]https://www.ltp-cloud.com
[2]http://www.datatang.com/data/19300

I followed the conventions defined by Pele *et al.* [95]. I first utilize the chi-square distance metric to calculate the distance between two screenshots. For each pair of screenshots s_i and s_j, I calculate their distance using their histograms generated from SPM H_i and H_j and the distance measure in Equation 6.2. Considering that in practice there are often more than one screenshot in a test report, I set a threshold ℓ_S to identify whether two screenshots are describing the same scenario. Using Equation 6.2 and ℓ_S to define a corpus of representative screenshots. As such, I identify the unique representative screenshots from each test report, then calculate the final screenshots-set distance between two test reports with the Jaccard distance metric. The Jaccard distance formula is listed in the Equation 6.3, in which S_i, S_j are the representative set of reports r_i and r_j respectively.

$$d_S^*(s_i, s_j) = \| H_i, H_j \|_2 = \sqrt{\sum_{k=1}^{n} (H_i^k - H_j^k)^2} \tag{6.2}$$

$$d_S(r_i, r_j) = 1 - |S_i \cap S_j| / |S_i \cup S_j| \tag{6.3}$$

Once **CTRAS** has calculated the textual distance d_T and the screenshot distance d_S, it can compute the overall test-report distance for each pair of test reports. **CTRAS** employs the balanced formula, which I have introduced in the Chapter 5, as the report distance calculation formula.

6.2.2 Aggregator

Based on the distance matrix produced by the *distance calculator*, the *aggregator* is capable of measuring the similarity between test reports and further grouping the duplicates. Considering that in practice the number of groups cannot be predicted, I adopt Hierarchical

Agglomerative Clustering (HAC) [29], which can determine the number of groups based on a threshold distance value, to group the duplicates.

Hierarchical Agglomerative Clustering method treats all reports as independent clusters, uses report distance as input and groups test reports over a variety of scales by creating a hierarchical cluster tree, I utilize threshold t to determine the cut point and assign all the test reports below the cut point to a single cluster. Particularly, I adopt Single-linkage algorithm in the implementation of **CTRAS**, which takes the shortest distance between all test reports in two clusters as their cluster distance. Through Single-linkage I can attain relatively abundant and comprehensive duplicate reports.

6.2.3 Summarizer

The *summarizer* is the core component of **CTRAS**. For each duplicate test-report group $\mathcal{G}$, it performs the following three steps to generate a summary to assist developers in forming a comprehensive understanding over all reports in $\mathcal{G}$: (1) identifying the master report $r^{\star}$, (2) generating supplementaries, and (3) forming and generating final summaries.

To ease explanation, I take a real group containing six test reports in our empirical study as an example, and I illustrate each substep in Fig. 6.2. I list the six similar reports in the exemplary group in Fig. 6.2-a. All of these six reports are describing the same bug, *i.e.,* logging in error the App via a third-party tool, from different respects. As shown in Fig. 6.2-a, there are six similar reports describing the error of third-party login in the exemplary group. Each report of this group consists of its basic attributes, such as report names, creation time, as well as its textual description and several screenshots.

Master Report Identification. To help the developers concisely understand the topic of the test reports within the group, I identify the *master report* $r^{\star}$ in the first step. I abstract

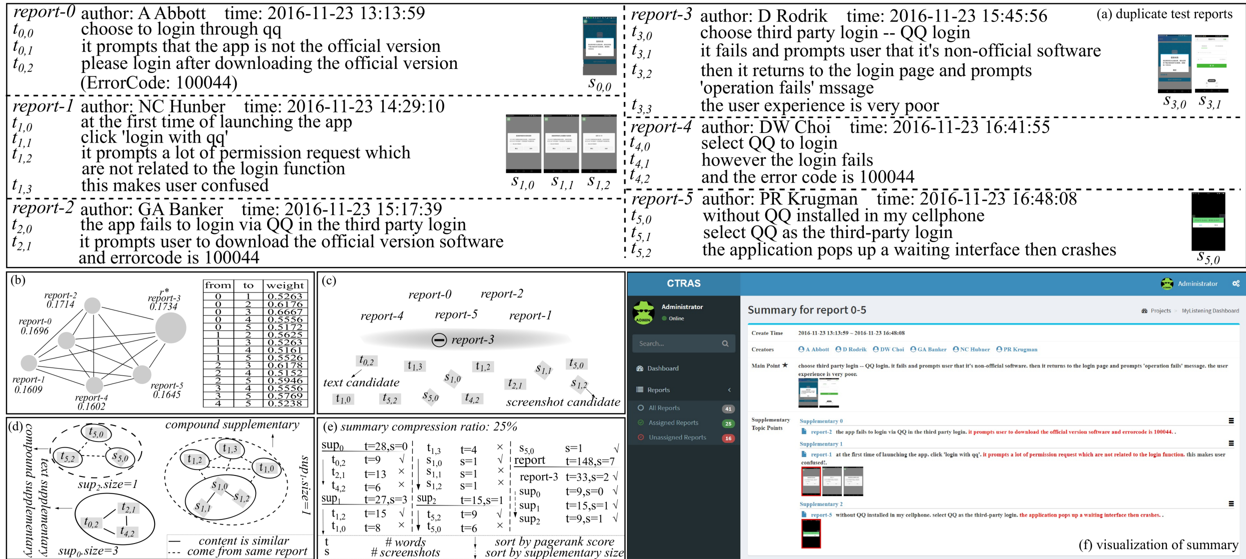


Figure 6.2: A Running Example of **CTRAS**

each test-report group into a graph, within which each node represents an individual report. The weight of edges between two nodes indicates the similarity between these two reports. Thus, **CTRAS** can apply the PageRank [93] algorithm, which can compute a numerical rank score and measure the importance for each node within a weighted graph, on each test-report group. **CTRAS** identifies the test report having the greatest page rank score as the master report for the group.

Example: The graph representing the exemplary group is illustrated in Fig 6.2-b, and the table shows the hybrid similarity between each pair of these reports. **CTRAS** compute the weight of each node by applying the PageRank algorithm. **CTRAS** can find that *report-3* has the highest weight, and thus it is selected as the master report (labeled with $r^\star$) of these six reports. Through reading the contents of *report-3*, developers can reach a high-level understanding of the whole test report group, *i.e.,* , there is a bug that users can't login the App through QQ social login service as it fails the authentication and is regarded as non-official software.

Supplementary Generation. Even though I have identified the *master report* $r^\star$ and

helped the developer get the most informative report within the group $\mathcal{G}$, describing the same bug from other perspectives and providing supplementary materials is critical for developers in fixing the bug properly. Thus, I further analyze the other reports and identify the content that is shared among them as the supplementary points. In this procedure, **CTRAS** perform two substeps for identifying the supplementaries: (1) identifying candidate items from $(\mathcal{G} - \{r^\star\})$; (2) grouping the candidate items to form a *supplementary*.

Candidate Item Identification. Because the sentence is considered to be the immediate integral unit in linguistic theory, a number of prior research efforts that aimed at analyzing test/bug reports to assist developers in understanding the bug descriptions have selected the sentence as the basic unit [105, 75, 77]. I thus also do so accordingly and measure the similarity between two sentences by computing the Jaccard Distance between their keyword vectors. Jaccard Distance is a useful metric to compare the similarity and diversity of two sets, which is shown in Equation 6.4. In this equation, t and t' denote the keyword sets of two sentences, $|t \cap t'|$ denotes the number of words in the intersection of both sets, and $|t \cup t'|$ denotes the number of words in the union of both sets. Regarding the screenshot, as I discussed in Section 6.2.2, I adopt the method described in section 5.2 and corresponding parameters to identify different screenshots.

Given a test-report group $\mathcal{G}$ and its *master report* $r^\star$, to generate supplementaries for $r^\star$, the first step is to identify candidate items, *i.e.,* sentences and screenshots, which are NOT included in $r^\star$ from $(\mathcal{G} - \{r^\star\})$. From the set $(\mathcal{G} - \{r^\star\})$, I extract all singleton items, *i.e.,* individual sentences and screenshots, to get the set of sentences $T = \{t_{ij}\}$ and the set of screenshots $S = \{s_{ij}\}$ of $(\mathcal{G} - \{r^\star\})$. Similarly, I can get the set of sentences and set of screenshots from $r^\star$, and I denote them $T^\star = \{t^\star_j\}$ and $S^\star = \{s^\star_j\}$ respectively. For each sentence, of which the keyword set is $\{t\}$ in T, if not existing any element in $S^\star$ having the $J(t, t^\star)$ is smaller than the predefined threshold value, I consider it is a candidate sentence

for $r^\star$. Similarly, given S and $S^\star$, I can identify the candidate screenshot for $r^\star$.

$$J(t, t') = 1 - |t \cap t'|/|t \cup t'| \tag{6.4}$$

Example: As shown in Fig. 6.2-c, I label the text items with rectangles and the screenshot items with diamonds. **CTRAS** identified eight candidate sentences and four candidate screenshots from the exemplary group. For example, $t_{0,2}$ is a candidate sentence as it supplements the "expected result" information (*i.e.,* "ErrorCode: 100044") for $r^\star$(*i.e., report-3*); $t_{5,0}$ expresses a special case that "without QQ installed" and $s_{5,0}$ is a candidate screenshot for this case.

Candidate Item Refinement. Through candidate item identification, all candidate items, which are not similar to any item in the master report, are identified. However, some of these items may be too brief to understand, or they are similar with each other. Therefore, I refine these candidate items into concise and representative *supplementaries*.

The refinement process of **CTRAS** consists of three sub-steps: 1. clustering similar candidate items; 2. providing an additional clustering of the candidate clusters; and 3. adding weights to candidates within the clusters. In the first step, **CTRAS** groups similar candidate items to remove redundancy and improve the conciseness of the supplementary. It applies hierarchical agglomerative clustering on the set of candidate sentences to form some candidate sentence clusters. For each candidate sentence cluster, it further applies the PageRank algorithm to compute the weight for every element, and then pick up the one with highest weight as the representative.

Similarly, **CTRAS** can apply the same strategy on the candidate screenshot set and get the candidate screenshot clusters. Moreover, it records the origin information for each candidate, so that it can map candidates to reports and vice versa. This information is not only useful for further aggregating candidate sentence clusters in the next sub-step but also helpful

for developers to track back original reports in practice. Note that, even though **CTRAS** conducts clustering over the candidate item sets and get the candidate item clusters, clusters may have only one element because some items contain distinct information.

Next, it further groups the candidate clusters. The purpose of this step is to restore context of the candidate sentences and screenshots. This extra level of grouping is useful particularly for singleton clusters. **CTRAS** merges clusters based on the origin information: clusters that contain candidates originating from much of the same test reports are clustered. Because restoring the sentence and screenshot to semantic context is helpful for improving the understandability, this step is designed for grouping all candidate item clusters from the same report. The benefits of doing this is two-fold: 1. connecting the singleton clusters from the same report can form semantic context, which is capable of improving the understandability; 2. the former substep has grouped these items having similar meaning into clusters, further grouping these clusters coming from similar source reports is helpful to eliminate the redundancy.

I define the distance between two candidate item clusters t and s as Equation 6.5, in which, each cluster can be either candidate sentence cluster or candidate screen cluster and the $\Phi(t)$ represents the set of test reports that contributed to the candidate item cluster t. Based on the Equation 6.5, candidate item clusters are aggregated into *supplementaries* when the distance between them is smaller than the threshold value θ.

$$D(t,s) = 1 - |\Phi(t) \cap \Phi(s)| \,/\, |\Phi(t) \cup \Phi(s)| \tag{6.5}$$

Further, I identify the most representative candidate items in each supplementary cluster. Based on our definition of sentence similarity and screenshot similarity, I abstract all sen-

tences and screenshots within a supplementary into a weighted graph respectively and employ the similarity value as the weight of edges. Given these two weighted graphs, I apply the PageRank algorithm and obtain the PageRank score for each of the node, *i.e.,* sentences and screenshots. These weights will be used within the next phase of content extraction to highlight the most relevant and representative information for each supplementary.

Example: Fig. 6.2-d displays the refinement result of all *candidate items*: three *candidate sentences* (*i.e.,* $t_{0,2}$, $t_{2,1}$ and $t_{4,2}$) are grouped together because they contain "100044 error code"; *candidate clusters* $\{t_{5,0}\}$, $\{t_{5,2}\}$ and $\{s_{5,0}\}$ are grouped because they belong to *report-5*. Particularly, the size of *supplementary-0* is 3 as its content comes from three reports.

Difference Grouping. After detecting all differences, **CTRAS** again applies Hierarchical Agglomerative Clustering to group them into clusters, which are called *text supplementaries* or *image supplementaries* according to their content type. Moreover, for each supplementary, **CTRAS** adopts PageRank algorithm to sort its sentences/screenshots decreasingly.

Furthermore, for each supplementary *sup*, I set its attribute $sup.sources = \{r | \exists t_i \in r, t_i \in sup\}$ and $sup.size = |sup.sources|$. There are several advantages of maintaining these attributes, firstly, developer can track the relationship between supplementary and report through *sup.sources* which is helpful to understand where the supplementary topic comes from and how does it evolve. Moreover, the size of *supplementary* indicates the frequency that the topic is mentioned in reports, thus it can be treated as supplementary prioritization criteria.

Taking all different sentences/screenshots as input, **CTRAS** adopts hierarchical clustering technique (cut point threshold is set γ^T/γ^S) to cluster sentences/screenshots into groups. In addition, there exists a lot of redundant information in the text/screenshot groups, so **CTRAS** applies weighted pagerank algorithm to sort the sentences/screenshots in each

group and extract few of them. I define Text/Screenshot Supplementary here as the extracted sentences/screenshots and the size of Text/Screenshot Supplementary is the number of reports that contributes to its content.

Algorithm 4: COMBINATION ALGORITHM

```
function SupplementaryCombining(𝒯, 𝒮, 𝒟, θ):
    𝒞 ← ∅;
    while 𝒯 ≠ ∅ and 𝒮 ≠ ∅ do
        // find nearest pair between text and image supplementaries
        (t, s) ← argmin_{t∈𝒯, s∈𝒮} 𝒟[t, s];
        if 𝒟[t, s] > θ then // combination stops
            break;
        append (t, s, |t.sources ∪ s.sources|) to 𝒞;
        𝒯 ← 𝒯 \ {t};
        𝒮 ← 𝒮 \ {s};
    return {𝒯, 𝒮, 𝒞};
```

Content Extractor. Based on *master report* and *supplementaries*, **CTRAS** can further refine them and generate a concise final summary.

In many textual summarization techniques (*e.g.,* [104, 105, 75, 57]), the compression ratio K controls the conciseness of the final summary. In previous works, compression ratio is computed as the ratio of the number of selected keywords to the number of total keyword within the original document. However, because **CTRAS** aims at generating summary over both text and screenshots, I extend the classic definition. For the text, I define the compression ratio as the ratio of the number of unique selected word to the total number of unique word within the supplementary. Similarly, for the screenshot, the compression ratio is the ratio of the number of selected screenshots to the total number of screenshots within the supplementary. I weight text and screenshot equally and thus utilize the mean value of these two compression ratios as the compression ratio for the whole summary.

To generate the final summary, I first include the master report, and then list all *supplementaries* sorted by the number of test reports that contributed to them in descending order.

For each *supplementary*, I iteratively select the sentence or screenshot based on the weights (computed in Step 3 of the candidate refinement phase) and include them into the summary, until reaching the summary compression ratio set by the user.

Example: The detail summarizing process is shown in Fig. 6.2-e. I take the *supplementary-0* as sample. It contains 28 keywords and 0 screenshots. At the beginning of summarization, the sentence $T_{0,2}$ is selected as it has the highest PageRank score, then the summary contains 9 keywords and the compression ratio has reached the limit (*i.e.,* the compression ratio is 9/28 > 0.25), the summarization process ends.

