On the Identification of Accessibility Bug Reports in Open Source Systems

Wajdi Aljedaani
wajdialjedaani@my.unt.edu
University of North Texas

Mohamed Wiem Mkaouer
mwmvse@rit.edu
Rochester Institute of Technology

Stephanie Ludi
Stephanie.Ludi@unt.edu
University of North Texas

Ali Ouni
ali.ouni@etsmtl.ca
ETS Montreal, University of Quebec

Ilyes Jenhani
ijenhani@pmu.edu.sa
Prince Mohammad Bin Fahd University

Abstract

Today, mobile devices provide support to disabled people to make their life easier due to their high accessibility and capability, e.g., finding accessible locations, picture and voice-based communication, customized user interfaces and vocabulary levels. These accessibility frameworks are directly integrated, as libraries, in various apps, providing them with accessibility functions. Just like any other software, these frameworks regularly encounter errors. These errors are reported by app developers in the form of bug reports. These bug reports related to accessibility faults need to be urgently fixed since their existence significantly hinders the usability of apps. In this context, the manual inspection of a large number of bug reports to identify accessibility-related ones is time-consuming and error-prone. Prior research has investigated mobile app user reviews classification for various purposes, including bug reports identification, feature request identification, app performance optimization etc. Yet, none of the prior research has investigated the identification of accessibility-related bug reports, making their prioritization and timely correction difficult for software developers. To support developers with this manual process, the goal of this paper is to automatically detect, for a given bug report, whether it is about accessibility or not. Thus, we tackle the identification of accessibility bug reports as a binary classification problem. To build our model, we rely on an existing dataset of manually curated accessibility bug reports, extracted from popular open-source projects, namely Mozilla Firefox and Google Chromium. We design our solution to learn from these reports the appropriate discriminative features i.e., keywords that properly represent accessibility issues. Our trained model is evaluating using stratified cross-validation, and the findings show that our classifier achieves high F1-scores of 93%.

W4A'22, April 25–26, 2022, Lyon, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9170-2/22/04…$15.00
https://doi.org/10.1145/3493612.3520471

CCS Concepts

• **Human-centered computing → Empirical studies in accessibility**; **Ubiquitous and mobile devices**.

Keywords

Bug Report, Accessibility, Machine Learning, Bug Repository, Open Source.

ACM Reference Format:
Wajdi Aljedaani, Mohamed Wiem Mkaouer, Stephanie Ludi, Ali Ouni, and Ilyes Jenhani . 2022. On the Identification of Accessibility Bug Reports in Open Source Systems. In *19th Web for All Conference (W4A'22), April 25–26, 2022, Lyon, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3493612.3520471

1 Introduction

Open source and industrial software utilize bug-tracking systems — also called issue-tracking systems — such as Bugzilla [2, 4, 12]. These tracking systems are used to help developers maintain the software by allowing the end-users to submit the issue description they faced while they are using the software. Bug reports can describe accessibility issues that could have prevented or limited users with a disability, special needs, or functional constraints.

People with disabilities or special needs rely heavily on accessibility software applications in their everyday life (find accessible location, customized UIs, voice translation, communication, driving, shopping, etc.). Having accessibility-related bugs can have severe impacts on their lives that can go from preventing them from participating in new activities, to threatening their lives in critical situations due to the sensitive nature of disabled people. Therefore, identifying and prioritizing these bugs are of crucial importance. Yet, the manual identification of these bug reports is time-consuming, human-intensive, and error-prone. The textual nature of bug reports adds another layer of challenge related to the meaning ambiguity of these natural language descriptions. To illustrate this problem, let us consider the following two examples:

> ***Example 1:*** *"Missing labels on the buttons in the "Select how you want to use Weave"* [1]
>
> ***Example 2:*** *"Performance issue: TextArea very slow when accessibility API turned on"* [2]

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=533573
[2] https://bugs.chromium.org/p/chromium/issues/detail?id=868830

While the first bug report describe a missing textual label in a graphical component, making it not accessible for blind users, the second bug report is related to a performance issue. Despite containing the keyword accessibility, this bug is not related to the accessibility of the software, but to a performance regression detected when integrating the accessibility library, through its API, to the system. These examples show that we cannot rely on the keyword accessibility to identify accessibility related bug reports, as the first example (accessibility bug report) did not contain the keyword *accessibility*, while the second example (non-accessibility bug report) did.