Taking the *master report* and all supplementaries as input, **CTRAS**'s content extractor generate the final summarized report through Algorithm 5. Given *aggregate report* RA and its summarization ratio threshold K, **CTRAS** first detects *master report* r_m and supplementaries SU from RA, then it utilizes Algorithm 5 to generate summarized report RS. In the beginning, RS is empty, **CTRAS** first appends *master report* r_m to summary and calculate the current summarization ratio k, then **CTRAS** appends supplementary to RS until k reaches K or there is no more supplementary.

6.3 Implementation

6.3.1 Architect and Design

I present the architecture of **CTRAS** in Figure 6.3. In the implementation, **CTRAS** is composed of four main components: aggregator, summarizer, analyzer, and visualizer. **CTRAS** aims at assisting software developers in four software-maintenance tasks: bug triaging, test-report comprehension, duplicate bug-report aggregation, and expert recommendation. First, the aggregator computes the distance between each pair of test reports based on both the tex-

Algorithm 5: SUMMARIZATION ALGORITHM

```
function CalCurrentRatio(𝒮, 𝒢):
    k_T ← |Φ(𝒮)|/|Φ(𝒢)|; k_S ← |Φ*(𝒮)|/|Φ*(𝒢)|;
    k ← (k_T + k_S)/2;
    return k;

function Summarization(𝒢, K, r*, 𝒮𝒰):
    𝒮 ← ∅;
    append r* to 𝒮;
    k ← CalCurrentRatio(𝒮, 𝒢);
    while k < K and SU ≠ ∅ do
        su ← GetTopItem(SU, K);
        append su to 𝒮;
        k ← CalCurrentRatio(𝒮, 𝒢);
        SU ← SU \ {su};
    return 𝒮;
```

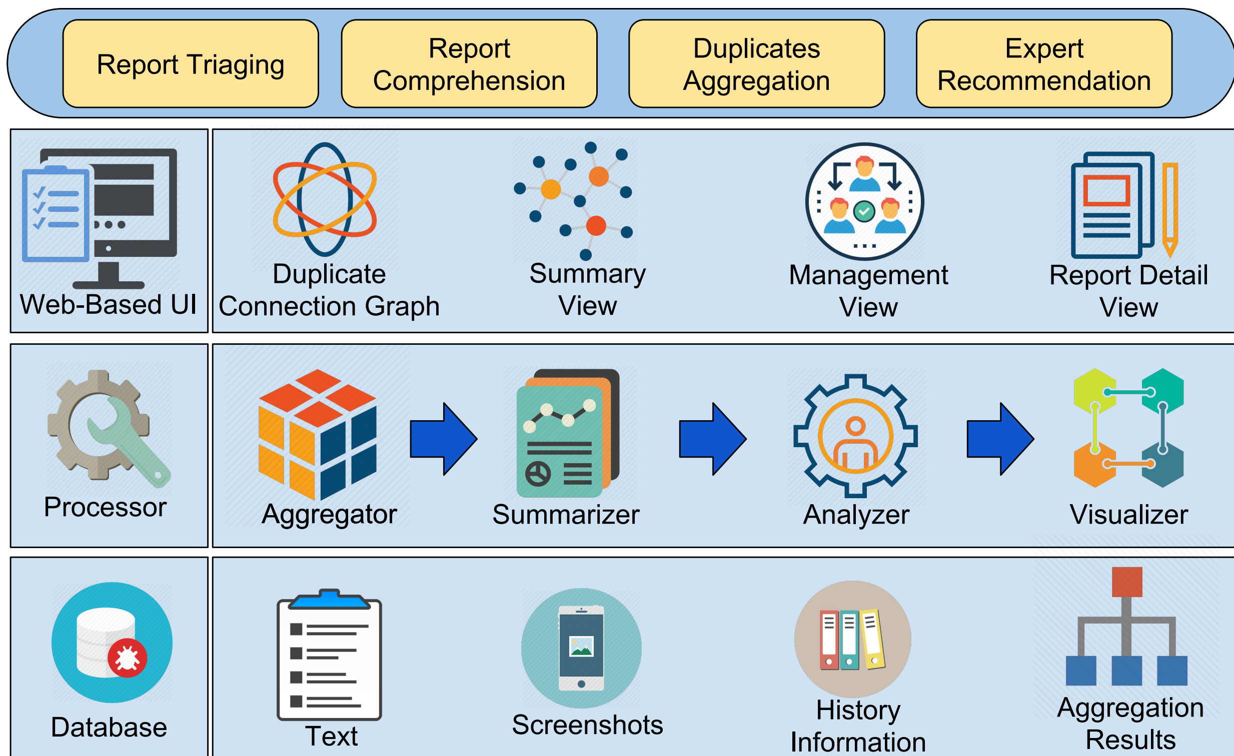


Figure 6.3: Components and features of **CTRAS**

tual description as well as the screenshots to form a distance matrix. Based on the distance matrix, the aggregator is capable of detecting duplicates and grouping them into clusters. And then, the summarizer identifies the most informative test report, *i.e., master report*, and distinct topics for each duplicate group. Summarizer refines these distinct topics and identifies the most representative items to form supplementaries. These supplementaries, along with the master report, comprise the *summary* that summarizes the cluster of duplicates to help developers comprehend the failures and localize bugs. Based on the *summary*, the analyzer mines employ the historical information to predict the severity level, identify the domain expert, and recommend the expert for fixing the bug. Further, to show the aggregate result and help the developers track the summarization procedure, the *visualizer* also shows *aggregation graphs* and *duplicate-relationship graphs*.

Duplicate Detector. Detecting duplicates from the overwhelming number of crowdsourced test reports is the first step of our data processing. **CTRAS** measures the similarity in both of the textual descriptions and screenshots of test reports. **CTRAS** employs NLP techniques to process the textual descriptions, including parts of speech, stop-word removal, and similarity computation. To measure the distance of screenshots, **CTRAS** uses SPM (Spatial Pyramid Matching) to extract the SIFT (Scale-Invariant Feature Transform) features and computes the Chi-Square distance. Based on the similarity of the textual descriptions and screenshots, **CTRAS** detects duplicates and groups the test reports into test-report clusters.

Aggregator. Aggregator component has two fundamental functions: (1) identifying the *master report*, and (2) extract supplementary topics from the duplicates. Based on the distance matrix generated in the duplicate detector, the aggregator employs the PageRank algorithm to measure the importance of reports within each cluster and further identify the master report. After **CTRAS** identifies the master report, it compares each sentence of other reports with the master report and marks the different ones. Then, **CTRAS** ranks the marked sentences based on the PageRank algorithm and conducts hierarchical clustering

on the set to identify topics. Because I consider all topics are helpful for reproducing and understanding the bug, and thus should be included as supplementary descriptions, **CTRAS** further groups the duplicate reports based on the sentence-clustering result and highlights crucial sentences for each report. Users can check the aggregation result and choose to confirm or modify it. To help users make this decision, the aggregator provides the *duplicate-relationship graph* for each of the test reports. Finally, for test reports that are not aggregated into any aggregate reports (*i.e.,* their descriptions and screenshots are not similar to any other test reports), **CTRAS** considers them as revealing a distinct bug. Thus, **CTRAS** allows users to manually process and aggregate such test reports.

Analyzer. **CTRAS** employs the weighted keywords of each of the test-report clusters to mine the historical assignment information in the database. Based on this historical information, **CTRAS** predict the severity of each test report and matches the best fixer for each group of test reports. Compared with similar conventional tools, with which users need to assign reports manually, users only need to check the correctness of the recommendation and choose to confirm or modify it. Note that for some new reports, there may be no domain expert; under this situation, **CTRAS** cannot automatically identify the proper fixer and as such leaves the recommendation as "None".

Expert Recommender. After the duplicate detector completes its clustering, **CTRAS** use the weighted keywords of each of the test-report clusters to mine the historical assignment information in the version-history database. Based on this information, **CTRAS** finds the best fitting domain expert for each cluster, and these recommendations are presented to the user. Compared with similar conventional tools, with which users need to manually assign reports, **CTRAS** only requires users to check the recommendations and choose to confirm or modify them. Note that for some new reports, there may be no domain expert; in this situation, the recommender cannot automatically identify the proper fixer and as such leaves the recommendation as "None".

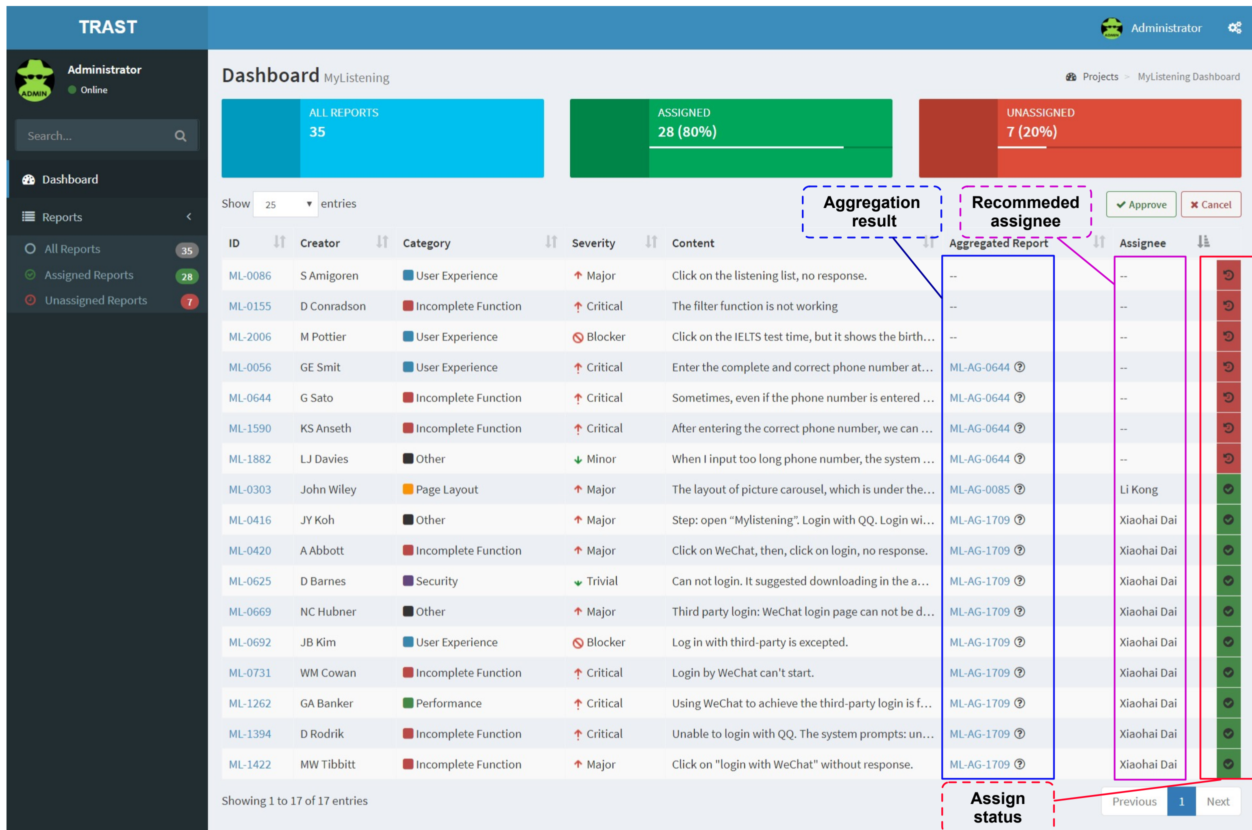
Visualizer. The results of the duplicate detector, aggregator, and expert recommender are presented in a web-based UI, including several visualizations. A tag cloud is presented to depict and summarize the main topics within an aggregate report. The duplicate-relationship graph is visualized to show the relationships of an individual test report with the similar test reports and the aggregate report. An aggregation graph is visualized to show the relationships among all of aggregated test reports and their topics within a cluster. The interface is further described, as well as their usage, are further described in the following section.

To assist users to comprehend the reports, I present a set of web-based UI view and several visualization results in **CTRAS**. For each of the test report clusters, **CTRAS** generate a weighted keyword cloud from all textual description to present a high-level abstraction for describing the bug. Because **CTRAS** split the text description and only keep the noun and verb, with the keywords cloud, users can immediately catch fundamental operation and object that are related with the bug. Besides, to assist developer to comprehend the test reports, **CTRAS** also provides the **aggregation graph** in the aggregate report view, which is helpful for users to understand the relationship between supplementaries.

6.3.2 Interface and Usage

Figures 6.4 and 6.5 show two main views of **CTRAS**. After a user, such as a project manager, logs into **CTRAS**, she can choose the application to work on. Once an application has been chosen, Figure 6.4 is presented. And at the top of Figure 6.4, **CTRAS** shows the total number of submitted test reports, along with the number of assigned and unassigned test reports. Below that is the list of individual test reports. Each test report is presented in a row, along with user-submitted information, such as the test-report identifier, authors, category, severity, and a snippet of the textual description. Additionally, automatically

assigned information is further presented in each row, such as its membership to an aggregate report and the developer to which it was assigned.



ID	Creator	Category	Severity	Content	Aggregated Report	Assignee
ML-0086	S Amigoren	User Experience	Major	Click on the listening list, no response.	--	--
ML-0155	D Conradson	Incomplete Function	Critical	The filter function is not working	--	--
ML-2006	M Pottier	User Experience	Blocker	Click on the IELTS test time, but it shows the birth...	--	--
ML-0056	GE Smit	User Experience	Critical	Enter the complete and correct phone number at...	ML-AG-0644	--
ML-0644	G Sato	Incomplete Function	Critical	Sometimes, even if the phone number is entered ...	ML-AG-0644	--
ML-1590	KS Anseth	Incomplete Function	Critical	After entering the correct phone number, we can ...	ML-AG-0644	--
ML-1882	LJ Davies	Other	Minor	When I input too long phone number, the system ...	ML-AG-0644	--
ML-0303	John Wiley	Page Layout	Major	The layout of picture carousel, which is under the...	ML-AG-0085	Li Kong
ML-0416	JY Koh	Other	Major	Step: open "Mylistening". Login with QQ. Login wi...	ML-AG-1709	Xiaohai Dai
ML-0420	A Abbott	Incomplete Function	Major	Click on WeChat, then, click on login, no response.	ML-AG-1709	Xiaohai Dai
ML-0625	D Barnes	Security	Trivial	Can not login. It suggested downloading in the a...	ML-AG-1709	Xiaohai Dai
ML-0669	NC Hubner	Other	Major	Third party login: WeChat login page can not be d...	ML-AG-1709	Xiaohai Dai
ML-0692	JB Kim	User Experience	Blocker	Log in with third-party is excepted.	ML-AG-1709	Xiaohai Dai
ML-0731	WM Cowan	Incomplete Function	Critical	Login by WeChat can't start.	ML-AG-1709	Xiaohai Dai
ML-1262	GA Banker	Performance	Critical	Using WeChat to achieve the third-party login is f...	ML-AG-1709	Xiaohai Dai
ML-1394	D Rodrik	Incomplete Function	Critical	Unable to login with QQ. The system prompts: un...	ML-AG-1709	Xiaohai Dai
ML-1422	MW Tibbitt	Incomplete Function	Major	Click on "login with WeChat" without response.	ML-AG-1709	Xiaohai Dai

Figure 6.4: The test-report-list view of **CTRAS**

Note that **CTRAS** can aggregate and summarize test reports. The red button (*i.e.*, the *Auto Assign*) control the automatic test report aggregation and assignment process. Upon completion of such processing, the aggregation result and assignee are displayed for each test report, and the user can either approve or change the aggregation and assignment. Also, the user can then view individual test reports by clicking a test-report ID, or view aggregated reports by clicking the Aggregator ID.

I present the view of an exemplary *summary* of the duplicate group "ML-AG-1709" in Figure 6.5. In the top of the center pane, **CTRAS** presents the basic information about the group, including names of all crowd testers who submitted test reports, the assignee of this report group, tags and so on. Additionally, the view shows the detailed information,

Figure 6.5: The summary view of **CTRAS**

including both the textual description as well as the screenshots, of the *master report* which is identified as the most informative one of this group.

Below this pane, supplementaries are shown. Supplementaries are extracted from multiple test reports within the aggregate report. These supplementaries can enrich context and descriptions of the *master report*, which may help users to diagnose and fix bugs. Also, to help users catch the key points of these supplementaries, **CTRAS** highlights the most representative sentence or screenshot within each of them. Below the tag cloud, individual test reports can be found and inspected.

Finally, at the bottom of the view is a graph that depicts the relationships between test reports and their supplementaries. The orange circle denotes the summary, and yellow circles represent individual reports, among which the largest one overlaps the orange circle is the master report. The blue circles surrounding the orange circle represent the supplementaries. The yellow circles radiating from the blue one represent the individual test report that shares the information of the supplementary. The length of the edge denotes the distance between nodes. Note that users can click on any of these nodes and view their details in the right pane. Given this visualization, users can reach a high-level understanding of the aggregation as well as the information of this group.

6.4 Experiment

6.4.1 Research Questions

To assess the performance of **CTRAS** in achieving its goals, *i.e.,* to assist developers in (1) processing test reports, (2) providing comprehensive and comprehensible summaries, and (3) saving efforts, I conduct mixed evaluations to answer the following three research questions:

RQ1. Effectiveness of Duplicate Aggregator. Can the *aggregator* accurately group duplicates? To what extent do the screenshots improve the accuracy of detecting and aggregating duplicate reports?

RQ2. Effectiveness of Summarizer. To what extent can the *summarizer* refine the informative and non-redundant content from the duplicates?

RQ3. Effectiveness of CTRAS. Can **CTRAS** help developers save time costs in inspecting mobile crowdsourced test reports without loss of accuracy?

Both RQ1 and RQ2 are designed to evaluate the effectiveness of the *Aggregator* and *Summarizer* components of CTRAS. Because identifying and aggregating duplicate reports are foundational steps of correctly leveraging the information from test reports, RQ1 aims at evaluating the accuracy of **CTRAS** in detecting duplicates and revealing the effectiveness of employing the screenshot information to aggregate the duplicates. Also, identifying the critical, complementary, and non-redundant information from the large volume of information plays an important role in helping developers to understand and fix the bugs. Thus, RQ2 aims at evaluating the effectiveness of the *summarizer* regarding the metrics of information theory. Further, although RQ1 and RQ2 present quantitative and theoretical evaluations, understanding the practical performance of **CTRAS** is critical. Thus, we design RQ3, which is a task-based user study, to investigate the efficiency improvement as well as reporting any accuracy loss.

6.4.2 Dataset Description

To produce the dataset for our evaluation, I utilized the results of the national software-testing contest[3], which simulated crowdsourced testing of several popular mobile applications across multiple domains (including games, education, social media, and so on).

[3] `http://www.mooctest.org/cst2016/index_en.html`

Table 6.1: Statistical Information of Testing Applications

	Name	Version	Category	$\|R\|$	$\|S\|$	$\|R_s\|$	$\|D\|$
$p1$	CloudMusic	2.5.1	Music	157	259	62	45
$p2$	Game-2048	3.14	Games	210	219	164	96
$p3$	HW Health	2.0.1	Health	262	327	201	109
$p4$	HJ Normandy	2.12.0	Education	269	436	241	123
$p5$	MyListening	1.6.2	Education	473	418	306	128
$p6$	iShopping	1.3.0	Shopping	290	508	150	83
$p7$	JayMe	2.1.2	Social	1400	1997	1168	678
$p8$	JustForFun	1.8.5	Photo	267	112	76	141
$p9$	Kimiss	2.7.1	Beauty	79	58	48	31
$p10$	Slife	2.0.3	Health	1346	2238	1124	885
$p11$	Tuniu	9.0.0	Travel	531	640	418	236
$p12$	Ubook	2.1.0	Books	329	710	88	108
total				5613	7922	4046	2663

The contestants of the contest were required to test the applications and report bugs in four hours. They could write descriptions and take screenshots to document their testing procedures and the unexpected behavior of applications. This contest attracted 4000 participants and generated over 5000 test reports. More than 10 professional testers and members of the organizational committee manually labeled and evaluated the quality of these reports. The detailed information of the dataset is shown in Table 6.1, in which, $|R|$ denotes the number of reports, $|S|$ denotes the number of screenshots, $|R_s|$ denotes the number of reports that contain at least one screenshot and $|D|$ denotes the number of duplicates.

6.4.3 Parameter Settings

As discussed in Section 6.2, several fundamental parameters may influence the performance of **CTRAS**. I provide our parameterizations for all three experiments to assist verifiability and repeatability. Feng *et al.* [33] suggested that the β should be adjusted based on different tasks, and given previous researchers, Bettenburg *et al.* [144], found that the textual information (*e.g.,* description, observed and expected behavior, reproduction steps) is capable of providing more accurate description than screenshots for developers in debugging, I set $\beta = 5$

to weight textual descriptions more. Also, there are two fundamental parameters of the HAC algorithm: the linkage type, which defines the method of calculating the distance between clusters, and the threshold γ for terminating the clustering. I choose the single-linkage type because it makes the HAC to solely focus on the area where the two clusters come closest to each other and ignore distant parts [111], which fits the goal of **CTRAS** well. I set the γ to 0.5 that is the medium value of the scale of the distance between test reports. Further, I apply strict combination strategy by setting $\theta = 0.2$ that I defined in Section 6.2.3 to group candidate item clusters. In the study of RQ1 and RQ3, I set the compression ratio $K = 0.25$, which is considered to be a proper value in the paper[105, 104].

6.4.4 RQ1. Effectiveness of Duplicate Aggregator.

Methods. While a number of classic duplicate test-report-detection methods only focus on the textual description to measure the similarity between reports [109, 119, 118, 2, 69, 120, 91, 46, 27], **CTRAS** employs both textual description as well as screenshots to assist detecting duplicates. Thus, to answer the RQ1, I have the following three methods:

- **CTRAS.** Our duplicate detection method which employs both textual information and screenshots. In this method, the distance between two reports is calculated based on the balanced distance equation.
- **CTRAS-TXT.** The duplicate detection method employs only textual information. In this method, the distance is calculated based on only textual distance.
- **CTRAS-IMG.** The duplicate detection method employs only screenshot information. In this method, the distance is calculated based on only screenshot distance.

Evaluation Metrics. To measure the performance of these three methods, I employ three classic metrics for evaluating clustering: Homogeneity, Completeness, V-Measure [106].

Taking the classes set C and clusters set K as reference, I define the contingency table $\mathcal{A} = \{a_{ij} | i = 1, ..., n; j = 1, ..., m\}$, where a_{ij} denotes the number of test reports that belongs to both c_i and k_j.

Homogeneity reflects the extent to which each cluster contains only members of a single class. It can be calculated via $h = 1 - H(C|K)/H(C)$, where

$$H(C|K) = -\sum_{j=1}^{|K|}\sum_{i=1}^{|C|} \frac{a_{ij}}{N} \cdot log \frac{a_{ij}}{\sum_{k=1}^{|C|} a_{kj}} \tag{6.6}$$

$$H(C) = -\sum_{i=1}^{|C|} \frac{\sum_{j=1}^{|K|} a_{ij}}{n} \cdot log \frac{\sum_{j=1}^{|K|} a_{ij}}{n} \tag{6.7}$$

Completeness is a symmetrical criterion of homogeneity, which measures the extent to which all members of a given class are assigned to the same cluster. It can be calculated via $c = 1 - H(K|C)/H(K)$, where

$$H(K|C) = -\sum_{i=1}^{|C|}\sum_{j=1}^{|K|} \frac{a_{ij}}{N} \cdot log \frac{a_{ij}}{\sum_{k=1}^{|K|} a_{ik}} \tag{6.8}$$

$$H(K) = -\sum_{j=1}^{|K|} \frac{\sum_{i=1}^{|C|} a_{ij}}{n} \cdot log \frac{\sum_{i=1}^{|C|} a_{ij}}{n} \tag{6.9}$$

V-measure is the harmonic mean of homogeneity and completeness. It is widely used as the measure of the distance from a perfect clustering. In our paper, a higher V-Measure score indicates a better duplicate detection and aggregation result, which is calculated by the Equation 6.10.

$$v = 2 \cdot (h \cdot c)/(h + c) \tag{6.10}$$

6.4.5 RQ2. Effectiveness of Summarizer

Methods. To investigate the theoretical effectiveness of the summarizer of **CTRAS**, I compared its performance with two classic summarization methods: Max-Coverage-based (MCB) [34, 121] and Maximal Marginal Relevance (MMR) [18].

- **CTRAS.** Our summarization method that generates the summarized report under ratio K.
- **MCB.** The Max-Coverage-based method is a greedy algorithm. *MCB* iteratively selects the test report with maximal coverage score and inserts it into the summarization until K is met. The original definition of coverage score in paper [34, 121] refers the ratio of the number of selected conceptual units to the total number. In this experiment, I have two kinds of conceptual units, *i.e.,* keywords and screenshots. Thus, I define the coverage score as the mean value of the keyword coverage score and screenshot coverage score.
- **MMR.** The MMR method is a typical method for summarizing multiple topically related documents, which employs keywords that have the highest frequency to build a query [40]. This query is used to select the document from a set based on the maximum-marginal-relevance strategy, which selects the one having the largest distance from the selected set while being relevant to query in each step. In our implementation, I adopt the same idea and construct the query with keyword and screenshot having the highest frequency. I employ the distance between test reports, which I defined in Section 6.2.2, as the distance measurement to implement the maximum-marginal-relevance strategy.

Note that I define the compression ratio of the final summary as the mean value of the text compression ratio and screenshot compression ratio (see Section 6.2.3) — this strategy is also applied in this experiment.

Evaluation Metrics. I adopt a fully automatic evaluation method for content selection: the **Jensen Shannon divergence** (JS divergence), which has been shown to be highly correlated with manual evaluations and sometimes even outperforms standard Recall-Oriented Understudy for Gisting Evaluation (ROUGE) scores [76].

JS divergence employs the probability distribution of words to measure the distance between documents. A good summary is expected to have low divergence with the original document. In this paper, I calculate the JS divergence of textual information and screenshots respectively.

The JS divergence is represented in Equation 6.11, in which, P and Q denote the probability distributions of word $\mathcal{G}$ and summary $\mathcal{S}$, respectively. I entirely adopt the recommended parameter settings in [76], *i.e.,* , $A = (P + Q)/2$ denotes the mean distribution of P and Q, C denotes the frequency of keyword ω, N is the sum of frequencies of all keywords, $B = 1.5|V|$ where V denotes the text corpus, and δ is assigned to 0.0005 to perform a small smoothing. JS_S is defined in a similar manner.

$$JS_T(P||Q) = (D(P||A) + D(Q||A))/2 \tag{6.11}$$

where

$$D(P||Q) = \sum_\omega p_P(\omega) log_2 \frac{p_P(\omega)}{p_Q(\omega)}$$

$$p(\omega) = (C + \delta)/(N + \delta * B)$$

After JS_T and JS_S are calculated, I utilize their harmonic mean as the measure of these summarization methods.

6.4.6 RQ3. Effectiveness of CTRAS

Although RQ2 evaluates the theoretical performance of **CTRAS**, I also seek to understand its practical performance for real users. In [105], Rastkar *et al.* designed a task-based user study to investigate whether the generated summaries can help developers in processing test reports. In their study, participants were asked to read a new test report and a list of potential duplicated reports, which were presented under their *original* or *summary* format, and then determine for each whether it was duplicated with the new test report or not. Considering both Rastkar *et al.*'s work and ours share the same goal, I adopt the task-based user study to answer the RQ3. However, because their work is designed to produce a summary for a single test report while **CTRAS** generates a summary for multiple test reports, I adjust the duplicate test-report-detection task into duplicate test-report clusterization tasks in our study.

For our study, I utilized a modified version of the web-based **CTRAS** tool to assist participants to cluster test reports. Our participants are given a set of test reports (see Table 6.1), and optionally a set of summaries, and asked to group duplicate test reports (*i.e.,* test reports describing the same bugs). Our hypothesis is that **CTRAS** can help developers reach a multi-perspective understanding of the bug and thus identify the duplicates more efficiently without loss of accuracy. The rationale of this study design is straightforward: if the summary generated by **CTRAS** failed to provide *sufficient* and *correct* knowledge for participants to understand the bug, it cannot help participants and further improve their efficiency in grouping the duplicates that are describing the same bug.

Study Setting. I recruited 30 second-year master students majoring in computer science or software engineering as participants of this study. All of these 30 participants have at least 5 years programming experience but have no experience using any of these experimental applications.

I select 5 applications, *i.e., MyListening (p5)*, *iShopping (p6)*, *Kimiss (p9)*, *Tuniu (p11)*, *Ubook (p12)* as subject programs. The categories of these applications are diverse and the number of reports varies from 79 to 531, thus I believe these applications are representative.

I randomly divide the 30 participants into 3 groups. In the study, these three groups are provided with different reference materials:

- **Group A (Control Group):** The participants of this group are only provided with the original test report. There are no supportive materials for the participants of this group.
- **Group B (CTRAS):** The participants of group B are provided with the original test report and the summary that is generated by **CTRAS**. Note that, for this group, the summarizer works on the fully-automated aggregator, which groups the test reports based on both image and text similarity.
- **Group C (Golden):** The participants of group C are provided with the original test report and summaries that are generated by the summarizer of **CTRAS** working on the ground-truth clustering results(as manually determined by the professional developers, described in Section 6.4.2). Because the quality of summaries is influenced not only by the summarization algorithm but also by the duplicate aggregation algorithm, I set up this group to evaluate the gap between the performance of **CTRAS** and the perfect situation.

Within each group (10 participants, each), every subject application (5 software applications) is assigned to two participants. Participants are required to manually cluster these original test reports independently — without any collaboration. The modified version of **CTRAS** shows summaries without showing any information that reveal test report identities — simply showing the summarized sentences and screenshots that describe bugs. This version of **CTRAS** also provides keyword search and keyword filtering.

Table 6.2: Details of the summarization result in RQ3

		p5		p6		p9		p11		p12	
		B	**C**	**B**	**C**	**B**	**C**	**B**	**C**	**B**	**C**
#summary		98	97	35	63	15	25	145	182	64	96
#sentences	**mean**	3.12	4.13	2.33	3.15	2.25	6.00	4.51	4.80	3.85	5.00
	std	1.95	3.77	2.01	1.96	1.85	2.10	3.46	2.63	3.37	7.93
#screenshots	**mean**	1.73	2.35	3.38	4.05	1.25	4.40	3.51	3.69	7.60	9.42
	std	1.84	2.73	5.38	3.11	1.09	3.38	3.80	3.30	10.69	21.92

I employ **CTRAS** to generate summaries under the predefined condition of group B and C, and then provide these summaries to the participant of corresponding groups as reference material. Table 6.2 illustrates summarization results. For each subject application and group (B & C), I show the number of summaries and their mean number of sentences and screenshots.

Evaluation Metrics. I evaluate **CTRAS** based on three aspects: efficiency, accuracy, and satisfaction.

- **Efficiency.** I adopt the average completion time for each report as the evaluation metric of efficiency.
- **Accuracy.** Using the ground truth data described in Section 6.4.2, I determined the accuracy of the participant's inspection by utilizing V-measure metric (see definition in Section 6.4.4).
- **Satisfaction.** The satisfaction of summary is measured upon the qualitative feedback from the questionnaire, which is shown in Table 6.3. Particularly, I present the questionnaire only to participants of group B and C.