To support software developers with the correction of accessibility errors in their systems, we propose a classification-based approach for the automatic detection of accessibility bug reports. However, the detection of such reports is challenging, besides the inherited ambiguity of distinguishing meanings, in any natural language text, the above example show how the keyword *accessibility* can be misleading, which hardens the reliance on that keyword alone. To cope with these challenges, we design our study to harvest a potential terminology that can be used to describe accessibility errors and faults.

Our approach relies on Natural Language Processing (NLP) techniques to distill from a training sample (set of accessibility bug reports) the proper *features*, i.e., phrases that tend to specifically describe accessibility related faults in code. We performed our study on seven open-source systems hosted in two popular issue tracking systems Bugzilla [12] and Monorail [34] repositories. We mine all the bug reports for the selected projects to identify accessibility and non-accessibility bug reports based on their tags (manual inspection). To the best of our knowledge, this is the first study that builds classification models to classify bug reports and identify accessibility issues.

Specifically, we address the following research questions:

RQ1: *Can we accurately detect accessibility-related bug reports?* Our aim is to design an approach that can automatically identify accessibility-related bug reports. Therefore, we put under test, various classifiers, such as neural networks, decision trees, and SVM, known to be efficient and widely used for binary classification problems. Answering to this research question would reveal the best performing model that we should deploy for our current problem, along with showing how much we can advance the state-of-the-art of detecting accessibility-related bug reports.

RQ2: *What is the size of the training dataset needed for the classification to effectively identify accessibility bug reports?* After evaluating the accuracy of our model, we analyze the number of bug reports needed for training in order to achieve our optimal model classification accuracy. We anticipate our model to be easily exported and extended if it can achieve an acceptable performance using a relatively small set of training data. Otherwise, if the model requires a large number of bug reports, for training, then we report a need for a considerable time and effort for labeling.

To summaries, the paper makes the following contributions:

- We present an automatic accessibility identification on seven open-source systems to identify accessibility-related bug reports by using machine learning algorithms. To the best of our knowledge, this is the first accessibility classification study to date on the bug reports dataset.
- An experimental study on a real world dataset of 256,700 bug reports. Our key findings show that our model accurately identifies accessibility-related bug reports achieves high F1-scores of 93%. Furthermore, we infer which features, *i.e.,* keywords, are relevant for the detection of such type of bug reports
- We also publicly provide our dataset that served us as the *ground-truth*, for replication and extension purposes[3].

Table 1: List of the keywords used to further verify bug reports related to accessibility.

Guideline	Keywords
Principles	accessibility, disability, screen reader, talkback, operable, impaired, impairment
Audio/video	Subtitle, sign language, audio description, transcript, blind, visual cue
Forms	unique label, missing label
Text equivalent	alternative text, non-visual, content description

Paper organization. Section 2 summarizes the related work. Section 3 describes our process of the classification approach. It illustrates the process in which we prepare the data collection, data preprocessing, data transformation, data classification, and the machine learning algorithms used in our study. Section 4 presents and discusses the results of our two research questions. We discuss the threats to validity in Section 5. Finally, we conclude the paper in Section 6.

2 Related Work

In software development and management of large-scale applications, bugs databases have become a crucial archive. They provide developers with valuable details and encourage users to notify developers of the problems facing users through the use of the software. Many studies primarily refer to the bug repositories due to their significance. This section presents two aspects of similar studies, and shows the differences between our research and the corresponding study.

2.1 Classification in Open-Source Repository

Previous studies have conducted Classifications on different aspects of bug reports [3, 39]. For example, Zhou et al. [49] proposed an automation approach to identify security-related by using NLP and machine learning techniques. Their study was performed on bug reports and commits of open-source projects from GitHub, Jira, and Bugzilla. They used imbalance data where the security-related bug reports and commits are less than 10% to challenge the classifier. Another study used NASA datasets to identify security and non-security bug reports by applying six algorithms [20]. Peters et al. [38] presented a framework named FARSEC, which is used to filter and rank security bug reports. The authors performed TF-IDF techniques to observe security- related keywords to identify bug reports. Alkhazi et al. [7] trained a learning-to-rank algorithm to recommend suitable developers for fixing a given bug report.

[3]https://smilevo.github.io/access/

Fang et al. [16] built a binary classifier to detect whether a bug report's writing is rich enough for developers to easily locate the bug. Aljedaani et al. [6] proposed an automated sentiment analysis-based approach to classify accessibility user reviews to support the developers detect issues and enhancing their app's performance.