Table 6.3: Interview Questions in RQ3

1. On a scale of 1–5 with 5 being the most positive, how would you describe the overall performance of the summary in assisting your clusterization task?
2. Does the master report reflect the topic of the summary; if yes, how does it reflect it?
3. Is there additional information in the supplementaries that helped you cluster test reports? If so, describe it.
4. Which type of information is more important for you in inspecting reports: textual descriptions or screenshots? Why?

6.5 Results and Analysis

In this section, I present the experimental results to answer the three research questions.

6.5.1 Answering RQ1: Effectiveness of the Duplicate Aggregator

I present the homogeneity (H), completeness (C) and V-Measure (V) results of **CTRAS** and the two baseline techniques in Table 6.4.

On average, **CTRAS** achieves 0.87 V-Measure score, while these two baseline techniques, *i.e.,* **CTRAS**-TXT and **CTRAS**-IMG, obtain 0.81 and 0.60 respectively.

Further, **CTRAS** outperforms these two baseline techniques over all subject projects except the *"Slife"(p10).* I investigated the content of test reports of subject projects. I found the application *Slife* is a daily activity tracker, which is designed for tracking the health data of users' daily activity. Even though its operation is simple, the testing procedure, which requires a number of activities beyond the regular operations, becomes relatively difficult. Given the fact that our test reports come from the contest which requires participants to finish the tasks in a short time (4 hours), I speculate that the test reports of *Slife* could only reveal simple bugs, and as such and their text descriptions were accurate. Thus, **CTRAS**-TXT obtains the highest homogeneity score, which results in the relatively higher V-Measure

Table 6.4: Evaluation Results for the Duplicate Aggregator: RQ1

		$p1$	$p2$	$p3$	$p4$	$p5$	$p6$
H	**CTRAS**	**0.967**	**0.991**	**0.990**	**0.991**	**0.957**	**0.948**
	TXT	0.679	0.818	0.918	0.930	0.874	0.857
	IMG	0.444	0.416	0.724	0.789	0.490	0.507
C	**CTRAS**	**0.778**	**0.782**	**0.876**	0.855	**0.925**	**0.904**
	TXT	0.660	0.755	0.872	0.856	0.909	0.903
	IMG	0.748	0.682	0.863	**0.863**	0.883	0.874
V	**CTRAS**	**0.862**	**0.874**	**0.929**	**0.918**	**0.941**	**0.926**
	TXT	0.670	0.785	0.894	0.891	0.891	0.879
	IMG	0.557	0.517	0.788	0.824	0.630	0.642

		$p7$	$p8$	$p9$	$p10$	$p11$	$p12$	avg
H	**CTRAS**	**0.958**	0.865	**0.995**	0.862	**0.994**	**0.967**	**0.957**
	TXT	0.931	**0.878**	0.900	**0.932**	0.932	0.698	0.862
	IMG	0.722	0.222	0.386	0.650	0.703	0.212	0.522
C	**CTRAS**	**0.851**	**0.493**	**0.845**	**0.657**	0.877	**0.774**	**0.801**
	TXT	0.839	0.426	0.834	0.628	**0.880**	0.713	0.773
	IMG	0.821	0.318	0.693	0.602	0.864	0.525	0.728
V	**CTRAS**	**0.901**	**0.628**	**0.914**	0.745	**0.932**	**0.860**	**0.869**
	TXT	0.883	0.574	0.866	**0.750**	0.905	0.705	0.808
	IMG	0.768	0.262	0.496	0.625	0.775	0.303	0.599

score in comparison with **CTRAS**.

Summary: The high V-Measure score indicates that the **duplicate aggregator** ***is capable of accurately detecting and aggregating duplicate reports. In comparison with the classic text-only-based strategies, the screenshot information is able to improve the performance of detecting duplicates (in 11 out of 12 subject applications).***

6.5.2 Answering RQ2: Effectiveness of the Summarizer

The results of RQ2 are shown in Fig. 6.6, for all subject application with varying compression ratios. I note that regardless of the compression ratio, **CTRAS** generally outperforms MCB and MMR methods in all projects except "JustForFun."

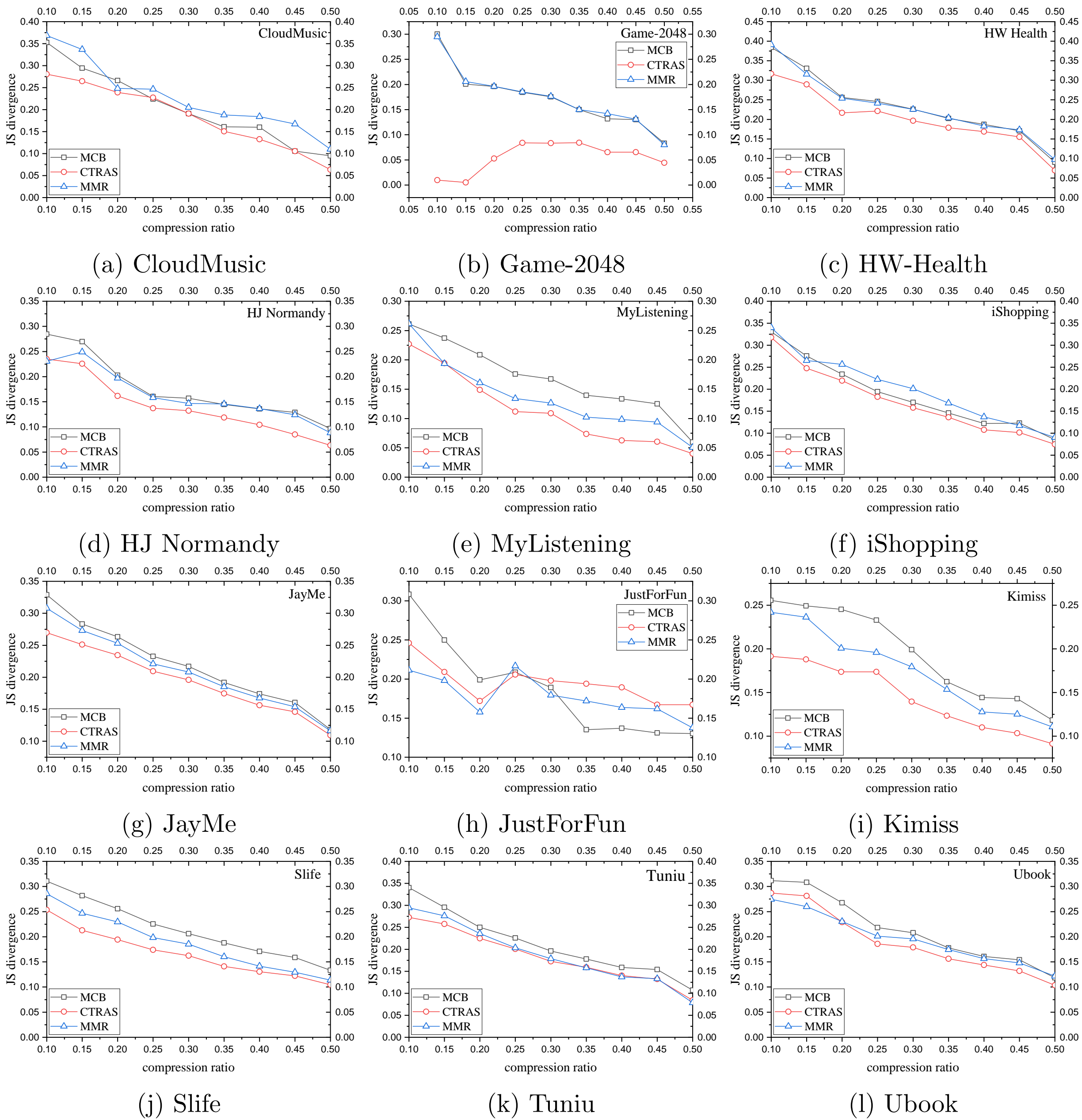
Through further investigation, I found that "JustForFun" is an image editing and sharing application. Thus, the screenshots are largely composed of user content (*i.e.,* their photos) instead of more standardized activity views, so most screenshots have a large distance from each other, which causes them to be categorized as independent supplementaries. This causes a decrease in JS divergence. In addition, as the summarization ratio increases, the score of the JS divergence decreases except for the **CTRAS** result on project "Game-2048," which is caused by the fact that there is only one gaming interface. That is to say the overwhelming majority of screenshots are similar. Few screenshots can represent the whole corpus, thus the JS divergence is smaller under lower summarization ratio.

Summary: *For most projects,* ***CTRAS*** *is more effective than classic summarization methods: MCB and MMR.*

6.5.3 Answering RQ3: Effectiveness of CTRAS

Efficiency & Accuracy. Table 6.5 shows the average test report inspection time cost, per test report, for group A (*i.e.,* control group), B (*i.e.,* **CTRAS**) and C (*i.e.,* golden). According to Table 6.5, the average completion time cost of each report are 19.32 and 20.58 seconds, respectively for group B and C, which saves 30.0% and 25.5% compared with group A (27.63 seconds); and the average V-measure scores of group A, B, and C are 0.9071, 0.9316, and 0.9400 respectively, which shows that group B and group C improve 2.7% and 3.6% accuracy compared with group A. This result indicates that with the help of summarization, people can substantially save their time in duplicate test report clustering work not only without loss of accuracy, but even with slight improvement.

In addition, surprisingly, group B cost less time than group C on average. I investigated the details of the summarization result that is presented in Table 6.2. I find that that **CTRAS** performs a more strict duplicate aggregation than the professional developers, which leads



(a) CloudMusic (b) Game-2048 (c) HW-Health

(d) HJ Normandy (e) MyListening (f) iShopping

(g) JayMe (h) JustForFun (i) Kimiss

(j) Slife (k) Tuniu (l) Ubook

Figure 6.6: Evaluation results for the Summarizer

(lower is better.)

Table 6.5: Task Evaluation Results: RQ3

	A	**B**	**C**
completion time (s)	27.6293	19.3179 (30.0%)	20.5754 (25.5%)
v-measure	0.9071	0.9316 (2.7%)	0.94 (3.6%)

the number of clusters for group B to be smaller than group C. As such, participants of group B generally have fewer summaries to reference in the inspection procedure. Thus, in comparison with group C, group B can save some time-cost at the expense of loss of accuracy.

Satisfaction. The participants' satisfaction rating on average was high: 4.1 on a 1–5-point scale. More subjectively, the semi-structured interviews produced qualitative results. All participants thought that the *master report* can reflect the topic of summary, and it "*helps them get a general idea of the summary*", "*instructs the granularity of clustering*", 18 participants (90%) mentioned supplementaries contained additional information which "*is clear and coherent*", "*can be used as valuable reference when it comes to uncertain condition*" and "*provides detailed operation steps.*"

Many participants mentioned text was more useful, which supports our strategy of setting distance calculation parameters described in Section 6.4. Some participants stated that screenshots "*are open to various interpretations*" and "*can't tell where's the problem.*" Moreover, some suggestions for improvement were proposed, such as "*the description is not well structured,*" and "*highlighting important parts of screenshots.*"

6.6 Threats to Validity

Subject Program Selection. The first threat is related to the generality of **CTRAS**. I evaluated our approach on 12 projects, all of which are Android applications, thus it is unclear whether **CTRAS** can achieve similar results on other projects from Android and

other mobile platforms (*e.g.,* iOS). However, the categories of our projects vary widely, such as Music & Audio, Games, Health & Fitness, Education, Travel & Local, and so on. Therefore, I believe the experiment result can indicate the usefulness of our method.

Natural Language Selection. All the crowdsourced test reports utilized in this paper are written in Chinese, which may affect the generalization of **CTRAS**. However, the purpose of our method is to generate comprehensive summaries by leveraging the information in duplicated reports, and its key part is to measure the similarities between reports utilizing textual description and screenshots. In the aspect of textual descriptions, the similarities are effected by the keyword-corpus extraction methods, and NLP researchers have proposed many relatively mature approaches targeting different languages.

Student Participant. In our user study, 30 students were recruited to complete the task, although the participants are students, none have used the subject applications, which means that diagnosing for these applications is a new task for them. Based on Salman *et al.*'s research [110], under this situation, students and professionals often perform similarly. Thus, I argue that the result can be generalizable to professionals.

6.7 Conclusion

The problem of diagnosing the overwhelming number of test reports has been a fundamental challenge for crowdsourced testing. To alleviate this problem, in this paper I present **CTRAS**, a novel approach for aggregating and summarizing crowdsourced test reports. **CTRAS** leverages the duplicate reports to assist professional testers to manage and understand crowdsourced test reports. It overcomes several shortcomings by: (1) aggregating duplicates to enable batch processing, (2) summarizing the supplementary topics from duplicates to facilitate developer comprehension of the reports. The evaluation result reveals that **CTRAS** is capable of assisting people's detecting and triaging crowdsourced test reports.

Chapter 7

An Empirical Study on Clustering Crowdsourced Test Reports of Mobile Applications Using Image Understanding

7.1 Study Design

To validate our technique, I conduct a comprehensive experiment on the industrial data. In this section, I first raise research questions of this experiment, and then I detail the dataset, settings, baselines, and evaluation metrics. I evaluate our technique through three aspects: effectiveness, usefulness, and potential. In addition, I analyze the parameter sensitivity for helping users to apply our technique in different settings.

7.1.1 Research Questions

Identifying the test reports that describe the same bug or present similar topics is critical for improving the efficiency of processing the overwhelming number of crowdsourced test reports. For our technique, one of the critical steps is to group these test reports into clusters. Given the fact that clustering result fundamentally determines the effectiveness of our technique, I design the $RQ1$ to investigate whether our image-understanding-based clustering technique is effective for grouping the test reports. $RQ1$ is formulated as follows:

[**RQ1. Effectiveness:**] To what extent can cluster with image features accurately group the crowdsourced mobile test reports?

On the other hand, even though I investigate the effectiveness of our technique in clustering the test reports in the $RQ1$, it is essential for validating its usefulness for the practical test report inspection task. To understand its practical usefulness, I compare our technique with the existing state-of-the-art crowdsourced test report processing techniques. In addition, considering investigating the potential of our technique could be helpful for engineers to optimize it in the application and inform the future research in this field, I design the following two research questions:

[**RQ2.1 Usefulness:**] To what extent can our approach substantially improve test-report inspection and find more unique buggy reports earlier?
[**RQ2.2 Potential:**] How large is the gap between our clustering method and IDEAL strategies?

Finally, because the performance of our technique is influenced by several key parameters, I analyze the performance of our approach under different parameter settings to present users an extensive understanding of our technique. The $RQ3$ is designed as follows:

[RQ3. Parameter Sensitivity:] How does the experimental parameter influence the performance of our approach?

7.1.2 Data Collection

To investigate the performance of our technique under real crowdsourced testing settings, my collegues and I collaborated with six industrial companies. Based on the mobile application and testing requirement provided by these companies, I host a national software-testing contest[1] to simulate the crowdsourced testing. In total, more than 100 third-year undergraduates come from 27 prestigious universities participate this contest. The contestants were required to test the subject applications and report bugs within four hours. They could write descriptions andor take screenshots to document their testing procedures and the behavior of applications. More than 10 professional testers and developers from our collaborators manually labeled and evaluated the quality of these reports. The detailed information of the dataset is shown in Table 7.1, in which, $|R|$ denotes the number of reports, $|S|$ denotes the number of screenshots, $|R_s|$ denotes the number of reports that contain at least one screenshot, and $|F|$ denotes the number of faults that revealed by the reports. Based on this figure, I can observe that 1119 out of 1644 crowdsourced test reports of mobile applications contains screenshots, and these reports detected 119 bugs.

7.1.3 Experiment Setup

In this section, I detail the experiment setup and the parameter settings. One of the features of crowdsourced testing is that it can provide the testing results of diverse devices. Thus, given the screenshots submitted by crowd workers are potentially of different resolutions,

[1] http://www.mooctest.org/cst2016/index_en.html

Table 7.1: Summary of Experimental Subjects

	Name	Category	$\|R\|$	$\|S\|$	$\|R_s\|$	$\|F\|$
$p1$	Wonderland	Travel	191	116	93	23
$p2$	Game-2048	Game	210	174	154	12
$p3$	TravelDiary	Travel	240	170	142	14
$p4$	HW Health	Health	262	274	201	33
$p5$	HJ Normandy	Education	269	381	241	22
$p6$	MyListening	Education	432	348	288	15
Total			1644	1463	1119	119

in this experiment, I resize all the screenshots to 480×480 pixels. Further, as discussed in the previous sections, some fundamental parameters can influence the performance of our experiment. I set the threshold of identifying the similar screenshots $\gamma = 0.1$, the factor of scaling the weight of text similarity $\alpha = 0.8$, and the threshold of balanced factor $\beta = 1$. Also, in agglomerative hierarchical clustering, I adopt the set the threshold of determining the stop point of clustering $\varepsilon = 0.8$.

In the sampling procedure, I set the sampling ratio parameter $\rho = 5\%$. This means, I randomly sample 5% reports from clusters in each iteration. Especially, for the clusters that contain only one test report, I take all singular test report in the first round. Note that, except the experiment of $RQ4$ that is designed for investigating the parameter sensitivity, I did not change these setting in the whole experiment to ensure the consistency of the results.

7.1.4 RQ1. Effectiveness

Baselines. Because $RQ1$ is designed for evaluating the quality of clustering results, I adopted *TXT*, which clusters test reports based on only the text distance, and *IMG*, which clusters test reports based on only the image distance to reveal the performance of image information, as baselines. Thus, I have the following three techniques:

- **TXT**: The clustering is conducted based on only the text distance between test reports,

which was presented as DT in equation 5.1.

- **IMG**: The clustering strategy based on the screenshot distance between test reports, which was presented as DS in equation 5.3.

- **TXT&IMG**: The clustering is conducted based on both textual information and screenshots. In this method, the distance between the two reports is calculated based on the balanced distance equation 5.4.

Evaluation Metrics.

Similar to the Section 7.1.4, I employ widely-used metrics to analyze the result of clustering, including Homogeneity, Completeness, and V-measure [106].

To ease the explanation, I use the class set C and cluster set K to denote ground truth and clustering results. Then I define n as the total number of reports, n_c and n_k as the number of reports respectively belonging to class c and cluster k, and $n_{c,k}$ denotes the number of reports from class c assigned to cluster k. Based on these annotations, I can formulate these metrics.

Homogeneity: A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class. It reflects the extent to which each cluster contains only members of a single class. Homogeneity scores are formally given by:

$$h = 1 - \frac{H\left(C|K\right)}{H\left(C\right)}$$

where

$$H(C|K) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|} \frac{n_{c,k}}{n} \cdot \log\left(\frac{n_{c,k}}{n_k}\right)$$
$$H(C) = -\sum_{c=1}^{|C|} \frac{n_c}{n} \cdot \log\left(\frac{n_c}{n}\right)$$

Completeness: A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster. It measures the extent to which all members of a given class are assigned to the same cluster. Completeness scores are formally given by:

$$c = 1 - \frac{H(K|C)}{H(K)}$$

where

$$H(K|C) = -\sum_{k=1}^{|K|}\sum_{c=1}^{|C|} \frac{n_{c,k}}{n} \cdot \log\left(\frac{n_{c,k}}{n_c}\right)$$
$$H(K) = -\sum_{k=1}^{|K|} \frac{n_k}{n} \cdot \log\left(\frac{n_k}{n}\right)$$

V-measure: The V-measure is a harmonic mean between homogeneity and completeness, which is widely used as a metric to evaluate the performance of clustering. In our experiment, a higher V-measure score means better clustering performance. The V-measure score is calculated as the following formula:

$$v = 2 \cdot \frac{(h \cdot c)}{h + c} \tag{7.1}$$

The value range of all these above metrics lay into $[0, 1]$, and they are the higher the better.

7.1.5 RQ2. Usefulness

Baselines. Besides the three techniques I employed for addressing the $RQ1$, I simulate the ideal inspection process, which can only be achieved theoretically, to investigate the usefulness and potential of my technique. Further, I employ the method, which introduced in Chapter 5 to prioritize test reports based on the distance of both text description and images, as one of the baselines. Also, I simulate the situation that developers have no any ancillary techniques for inspecting the reports. Under that situation, they may randomly inspect test reports, *i.e.,* in a non-systematic order. Thus, for $RQ2$, I have six techniques as follows:

- **Text**: The sampled report clusters are derived from the results of only text-distance-based clustering.
- **Image**: The sampled report clusters are derived from the results of only screenshot-distance-based clustering.
- **BDDiv**: A multi-objective test-report prioritization technique proposed in Section 5.2 to assist the inspection of crowdsourced test reports.
- **Text&Image**: The sampled report clusters are derived from the results of hybrid-distance-based clustering.
- **RANDOM**: The randomly inspection strategy, which is used to simulate the situation without ancillary techniques.
- **IDEAL**: The theoretically ideal inspection orders.

Evaluation Metrics. I adopted the APFD (Average Percentage of Fault Detected) [116], a widely-used evaluation metric of the classical test case prioritization, to measure the performance of our technique. For each fault, APFD marks the index of first test report which

reveals it. Based on the order of test reports and fault information they revealed, I can calculate APFD scores to evaluate the usefulness of our technique for report inspection. I present the formula of computing the APFD in the following equation:

$$APFD = 1 - \frac{T_{f1} + T_{f2} + \cdots + T_{fM}}{n \times M} + \frac{1}{2 \times n} \quad (7.2)$$

In Equation 7.2, n denotes the number of test reports, M denotes the number of faults revealed by all test reports. T_{fi} is the index of the first test report that reveals fault i. In our experiment, a higher APFD value indicates a better inspection procedure. That is, the method with higher APFD value can reveal more faults earlier. In addition, I employ **Gap**, which reflects the difference that our technique compared with the IDEAL strategy, to evaluate the potential of our technique. In the experiment, the **Gap** between technique X and the **IDEAL** can be calculated as $G = (Best - X)/X$. This metric indicates the potential of techniques, which is helpful for researchers to design and improve crowdsourced test report processing techniques.

7.1.6 RQ3. Parameter Sensitivity

In $RQ3$, I analyze the impact of parameters on the performance. I focus on the following parameters: balanced factor β, which controls the weight of text distance and image distance; clustering threshold ε, which determines the stop point of hierarchical clustering; and sampling ratio ρ, which determines the number of test reports to be sampled from each cluster.

The parameter β and ε are designed for controlling the process of clustering, thus, I employ the evaluation metrics of $RQ1$, *i.e.,* homogeneity, completeness, and v-measures, to analyze their impacts. Similarly, because of the sampling ratio ρ influences the practical performance

of our technique, I employ the evaluation metrics of $RQ2$, *i.e.,* APFD, to analyze its impacts.

7.2 Results and Discussion

In this section, I present the experimental results to answer the three research questions.

7.2.1 Answering Research Question 1

[RQ1. Effectiveness:] To what extent can cluster with image features accurately group the crowdsourced mobile test reports?

I present the homogeneity (H), completeness (C) and V-Measure (V) results of these three techniques in Table 7.2. Regarding the V-Measure score, TXT&IMG, *i.e.,* clustering based on balanced-distance, achieves 0.643 on average, while the two baselines, *i.e.,* TXT and IMG, obtain only 0.592 and 0.622 respectively. I can observe that TXT&IMG outperforms the two baselines over five subject projects, except the $p2(Game - 2048)$.

I investigate the $p2(Game - 2048)$ to identify the reason for this distinct result. I notice that while the TXT&IMG obtains a relatively high homogeneity score of more than 0.8 and outperforms the two baselines on $p1, p3, p4, p5, p6$, TXT reaches 0.857 when TXT&IMG achieves only 0.514 on $p2(Game - 2048)$. When the completeness scores of these three techniques are close to each other, the high homogeneity score of TXT naturally leads a high V-measure score. I further analyze the raw reports of $p2(Game - 2048)$. I found that within the test reports of $p2(Game - 2048)$ almost all screenshots submitted by crowd workers are the activity view of the game content panel. Even though these screenshots are used to describe different bugs, they are often similar to each other because all of them are captured from the same activity view. This effect misleads the two image-involved techniques, *i.e.,*

TXT&IMG and IMG, to group the test report that are describing different bugs into the same clusters.

Summary: The high V-measure scores indicate that our image-understanding-based test report clustering technique is capable of improving the test reports describing similar bugs together. On the other hand, I found that the information of screenshots may negatively influence the test report clustering of these applications that contain limited number of activity views.

Table 7.2: Experiment Reuslts of Test Report Clustering with Different Distance Metrics

	Method	*p*1	*p*2	*p*3	*p*4	*p*5	*p*6	*avg*
H	TXT&IMG	**0.900**	0.514	**0.890**	**0.914**	**0.807**	**0.800**	**0.804**
	TXT	0.751	**0.857**	0.746	0.879	0.800	0.669	0.784
	IMG	0.444	0.456	0.884	0.859	0.789	0.791	0.703
C	TXT&IMG	**0.694**	**0.44**	**0.537**	**0.643**	**0.546**	**0.390**	**0.542**
	TXT	0.660	0.386	0.513	0.592	0.444	0.316	0.485
	IMG	0.683	0.393	0.526	0.633	0.533	0.386	0.527
V	TXT&IMG	**0.783**	0.474	**0.670**	**0.755**	**0.651**	**0.525**	**0.643**
	TXT	0.702	**0.532**	0.608	0.707	0.572	0.429	0.592
	IMG	0.778	0.422	0.659	0.728	0.626	0.519	0.622

7.2.2 Answering Research Question 2

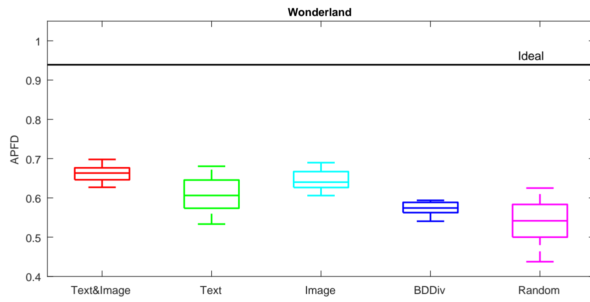
To reduce the bias that is introduced by the randomness in the iterative sampling process, I conducted the experiment 30 times and present the result in Figure 7.1 and Figure 7.2.

Figure 7.1 shows boxplots of the APFD results for the six projects and Figure 7.2 presents the average fault detection curves with the increasing number of inspected reports. In addition, I present the mean value of APFD of the 30 runs, the improvement over *RANDOM*, and the gap between our technique and *IDEAL*. Further, I conduct one-way ANOVA tests over these six techniques and present the result in Table 7.3.

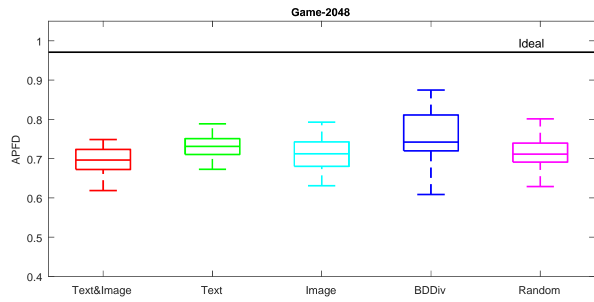
RQ2.1: To what extent can our approach substantially improve test-report inspection and find more unique buggy reports earlier?

Note that I implemented Feng *et al.*'s prioritization technique [33], which is denoted as *BDDiv*, and employed it as a baseline. Based on the boxplots of APFD values shown in Figure 7.1 and the third column of Table 7.3, I observe that, to different extents, all of these clustering techniques outperform the RANDOM inspection on all projects except $p2(Game-2048)$. Similarly, the curves in Figure 7.2 show that the *TXT&IMG* is able to detect all faults earlier on these projects. Especially, for the *BDDiv* method, which also employed the information of screenshots to analyze the test reports, I observed that the *TXT&IMG* technique consistently outperforms it on all projects except $p2(Game - 2048)$.

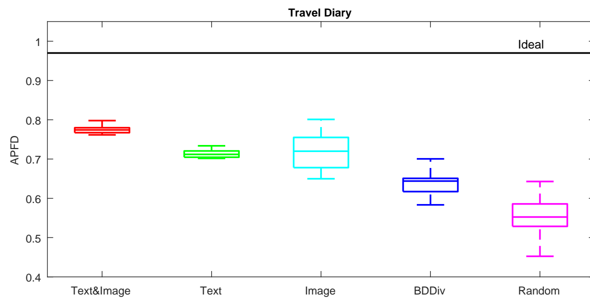
Further, considering I adopted the ramdom strategy to sample test reports from the clustering results, I repeat the experiment 30 times and conduct the one-way ANOVA tests based on the APFD scores. I present the one-way ANOVA tests results and the average improvement over *RANDOM* in Table 7.3. Based on Table 7.3, for all projects except $p2(Game - 2048)$, I can observe that the improvement of *TXT&IMG* ranges 21.3%-37.17% in comparison with the *RANDOM*, while *TXT* improves only 0%-26.02%. Given the fact that all $F - values$ are relatively larger and the $p - values \leq 0.001$, I can safely conclude that the improvements coming from these clustering techniques are statistically significant. Also, on all projects, I observe that the lengths of the boxplots of our clustering-sampling techniques, *i.e., TXT&IMG*, *TXT*, and *IMG*, are smaller than the boxplot of RANDOM. Because the box length indicates the data variability, this observation indicates the performance of these four techniques is more stable than *RANDOM*.



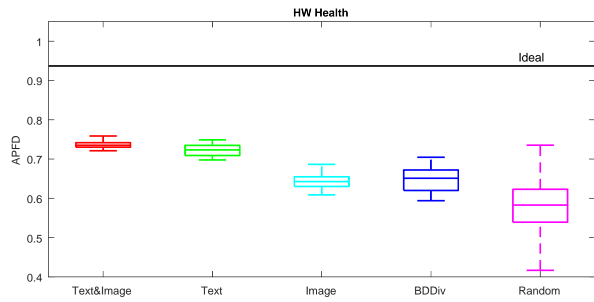
(a) APFD on Wonderland



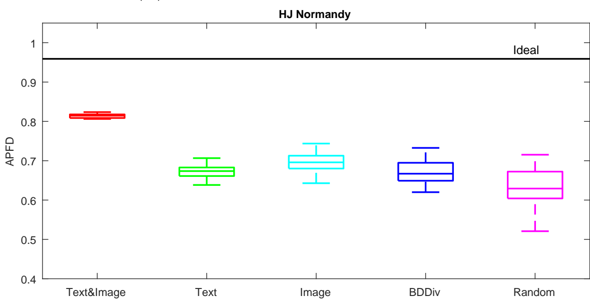
(b) APFD on Game-2048



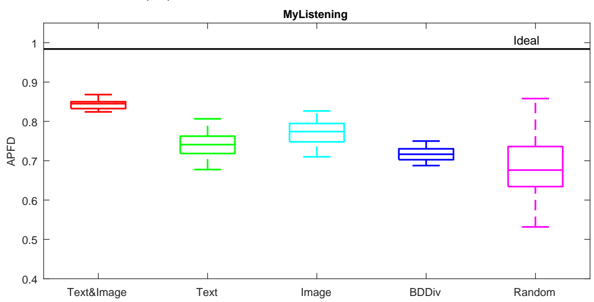
(c) APFD on Travel Diary



(d) APFD on HW Health



(e) APFD on HJ Normandy



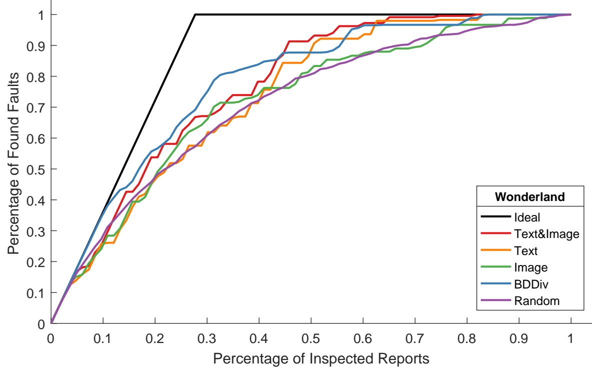
(f) APFD on MyListening

Figure 7.1: APFD of Experimental Subjects (averaged over 30 runs)

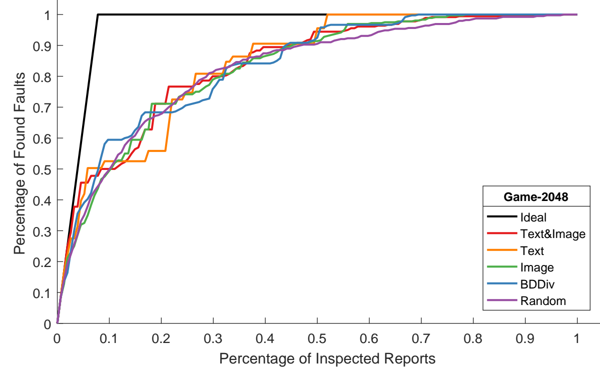
RQ2.2: How large is the gap between our clustering method and IDEAL strategies?