Our work applied a similar process, but we used classifier learning through the use of a statistic machine. We developed a classification model to classify accessibility bug reports. Most accessibility studies were on mobile platforms, in particular, Android, to investigate user review issues. There is no study performed on accessibility bug reports precisely to classify accessibility bug reports in open-source systems.

2.2 Accessibility in Open-Source Applications

Many studies have conducted qualitative mobile-bug reports platform analysis [5, 11, 30] and Android-related bug reporting tool [35]. Markus et al. [31] propose a Braille interface platform named MOST with such a wide range of applications. Al-Subaihin et al. [1] presented an assessment of mobile web application accessibility. McIlroy et al. [32] introduced an automatically labeling approach based on the types of user review issues. Liu et al. [29] conducted a study on Android applications to detect performance bugs to identify common patterns. Alshayban et al. [9] have analyzed 1,000 Android applications based on three perspective developers, users, and applications for accessibility issues. Panichella et al. [37] proposed an approach using machine learning, which incorporated three NLP, sentiment, and text analysis techniques to introduce a taxonomy for classifying user reviews.

Vendome et al. [44] examined the Stack Overflow developer discussions of the Android app's accessibility. They have identified posts based on a list of keywords that have been chosen from the accessibility guide for mobile applications. They analyzed all the questions asked in the Stack Overflow and answers that labeled Android and found 810 out of 1,442. In a study similar to ours, Eler et al. [15] performed an investigation on user reviews related to mobile accessibility. The study applied to user reviews of 701 applications from the Google Play Store. Their approach was to manually analyze the user reviews using a list of more than 200 keywords that refer to mobile accessibility.

3 Methodology

The following section explains our methodology and how we obtained and analyzed the data for classifying accessibility bug reports to answer the research questions of our study. Figure 1 presents an overview about our study which consists of the following main steps :

(A) **Data Collection (Step 1):** As an initial step of our study, we need to collect our experimental dataset which consists of a set of real world bug reports from open source projects. To do so, we mine the bug reports archive of seven selected open-source systems. We have implemented a parser that takes every bug report in the tracker as an input, then verifies whether it was tagged as an accessibility reports. If so, its corresponding information will be copied over to our database. We keep track of the project containing the bug report along with all the its metadata. It is important for us to keep as much information as possible about each bug report, so that the manual analysis that would be coming later would be easier for the authors.

(B) **Data Preprocessing (Step 2):** After the data collection step, we need to pre-process the text and only keep important textual information, which can be used to train a model afterwards [8]. The results of this step put the report's text into a format that the classification algorithms can easily transform. This way, the noise will be removed, allowing for informative featurization. Note that we only pre-process the textual description of the reports, and we do not alter any meta-data information.

(C) **Data Classification (Step 3):** In the final step , we apply machine learning techniques to build a classification model. In particular, a binary classifier is used to classify accessibility bug reports on five widely-used algorithms. We only used bug report description to identify accessibility bug reports.

Table 2: Statistics of the datasets.

System	Platform	#Non-Bug Reports	#Accessibility Bug Reports	Start Date	End Date
Firefox	Firefox	25,000	250	29-09-2000	06-04-2020
	Core	59,900	599	08-04-1997	05-04-2020
Chromium	Mac	30,700	307	23-09-2016	05-03-2020
	Windows	44,200	442	28-09-2016	05-03-2020
	Chrome	41,200	412	08-05-2017	05-03-2020
	Android	34,700	347	10-12-2012	05-03-2020
Apache	NetBeans	21,000	210	14-07-2000	02-01-2018
Total		**256,700**	**2,567**		

3.1 Step 1: Data Collection

Data collection is the first step in our study methodology. Our goal is to analyze bug tracking systems of various open-source software projects where their reports are publicly available. Our study uses two of the large open-source bug report repositories, Bugzilla [12], and Monorail [34]. We chose various project system domains that range from web browsers, mobile platforms, and desktop applications. We have also chosen these projects because they contain accessibility frameworks, integrated as libraries, and heavily used in their systems, to make their content and services accessible. We collected more than 15 projects to be analyzed, as our focus was only on bug reports identified as ***defect*** type. In order to select a project repository to be studied, we selected projects that support the type of bug report in their repositories, and we eliminated the projects that do not support the information of bug report types. From all the 15 projects, there are only seven projects that supported the bug report types The projects that used Bugzilla are Firefox-Platform [4], Firefox Core, Apache NetBeans, and projects that use Monorail are Google Chromium platforms[5] (Android, Windows, Chrome, and Macintosh).

After collecting all the bug reports, we discarded bug reports that were reported in a different language than English, and bug reports that were flagged as invalid, or not relevant. We provided some examples in Table 3.