The fourth column of Table 7.3 shows the gap between our strategies and the theoretical IDEAL. Over the six subject programs, I found the gap between *TXT&IMG* and *IDEAL* varies from 17.56% to 41.42% while the gap between *BDDIV* and *IDEAL* ranges from 29.12% to 64.16%. In Figure 7.2, which visualizes the growth rate of APFD value, the curves of *IDEAL* grow at a fast rate, and the best situation reached the top while the *TXT&IMG* stayed around 35%.

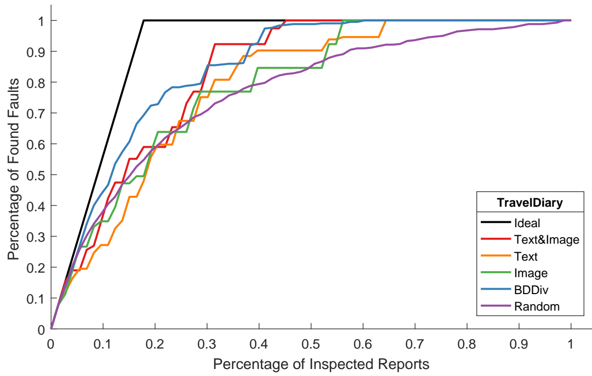
Summary: To answer the RQ2, I conducted one-way ANOVA tests over the results of 30 executions. Based on the test result, I can draw the following summaries: 1. all of these clustering techniques can improve the efficiency of the test reports inspection in comparison with the RANDOM method. 2. the image-based approaches distinctly improved the performance of conventional text-based clustering techniques. 3. on the projects with ample app-specific views, clustering techniques are more appropriate for report inspection than the prioritization techniques. Compared with other strategies, the TXT&IMG shows a smaller gap for the theoretical IDEAL result. However, there is room for future work to improve the clustering-sampling techniques for test report inspection.



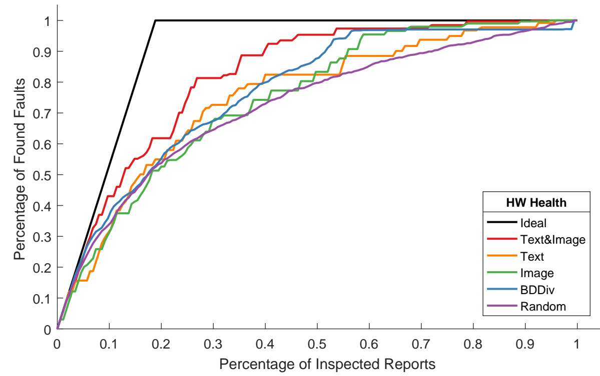
(a) AFDR on Wonderland



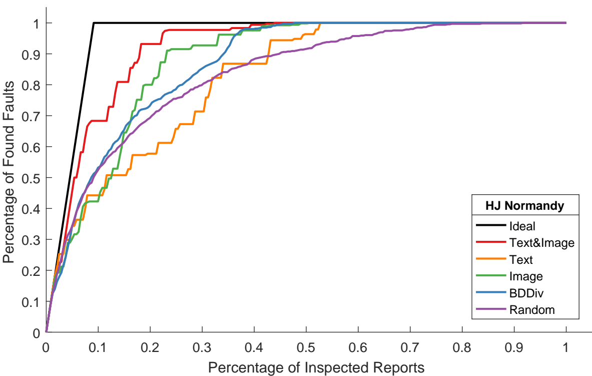
(b) AFDR on Game-2048



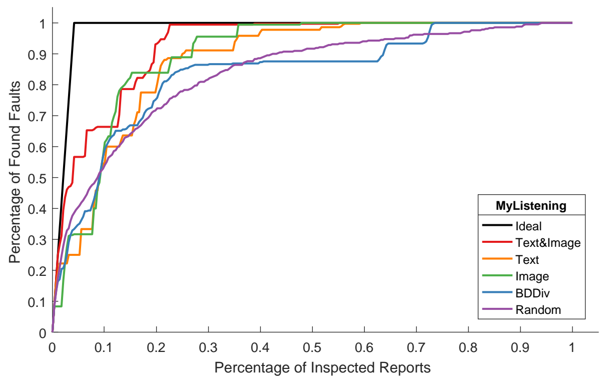
(c) AFDR on Travel Diary



(d) AFDR on HW Health



(e) AFDR on HJ Normandy



(f) AFDR on MyListening

Figure 7.2: Average Fault Detection Rates (AFDR) on Experimental Subjects (averaged over 30 runs)

Table 7.3: One-way ANOVA Tests

Method	**APFD Means**	**Improvement:** $\frac{X-Random}{Random}$	**Gap:** $\frac{Best-X}{X}$
Wonderland: $F(4,159) = 82.659, p-value \leq 0.001$			
IDEAL	0.939	73.89%	/
TXT&IMG	0.664	22.96%	41.42%
TXT	0.607	12.41%	54.7%
IMG	0.643	19.07%	46.03%
BDDIV	0.572	5.93%	64.16%
RANDOM	0.54	/	73.89%
Game-2048: $F(4,159) = 7.223, p-value \leq 0.001$			
IDEAL	0.971	36.95%	/
TXT&IMG	0.695	-1.97%	39.71%
TXT	0.709	0%	36.95%
IMG	0.708	-0.14%	37.15%
BDDIV	0.752	6.06%	29.12%
RANDOM	0.709	/	36.95%
TravelDiary: $F(4,159) = 131.442, p-value \leq 0.001$			
IDEAL	0.97	71.68%	/
TXT&IMG	0.775	37.17%	25.16%
TXT	0.712	26.02%	36.24%
IMG	0.722	27.79%	34.35%
BDDIV	0.64	13.27%	51.56%
RANDOM	0.565	/	71.68%
HW Health: $F(4,159) = 89.866, p-value \leq 0.001$			
IDEAL	0.937	60.17%	/
TXT&IMG	0.735	25.64%	27.48%
TXT	0.722	23.42%	29.78%
IMG	0.641	9.57%	46.18%
BDDIV	0.645	10.26%	45.27%
RANDOM	0.585	/	60.17%
HJ Normandy: $F(4,159) = 130.542, p-value \leq 0.001$			
IDEAL	0.959	51.74%	/
TXT&IMG	0.808	27.85%	18.69%
TXT	0.672	6.33%	42.71%
IMG	0.695	9.97%	37.99%
BDDIV	0.669	5.85%	43.35%
RANDOM	0.632	/	51.74%
MyListening: $F(4,159) = 51.934, p-value \leq 0.001$			
IDEAL	0.984	42.61%	/
TXT&IMG	0.837	21.3%	17.56%
TXT	0.74	7.25%	32.97%
IMG	0.77	11.59%	27.79%
BDDIV	0.718	4.06%	37.05%
RANDOM	0.69	/	42.61%

Table 7.4: The comparison of mean clustering results under different settings of β

(over 30 executions, $\varepsilon = 0.8$ and $\rho = 0.1$)

Project	**Metric**	**Balance Factor** β											*avg*	*std*
		0.5	0.6	0.7	0.8	0.9	1	1.1	1.2	1.3	1.4	1.5		
Wonderland	H	0.872	0.877	0.879	0.897	0.887	**0.9**	0.865	0.859	0.823	0.794	0.765	0.856	0.042
	C	0.676	0.678	0.682	0.682	0.68	**0.694**	0.681	0.684	0.683	0.677	0.672	0.681	0.005
	V	0.761	0.764	0.768	0.775	0.769	**0.783**	0.762	0.762	0.746	0.731	0.715	0.758	0.019
Game-2048	H	0.444	0.444	0.441	0.447	0.454	**0.514**	0.491	0.514	0.51	0.504	0.511	0.479	0.031
	C	0.439	0.439	0.429	0.433	0.436	**0.44**	0.43	0.421	0.411	0.409	0.429	0.429	0.01
	V	0.441	0.441	0.435	0.44	0.445	**0.474**	0.458	0.463	0.452	0.452	0.466	0.452	0.012
Travel Diary	H	0.765	0.776	0.817	0.866	0.861	**0.89**	0.861	0.826	0.783	0.742	0.739	0.811	0.051
	C	0.479	0.495	0.524	0.537	0.538	0.537	**0.542**	0.543	0.535	0.529	0.483	0.522	0.023
	V	0.589	0.604	0.638	0.663	0.662	**0.67**	0.665	0.656	0.635	0.617	0.584	0.635	0.03
HW Health	H	0.825	0.877	0.907	0.91	**0.914**	0.908	0.896	0.892	0.891	0.853	0.849	0.884	0.028
	C	0.588	0.602	0.64	0.64	**0.643**	0.641	0.638	0.621	0.618	0.577	0.569	0.616	0.027
	V	0.686	0.713	0.75	0.752	**0.755**	0.752	0.745	0.732	0.729	0.688	0.681	0.726	0.028
HJ Normandy	H	0.664	0.689	0.758	0.799	**0.807**	**0.807**	0.805	0.784	0.759	0.715	0.735	0.757	0.048
	C	0.478	0.503	0.528	0.541	0.541	**0.546**	0.532	0.531	0.526	0.518	0.523	0.524	0.019
	V	0.555	0.581	0.622	0.645	0.645	**0.651**	0.64	0.633	0.621	0.6	0.611	0.619	0.029
MyListening	H	0.764	0.774	0.789	0.794	**0.8**	0.791	0.779	0.783	0.776	0.776	0.751	0.78	0.013
	C	0.375	0.372	0.385	0.386	**0.39**	0.376	0.381	0.385	0.367	0.367	0.372	0.378	0.008
	V	0.503	0.502	0.517	0.519	**0.525**	0.509	0.511	0.516	0.498	0.498	0.497	0.509	0.009

7.2.3 Answering Research Question 3

RQ3: How does the experimental parameter influence the performance of our approach?

In this subsection, I further discuss the impact of parameter settings on the performance. This study is helpful for users of our technique to set proper parameters for different usage scenarios. I analyze the parameter sensitivity tests based on three key parameters: balanced factor β, clustering threshold ε and sampling percent ρ, which influence the three fundamental steps of the clustering-sampling process respectively. The parameter β controls the balance distance calculation as a harmonic weight. I analyze the clustering results when the value of β ranges from 0.5 to 1.5 with the increment of 0.1. The parameter ε is employed to control the stop point of hierarchical clustering. The clustering procedure terminates when the distance between the closest cluster pair is larger than the value of ε. In this study, I discuss the cluster results when the value of ε ranges from 0.5 to 0.9 with the increment of 0.1. Further, because the parameter ρ controls the number of reports sampling from each cluster, it influences the efficiency of test report inspection. I analyze the trends of APFD value when the value of ρ ranges from 5% to 30% with the increment of 5%.

Figure 7.3 shows the sensitivity of clustering results to the parameter β, given the $\varepsilon = 0.8$ and $\rho = 0.1$. And I present the average value of homogeneity, completeness, and v-measure in the same setting in Table 7.4 . From the table I observe that when the value of β reaches 1.0, four projects, *i.e.,* Wonderland, Game-2048, Travel Diary, and HJ Normandy, obtain the highest v-measure score. The other two projects, *i.e.,* HW Health and MyListening, reach the highest v-measure score while $\beta = 0.9$, and the difference of v-measure between $\beta = 1.0$ and $\beta = 0.9$ is less than 0.01. This indicates that setting the weight of the distance of textual description as well as screenshots closely is helpful for optimizing the balanced-distance-based clustering. Also, I present the average value and corresponding standard

derivation for each of these projects in the last two columns of the Table 7.4. I observe that the standard deviation value of homogeneity, completeness, and v-measure stays in a range of small number, *i.e.,* from 0.005 to 0.051, which indicates that the clustering result is relatively stable to the change of β.

Similarly, Table 7.5 shows the average APFD score with the changes of paramter ε, given the $\beta = 1.0$ and $\rho = 0.1$. From the table I observe that APFD of five projects $(p_1, p_3, p_4, p_5, p_6)$ reaches highest values when $\varepsilon = 0.8$. In this table I also present the average value and standard deviation. The results show that, for all the six subject programs, the standard deviation of the APFD values is marginal in comparison with the average value, which indicates the performance of our technique is stable under the setting of different ε value.

Further, the sample percent ρ influences the efficiency of test report inspection. I present the average APFD score with the changes of paramter ρ in Table 7.6 and Figure 7.4, given $\beta = 1.0$ and $\varepsilon = 0.8$. In Table 7.6, I observe that the APFD scores of these six subject programs reach the highest value under different ρ value. However, the standard deviation of the APFD value presented in Table 7.6 varies in 0.004∼0.012, and the curves are shown in Figure 7.4 are relatively smooth. This fact proves our technique is relatively stable under the different settings of sample percent ρ.

Summary: While all three parameters influence the performance of our technique to a different extent, the performance of our technique is generally stable to their changes. Because the experiment result indicates setting the weight of textual description and screenshots equally can make our technique perform well, I suggest the users of our technique adjust β ***starts from 1. Similarly, I suggest the users of our technique set the default value of*** ε ***into 0.8.***

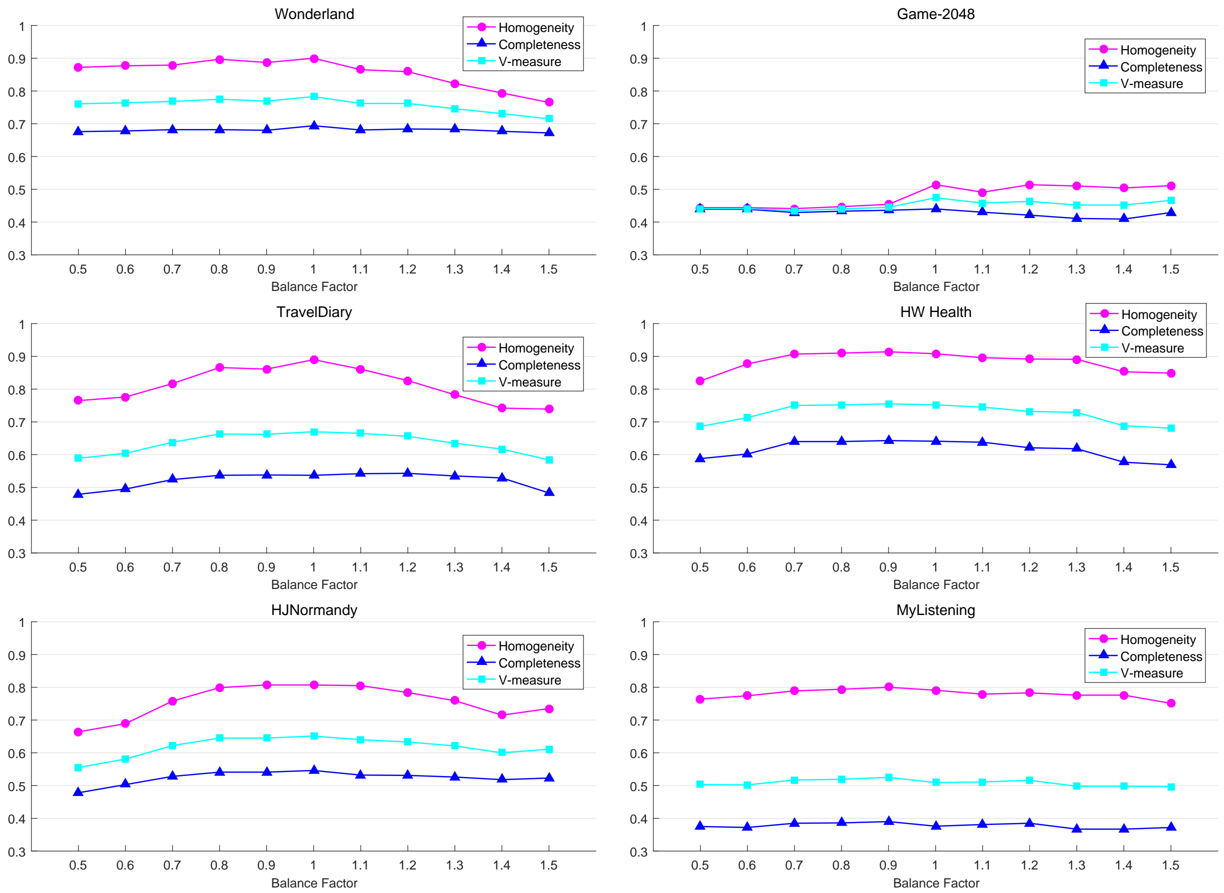


Figure 7.3: The sensitivity of clustering results to the parameter β ($\varepsilon = 0.8$ and $\rho = 0.1$)

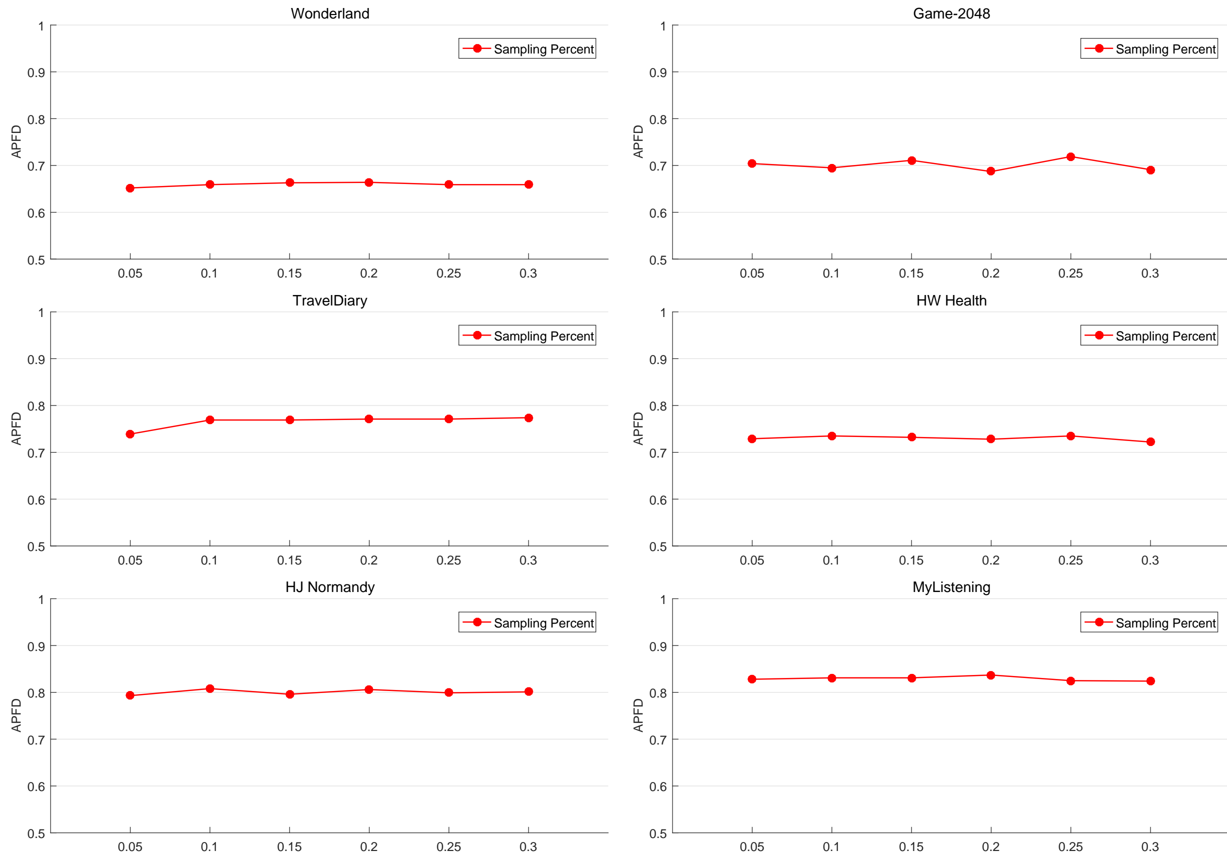


Figure 7.4: The sensitivity of APFD to the parameter ρ

($\beta = 1.0$ and $\varepsilon = 0.8$)

Table 7.5: The comparison of APFD mean value under different settings of ε

(over 30 executions, $\beta = 1.0$ and $\rho = 0.1$)

ε	$p1$	$p2$	$p3$	$p4$	$p5$	$p6$
0.5	0.626	0.634	0.684	0.677	0.781	0.808
0.6	0.642	0.675	0.696	0.701	0.802	0.806
0.7	0.649	**0.711**	0.686	0.681	0.779	0.820
0.8	**0.659**	0.695	**0.769**	**0.735**	**0.808**	**0.831**
0.9	0.655	0.701	0.724	0.721	0.661	0.809
avg	0.646	0.683	0.712	0.703	0.766	0.815
std	0.012	0.027	0.032	0.022	0.054	0.009

Table 7.6: The comparison of APFD mean value under different settings of ρ

(over 30 executions, $\beta = 1.0$ and $\varepsilon = 0.8$)

ρ	$p1$	$p2$	$p3$	$p4$	$p5$	$p6$
0.05	0.652	0.704	0.739	0.729	0.793	0.828
0.1	0.659	0.695	0.769	**0.735**	**0.808**	0.831
0.15	0.663	0.711	0.769	0.732	0.796	0.831
0.2	**0.664**	0.687	0.771	0.728	0.806	**0.837**
0.25	0.659	**0.719**	0.771	**0.735**	0.799	0.825
0.3	0.659	0.691	**0.774**	0.722	0.801	0.824
avg	0.659	0.701	0.766	0.73	0.8	0.829
std	0.004	0.011	0.012	0.004	0.005	0.004

7.3 Threats to Validity

Subject Program Selection. Although crowdsourced testing has covered a wide range of mobile platforms (*e.g.,* Android, IOS, WP), the limitations of data sources have allowed us to experiment with only six Android applications. I cannot guarantee the similar good results could generalize beyond the platforms. Nevertheless, this risk could be reduced because our subject applications vary different categories that diversifies the functionalities including health assistant, entertainment, travel assistant, diary editor, and language learning tools. Thus, I believe these applications can indicate the effectiveness and applicability of our methods.

Natural Language Selection. In this experiment, all the crowdsourced test reports are written in Chinese, which implies the similar results may not be observed based on the test reports written in other languages. However, the natural language processing technique is not the focus of our research. Instead, it works as the ancillary technique to generate the intermediate outputs of our techniques. Even though our technique involves natural language processing, this part focuses on building the keyword vector models to compute the text distance between reports. To build keyword vector models from different languages, many sophisticated methods and NLP tools are available, such as the CoreNLP, WordNet, NLTK. This fact illustrates the transplantable potential of our technique.

Crowd Workers. To collect the experimental data and validate our technique, I collaborated with several mobile application development companies and hosted a national contest. In this contest, students play the role of crowd workers. This compromising choice means that the population of our crowd workers may be less diverse than the population from the general populace. In theory, crowdsourcing techniques require workers to come from a large workforce pool. In this pool, individuals often have no relationship with each others [81]. Thus, this requirement 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.* [110], if a technique or task is new to both students and professionals, similar performance can be expected to be observed. In our experiment, I control all crowd workers have no experience in developing or using these subject applications. All testing tasks are new to these crowd workers. Thus, I believe this threat may not be a critical problem for our validation procedure.

7.4 Conclusion

In this paper, I proposed a novel clustering technique to alleviate the challenge of inspecting the overwhelming number of reports in crowdsourced software testing. In our preliminary investigation, mobile crowdsourced test reports usually contain shorter text descriptions and abundant screenshots. This fact motivates us to utilized image-understanding techniques to assist the traditional text-based techniques, and I proposed approaches for clustering test reports based on a hybrid information source. To the best of our knowledge, this is the first work to propose using image-understanding techniques to improve the accuracy and efficiency in test report clustering. I present the experimental results on four real industrial mobile crowdsourced projects, and evaluate the results from the standpoints of effectiveness, usefulness, and potential. I found that clustering-sampling technique, in almost all cases, is advantageous as compared to test report inspection with an orderless strategy. I also found that for most applications I studied, the practical usefulness which adopts the image-understanding technique are more promising, even if there is a minor class of applications may not be as applicable. As such, in future work, I will improve our technique to help to cluster for these classes of applications, and narrow the gap between our technique and the hypothetical ideal strategy.

Chapter 8

Conclusion

This dissertation presents a family of techniques to process crowdsourced test reports by employing natural language processing, image understanding, and information retrieval techniques. It also contains comprehensive studies that show the effectiveness and efficiency of these techniques on industrial applications.

These techniques provide a comprehensive solution to improve the efficiency of inspecting, analyzing and understanding crowdsourced test reports under different usage scenarios and settings. Specifically, the text-analysis-based crowdsourced test report prioritization technique can assist requesters in revealing as many faults as possible and also as early as possible. This technique is designed for these test reports that contain only textual descriptions. Aiming at the same goal, the image-understanding-based crowdsourced test report prioritization technique is designed for testing tasks of mobile applications. It can leverage both image features and text features. I have compared the effectiveness of the image-understanding-based and text-analysis-based prioritization techniques on mobile apps. The experiment results show the image-understanding-based prioritization technique can obtain a significant improvement regarding AFPD value. The crowdsourced test report aggregation

and summarization technique, **CTRAS**, is capable of automatically aggregating duplicates based on both textual information and screenshots, and further summarizes the duplicate test reports into a comprehensive and comprehensible report. To validate **CTRAS**, I conduct quantitative studies using more than 5000 test reports, collected from 12 industrial projects. The experimental results reveal that **CTRAS** can reach an accuracy of 0.87, on average, regarding automatically detecting duplicate reports, and it outperforms the classic Max-Coverage-based and MMR summarization methods under Jensen Shannon divergence metric. Furthermore, based on **CTRAS**, I implement a web-based tool to assist requesters in managing these reports. The tool includes many test report management functionalities, including bug triaging, duplicate bug-report aggregation, expert recommendation, and so on. And finally, I conduct a comprehensive empirical study to validate the effectiveness of the image features in clustering crowdsourced test reports. In this study, I analyze the parameter sensitivity of this technique, and further present suggests on settings for different application scenarios.

8.1 Contributions

The research presented in this dissertation provides a number of contributions for the field of software engineering:

1. A comprehensive literature review on the crowdsourced software testing;
2. A text-analysis-based approach that combines the risk-assessment and diversity together to prioritize the crowdsourced test reports;
3. An approach that takes the image information, as well as the text information of the crowdsourced test, reports into consideration to assist the inspection procedure;

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

5. An approach that can aggregate multiple duplicate test reports and further summarize them into an enriched report;

6. A web-based test report management tool, which is capable of identifying useful information from the duplicates to supplement the main topics and further assist developers to understand the reports;

7. An empirical study that investigates the effectiveness of the image features in clustering crowdsourced test reports;

8. An empirical evaluation on the parameter sensitivity of image-understanding-based crowdsourced test report clustering technique;

8.2 Future Work

This dissertation motivates future research efforts in four directions as follows.

Collaboration and Coordination in Crowdsourced Software Testing. Some studies [113, 130] have shown that contributions from many parties in a collaborative manner can significantly improve the quality of testing work. Based on these studies, several research groups have proposed a number of frameworks [7], approaches [44], and tools [6] to improve the collaborative testing process. In crowdsourced testing, requesters can obtain the testing results at a low cost by outsourcing testing tasks online to a large group of people in the form of an open call [74], Existing crowdsourcing platforms have made the worker-task matching process efficient; however, they fail to provide service for crowd testers to collaborate with

each other. Under this situation, crowd testers work on the crowdsourcing task without the awareness of the status of other workers and tasks. This lack may reduce the quality of crowdsourced testing results and increase the cost. Future research can explore the method to support collaborative testing in a crowdsourcing environment.

Leveraging Crowd Knowledge to Guide Automated Software Testing.. Automated mobile testing has achieved tremendous advancements in recent years [16, 43]. These fully-automated tools have significantly improved the efficiency and efficacy of software testing. They adopt some classical criteria, such as structural code coverage and diversity of inputs, to guide the testing process. However, some recent research shows that these fully-automated tools can save efforts on the developer's side but are incapable of finding complex bugs [101]. This lack results from the missing of human knowledge and makes these tools difficult to be applied widely. The features of crowdsourcing can be leveraged to address this problem. Researchers can propose semi-automated techniques to assist in software testing by leveraging the knowledge of crowd workers as guidance. For example, building platforms or tools to ease and guide crowds in contributing their domain knowledge to generate test cases. These test cases contributed by crowd workers can reflect the real usage scenarios better and be effective complementarity of automated generated ones [82].

Incentive Mechanism Design for Crowdsourced Software Engineering.. For crowdsourced software engineering, one of the primary challenges of crowdsourcing is to build a capable and reliable workforce. Because the inherent difference in skills and knowledge of crowd workers, crowd workers' expectations about effort and rewards for completing the same task vary widely. Expectations about effort and rewards can significantly influence people's preferences, decisions or behaviours [52, 54]. Incentive mechanisms, which define the task-worker matching and rewarding strategy, can be introduced to optimize the worker gathering process and effort-reward model for crowdsourced testing. However, the existing crowdsourcing platforms provide limited service and assistant for requesters. This lack may

make it difficult to attract qualified crowd workers and thus negatively influence the quality of testing results. The future research can lay on, but not limited to, the following questions: 1. how to automatically assign the reward for each task, given the fixed amount of budget, number of tasks, and estimated difficulty of solving each task? 2. how to automatically estimate the time cost of completing a task based on its text description? 3. how to automatically recommend tasks to workers, taking consideration of their history performance and reward expectations?

Tools for Crowdsourced Software Testing.. Designing and implementing tools can facilitate crowdsourced software testing and benefit both crowd workers and requesters in various perspectives. First, automated tools should be provided to ease crowd workers to complete tasks and thus involve more workforce. Given the fact that crowdsourcing testing has been applied to assist in various testing tasks, it is natural that some tasks may require crowd workers to be experienced or skilful in software testing techniques. For these tasks, the technical barrier makes gathering qualified crowd workers challenging for their requesters.