[4]https://bugzilla.mozilla.org/home
[5]https://bugs.chromium.org/p/chromium/issues/list
[6]https://bugs.chromium.org/p/chromium/issues/detail?id=1024836
[7]https://bugzilla.mozilla.org/show_bug.cgi?id=599707
[8]https://bugzilla.mozilla.org/show_bug.cgi?id=668458
[9]https://bugs.chromium.org/p/chromium/issues/detail?id=910827

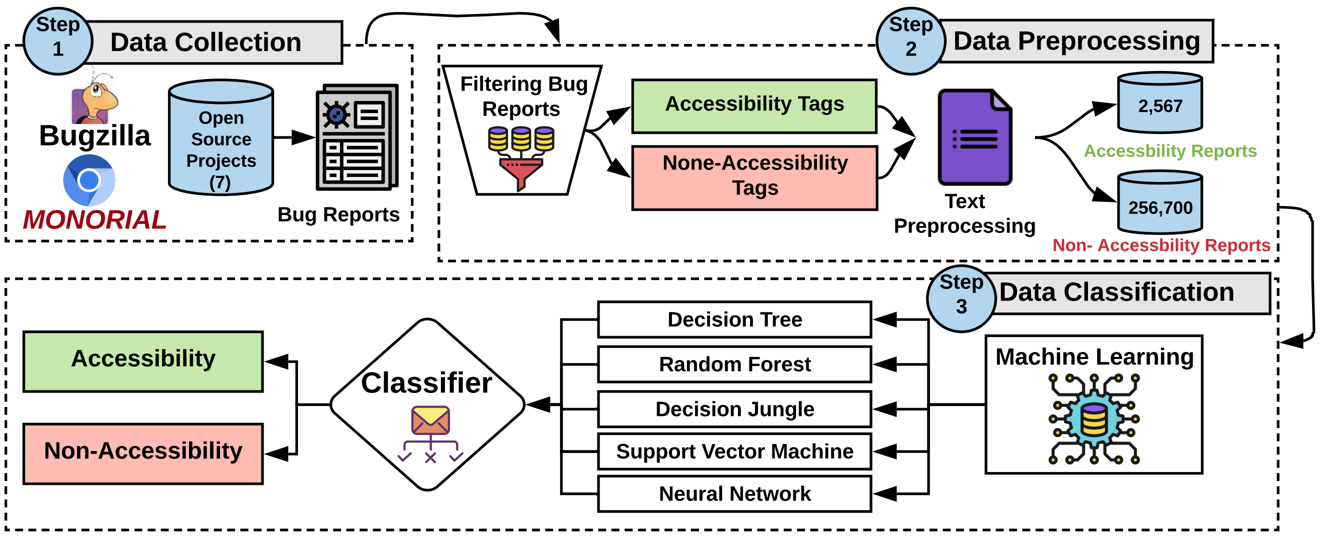

Figure 1: Overview approach of our study.

Table 3: Examples of invalid bug reports.

Type	Description
Non-English	Girdiğim eğitim sitesi güvenlik hatası veriyor[6]
Testing	testing a bug[7]
Non-Meaningful	afdfsadsdsad[8]
Thanking	Thank you[9]

As our accessibility bug reports were gathered based on their accessibility tags by the developers reporting them, and validated using the keywords that exist in the BBC guidelines [10, 45], we followed the process of [27] to further verify the collected data, which are referred to as accessibility bug reports. We randomly selected a 12% sample of bug reports, i.e., 334 out of the 2,567 bug reports. This quantity is equal to a sample size with a confidence level[10] of 95% and a confidence interval of 10. Two of the authors performed the labeling process separately. Both authors were given the same set of bug reports to label to either accessibility related or not. The chosen reports were not previously exposed to the authors. The analysis process took seven days to prevent exhaustion. The authors had the ability to search online for any unknown references in the reports. We cross-check results of the manual labeling to calculate the ratio of agreement and disagreement between the authors. For all cases of disagreement, a third author is requested to re-label the instance and break the tie. We present an example of an agreed on and disagreed on bug report. For the example of the disagreed on bug report, the third author has considered this to be a non-accessibility bug report, as it describes an error with handlers of copying accessibility.