Several automated tools have been developed for crowdsourced testing to lower the technical barrier. For example, the Applause crowdsourced testing platform provides a set of tools [1] to support the testing process [9]. These tools help crowd workers to automatically record runtime information in the on-going testing sessions, including program execution profiles, devices, environments, locations, and so on. Even though these tools have succeeded in saving crowd testers plenty of time cost of capturing a bug of the mobile app under test and taking screenshots to file the bug report, they failed to guide for crowd workers to detect bugs and analyze the potential vulnerabilities.

On the other hand, designing tools that enable requesters to monitor, evaluate, analyze and review the testing processes and results is critical for bug reproduction. They are critical for improving the efficiency of the bug diagnosis and fix. These tools should be integrated with

[1] www.applause.com/platform

data analytics to handle large scale of reports submitted by crowd workers, and further, provide the insights and critical features of the testing results. Further, considering the feedbacks from crowd workers are not limited to texts, these tools should be able to support various data types, such as screenshots, voice records and videos.

Bibliography

[1] *Alibaba Crowd Test*. https://mqc.aliyun.com/crowdtest.

[2] A. Alipour, A. Hindle, and E. Stroulia. A contextual approach towards more accurate duplicate bug report detection. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 183–192. IEEE, 2013.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[4] P. Awasthi, D. Rao, and B. Ravindran. Part of speech tagging and chunking with hmm and crf. *Proceedings of NLP Association of India (NLPAI) Machine Learning Contest 2006*, 2006.

[5] Y. Bachrach, T. Graepel, T. Minka, and J. Guiver. How to grade a test without knowing the answers—a bayesian graphical model for adaptive crowdsourcing and aptitude testing. *arXiv preprint arXiv:1206.6386*, 2012.

[6] X. Bai, Z. Cao, and Y. Chen. Design of a trustworthy service broker and dependence-based progressive group testing. *International Journal of Simulation and Process Modelling*, 3(1-2):66–79, 2007.

[7] X. Bai, Y. Wang, G. Dai, W.-T. Tsai, and Y. Chen. A framework for contract-based collaborative verification and validation of web services. In *International Symposium on Component-Based Software Engineering*, pages 258–273. Springer, 2007.

[8] *Baidu Crowd Test*. http://test.baidu.com.

[9] E. Bari, M. Johnston, W. Wu, and W.-T. Tsai. Software crowdsourcing practices and research directions. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 372–379. IEEE, 2016.

[10] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[11] N. Bettenburg, R. Premraj, and T. Zimmermann. Duplicate bug reports considered harmful ... really? In *IEEE International Conference on Software Maintenance*, pages 337–345, 2008.

[12] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source android apps. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 133–143. IEEE, 2013.

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

[14] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[15] *Bugzilla*. https://www.bugzilla.org.

[16] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[17] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. Vips: a vision-based page segmentation algorithm. 2003.

[18] J. Carbonell and J. Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336. ACM, 1998.

[19] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, I. d. C. Machado, T. F. Vale, E. S. de Almeida, and S. R. d. L. Meira. Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process*, 26(7):620–653, 2014.

[20] 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.

[21] F. Chen and S. Kim. Crowd debugging. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 320–332. ACM, 2015.

[22] K.-T. Chen, C.-J. Chang, C.-C. Wu, Y.-C. Chang, and C.-L. Lei. Quadrant of euphoria: a crowdsourcing platform for qoe assessment. *IEEE Network*, 24(2):28–35, 2010.

[23] K.-T. Chen, C.-C. Wu, Y.-C. Chang, and C.-L. Lei. A crowdsourceable qoe evaluation framework for multimedia content. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 491–500. ACM, 2009.

[24] N. Chen and S. Kim. Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 140–149. IEEE, 2012.

[25] 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.

[26] 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 *2012 34th International Conference on Software Engineering (ICSE)*, pages 1084–1093. IEEE, 2012.

[27] J. Deshmukh, S. Podder, S. Sengupta, N. Dubash, et al. Towards accurate duplicate bug retrieval using deep learning techniques. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*, pages 115–124. IEEE, 2017.

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

[29] R. C. Dubes and A. K. Jain. Algorithms for clustering data, 1988.

[30] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2014.

[31] 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.

[32] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu. Test report prioritization to assist crowdsourced testing. In *ESEC/SIGSOFT FSE*, pages 225–236, 2015.

[33] Y. Feng, J. A. Jones, Z. Chen, and C. Fang. Multi-objective test report prioritization using image understanding. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 202–213. IEEE, 2016.

[34] E. Filatova and V. Hatzivassiloglou. A formal model for information selection in multi-sentence text extraction. In *Proceedings of the 20th international conference on Computational Linguistics*, page 397. Association for Computational Linguistics, 2004.

[35] S. Foo and H. Li. Chinese word segmentation and its effect on information retrieval. *Information processing & management*, 40(1):161–190, 2004.

[36] B. Gardlo. Quality of experience evaluation methodology via crowdsourcing. *Unpublished doctoral dissertation, University of Zilina, Zilina*, 2012.

[37] B. Gardlo, S. Egger, and M. C. Seufert. 2.0: Enhancing execution speed and reliability of web-based qoe testing. *Proc. IEEE ICC, Sydney, Australia (June 2014).*

[38] B. Gardlo, M. Ries, T. Hoßfeld, and R. Schatz. Microworkers vs. facebook: The impact of crowdsourcing platform choice on experimental results. In *2012 Fourth International Workshop on Quality of Multimedia Experience*, pages 35–36. IEEE, 2012.

[39] M. Gegick, P. Rotella, and T. Xie. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 11–20. IEEE, 2010.

[40] J. Goldstein, M. Kantrowitz, V. Mittal, and J. Carbonell. Summarizing text documents: sentence selection and evaluation metrics. In *SIGIR*, volume 99, page 99, 1999.

[41] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 88–99. IEEE, 2016.

[42] S. Guo, R. Chen, and H. Li. A real-time collaborative testing approach for web application: Via multi-tasks matching. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 61–68. IEEE, 2016.

[43] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[44] M. Held and W. Blochinger. Structured collaborative workflow design. *Future Generation Computer Systems*, 25(6):638–653, 2009.

[45] A. Hindle. Stopping duplicate bug reports before they start with continuous querying for bug reports. Technical report, PeerJ Preprints, 2016.

[46] A. Hindle, A. Alipour, and E. Stroulia. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empirical Software Engineering*, 21(2):368–410, 2016.

[47] T. Hossfeld, C. Keimel, M. Hirth, B. Gardlo, J. Habigt, K. Diepold, and P. Tran-Gia. Best practices for qoe crowdtesting: Qoe assessment with crowdsourcing. *IEEE Transactions on Multimedia*, 16(2):541–558, 2013.

[48] T. Hossfeld, C. Keimel, and C. Timmerer. Crowdsourcing quality-of-experience assessments. *Computer*, 47(9):98–102, 2014.

[49] J. Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[50] M. Hu and B. Liu. Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 168–177. ACM, 2004.

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

[52] M. O. Jackson and L. Yariv. Diffusion, strategic interaction, and social structure. In *Handbook of social Economics*, volume 1, pages 645–678. Elsevier, 2011.

[53] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

[54] D. Jeske and C. Axtell. Effort and reward effects: Appreciation and self-rated performance in e-internships. *Social Sciences*, 6(4):154, 2017.

[55] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244. IEEE, 2009.

[56] H. Jiang, X. Chen, T. He, Z. Chen, and X. Li. Fuzzy clustering of crowdsourced test reports for apps. *ACM Transactions on Internet Technology (TOIT)*, 18(2):18, 2018.

[57] H. Jiang, N. Nazar, J. Zhang, T. Zhang, and Z. Ren. Prst: A pagerank-based summarization technique for summarizing bug reports with duplicates. *International Journal of Software Engineering and Knowledge Engineering*, 27(06):869–896, 2017.

[58] H. Jiang, J. Zhang, H. Ma, N. Nazar, and Z. Ren. Mining authorship characteristics in bug repositories. *Science China Information Sciences*, 60(1):012107, 2017.

[59] *Jira*. https://www.atlassian.com/software/jira.

[60] D. Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

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

[62] S. Komarov, K. Reinecke, and K. Z. Gajos. Crowdsourcing performance evaluations of user interfaces. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 207–216. ACM, 2013.

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

[64] 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.

[65] T. D. LaToza, M. Chen, L. Jiang, M. Zhao, and A. Van Der Hoek. Borrowing from the crowd: A study of recombination in software design competitions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 551–562. IEEE, 2015.

[66] T. D. LaToza, A. Di Lecce, F. Ricci, W. B. Towne, and A. Van der Hoek. Microtask programming. *IEEE Transactions on Software Engineering*, 2018.

[67] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. Van Der Hoek. Microtask programming: Building software with a crowd. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 43–54. ACM, 2014.

[68] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2010.

[69] A. Lazar, S. Ritchey, and B. Sharif. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 308–311. ACM, 2014.

[70] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2169–2178. IEEE, 2006.

[71] Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 510–514. IEEE Computer Society, 2009.

[72] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 451–460. IEEE, 2012.

[73] C. Liu, X. Zhang, and J. Han. A systematic study of failure proximity. *IEEE Transactions on Software Engineering*, 34(6):826–843, 2008.

[74] 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.

[75] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the hurried bug report reading process to summarize bug reports. *Empirical Software Engineering*, 20(2):516–548, 2015.

[76] A. Louis and A. Nenkova. Automatically evaluating content selection in summarization without human models. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 306–314. Association for Computational Linguistics, 2009.

[77] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *ACM Sigsoft International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[78] S. Mani, A. Sankaran, and R. Aralikatte. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. *arXiv preprint arXiv:1801.01275*, 2018.

[79] *Mantis*. https://www.mantisbt.org.

[80] M. V. Mäntylä and J. Itkonen. More testers–the effect of crowd size and time restriction in software testing. *Information and Software Technology*, 55(6):986–1003, 2013.

[81] K. Mao, L. Capra, M. Harman, and Y. Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017.

[82] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 16–26. IEEE Press, 2017.

[83] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 131–140. IEEE, 2009.

[84] F. Meier, A. Bazo, M. Burghardt, and C. Wolff. Evaluating a web-based tool for crowdsourced navigation stress tests. In *International Conference of Design, User Experience, and Usability*, pages 248–256. Springer, 2013.

[85] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[86] *Mooctest*. http://www.mooctest.net.

[87] B. Morschheuser, J. Hamari, J. Koivisto, and A. Maedche. Gamified crowdsourcing: Conceptualization, literature review, and future agenda. *International Journal of Human-Computer Studies*, 106:26–43, 2017.

[88] G. Murphy and D. Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[89] M. Nebeling, S. Leone, and M. C. Norrie. Crowdsourced web engineering and design. In *International Conference on Web Engineering*, pages 31–45. Springer, 2012.

[90] M. Nebeling, M. Speicher, and M. C. Norrie. Crowdstudy: General toolkit for crowdsourced evaluation of web interfaces. In *Proceedings of the 5th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 255–264. ACM, 2013.

[91] 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 *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 70–79. ACM, 2012.

[92] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, pages 75–84. IEEE Computer Society, 2007.

[93] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

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

[95] O. Pele and M. Werman. The quadratic-chi histogram distance family. In *European conference on computer vision*, pages 749–762. Springer, 2010.

[96] F. Petrillo, G. Lacerda, M. Pimenta, and C. Freitas. Visualizing interactive and shared debugging sessions. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 140–144. IEEE, 2015.

[97] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Guéhéneuc. Towards understanding interactive debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 152–163. IEEE, 2016.

[98] N. Pingclasai, H. Hata, and K.-i. Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 13–18. IEEE, 2013.

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

[100] 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.

[101] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[102] M. M. Rahman, G. Ruhe, and T. Zimmermann. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 439–442. IEEE Computer Society, 2009.

[103] M. S. Rakha, C.-P. Bezemer, and A. E. Hassan. Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports. *IEEE Transactions on Software Engineering*, 44(12):1245–1268, 2017.

[104] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 505–514. ACM, 2010.

[105] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 40(4):366–380, 2014.

[106] A. Rosenberg and J. Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420, 2007.

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

[108] 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.

[109] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007.

[110] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 666–676. IEEE, 2015.

[111] H. Schütze, C. D. Manning, and P. Raghavan. Introduction to information retrieval. In *Proceedings of the international communication of association for computing machinery conference*, page 260, 2008.

[112] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423. Springer, 2006.

[113] H. Shahriar and M. Zulkernine. Trustworthiness testing of phishing websites: A behavior model-based approach. *Future Generation Computer Systems*, 28(8):1258–1271, 2012.

[114] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu. Measuring the diversity of a test set with distance entropy. *IEEE Transactions on Reliability*, 65(1):19–27, 2015.

[115] E. Shutova and S. Teufel. Metaphor corpus annotated for source-target domain mappings. In *LREC*, volume 2, pages 2–2, 2010.

[116] P. R. Srivastava. Test case prioritization. *Journal of Theoretical & Applied Information Technology*, 4(3), 2008.

[117] O. Starov. Cloud platform for research crowdsourcing in mobile testing. 2013.

[118] 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.

[119] 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.

[120] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference*, pages 366–374. IEEE, 2010.

[121] H. Takamura and M. Okumura. Text summarization model based on maximum coverage problem and its variant. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 781–789. Association for Computational Linguistics, 2009.

[122] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375. ACM, 2011.

[123] D. Tao, L. Jin, W. Liu, and X. Li. Hessian regularized support vector machines for mobile image annotation on the cloud. *IEEE Transactions on Multimedia*, 15(4):833–844, 2013.

[124] A. Teinum. User testing tool: towards a tool for crowdsource-enabled accessibility evaluation of websites. Master's thesis, Universitetet i Agder; University of Agder, 2013.

[125] *Testin*. http://www.testin.net.

[126] *TestIO*. https://test.io.

[127] 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.

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

[129] N. Tillmann and J. de Halleux. White box test generation for .net. *TAP'08*, 4966:133–153, 2008.

[130] W.-T. Tsai, Y. Chen, R. Paul, N. Liao, and H. Huang. Cooperative and group testing in verification of dynamic composite web services. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 170–173. IEEE, 2004.

[131] Y.-H. Tung and S.-S. Tseng. A novel approach to collaborative testing in a crowdsourcing environment. *Journal of Systems and Software*, 86(8):2143–2153, 2013.

[132] *uTest*. https://www.utest.com.

[133] R. Vliegendhart, E. Dolstra, and J. Pouwelse. Crowdsourced user interface testing for multimedia applications. In *Proceedings of the ACM multimedia 2012 workshop on Crowdsourcing for multimedia*, pages 21–22. ACM, 2012.

[134] S. Wang, W. Zhang, and Q. Wang. Fixercache: Unsupervised caching active developers for diverse bug triage. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 25. ACM, 2014.

[135] 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 *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 461–470. IEEE, 2008.

[136] C.-C. Wu, K.-T. Chen, Y.-C. Chang, and C.-L. Lei. Crowdsourcing multimedia qoe evaluation: A trusted framework. *IEEE transactions on multimedia*, 15(5):1121–1137, 2013.

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

[138] H. Xue. *Using redundancy to improve security and testing*. PhD thesis, University of Illinois at Urbana-Champaign, 2013.

[139] M. Yan, H. Sun, and X. Liu. itest: testing software with mobile crowdsourcing. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*, pages 19–24. ACM, 2014.

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

[141] K. Zhang, H. Xu, J. Tang, and J. Li. Keyword extraction using support vector machine. In *international conference on web-age information management*, pages 85–96. Springer, 2006.

[142] T. Zhang, J. Chen, X. Luo, and T. Li. Bug reports for desktop software and mobile apps in github: What's the difference? *IEEE Software*, 36(1):63–71, 2017.

[143] T. Zhang, J. Gao, and J. Cheng. Crowdsourced testing services for mobile apps. In *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 75–80. IEEE, 2017.

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