> ***Agreed on Example:*** *"Panning incorrect with Fullscreen Magnifier Accessibility feature enabled while display set in a non-standard tablet rotatio"*[11]

> ***Disagreed on Example:*** *"Copy for accessibility permission incorrect for some PDFs with revision 2 security handlers"*[12]

We adopted Cohen's Kappa coefficient [14] to assess the inter-rater agreement level for the categorical classes. We obtained an agreement level of 0.83. According to Fleiss et al. [18], these agreement values are considered to have an almost *perfect agreement* (*i.e.*, 0.61˜0.80).

To summarize, we only considered the bug report that is typed as a defect or bug. We discarded the bug report that typed as enhancement, task, feature, or patch. After finalizing our target projects, we collected all bug reports archived in each of the selected project systems. The total number of **A**ccessibility **B**ug **R**eports (ABR) are 2,567 while the total number of non-accessibility bug reports are 256,700. Note that after we gathered all the defect bug reports in each project, we randomly selected non-accessibility bug reports. Table 2 illustrates the details of the collected data in the study. Table 1 also showcases the keywords we encountered during our manual analysis, and how they related to various types of accessibility guidelines.

3.2 Data Preprocessing

Next, we text preprocessing (TP) the textual information in each bug report in the *description* field. The bug report description *d* can be mixed with words and different characters, for example, comma, apostrophe, etc. In the text preprocessing, we clean up the documents by removing the unhelpful elements of special characters and stopping words such as "a", "the", "are", etc. Then, we use Natural Language Processing[13] (NLP) for identifying the basis of each word. Words can be written in different grammar styles, but the meaning is similar. During this process, each token shall be removed from appendices, and only the stem will remain. This

[10]https://www.surveysystem.com/sscalc.htm#one
[11]https://bugs.chromium.org/p/chromium/issues/detail?id=1009329
[12]https://bugs.chromium.org/p/chromium/issues/detail?id=989408
[13]https://nlp.stanford.edu/software/

process can help us to minimize the positive impact on the recall performance of the results.

$$\hat{d} = DPP(d) \tag{1}$$

For example, d is a bug report description, then $\widehat{d}$ is generated, using *DPP*. The *DPP* process is explained as below:

Text Preprocessing (*DPP*)

input (d): 'Print dialog too large for screen when accessibility features being used; needs to be resizable'[a]

1- Tokenization: In this step, we transform the textual information "words" into a tokens list as each single token will be processed separately.
['Print', 'dialog', 'too', 'large', 'for', 'screen', 'when', 'accessibility', 'features', 'being', 'used', ';', 'needs', 'to', 'be', 'resizable']

2- Numerical & Special Characters Removal: In this part, all the numbers and special characters (punctuation) will be eliminated, tags for instance ';' from the tokens list.
['Print', 'dialog', 'too', 'large', 'for', 'screen', 'when', 'accessibility', 'features', 'being', 'used', 'needs', 'to', 'be', 'resizable']

3- Stop-Word Removal: is the process of deleting all the common English words as well as reserved words[b] such as "too", "for", "be", "to", "when", "need".
['Print', 'dialog', 'large', 'screen', 'accessibility', 'features', 'used', 'needs', 'resizable']

4- Lemmatization: In this step, we minimize the words' derivationally to the root of the words, which helps remove the inflection. For example *prints, printing, printed,* $\Rightarrow$ *print, features,* $\Rightarrow$ *feature.*
['Print', 'dialog', 'large', 'screen', 'access', 'feature', 'use', 'size']

5- Ouput($\widehat{d}$): This is the final step where all the characters converted into lowercase and then merge all the tokens to a single string.
'Print dialog large screen access feature use size'

[a]https://bugzilla.mozilla.org/show_bug.cgi?id=327939
[b]http://www.textfixer.com/resources/common-english-words.tx

3.3 Data Transformation

Given machine learning, all the algorithms used in machine learning are trained using *feature vectors*. The feature vector is a numerical vector with data in numerical form [33]. The collected data that we used in this study does not come in vector form. Therefore, those data have to transform into feature vectors using feature extraction before using the data to train the machine learning algorithms. To transform the data to feature vector, we used the feature hashing technique known as the Hashing trick. Feature hashing is a popular and powerful technique in machine learning for handling spares and high dimensional features [46]. It is applied to reduce the dimensionality of the analyzed data [40, 47]. The hashing function does this transformation in the feature hashing technique. For example, if we input a bug report *description* using the feature hashing technique, it will output a fixed-length size of a hash value. Figure 2 shows an example of how feature hashing works.

There are several popular schemes for feature encoding or extraction like Bag-of-words, TF-IDF, etc. However, there are issues in the mentioned techniques, such as the curse of dimensionality as well as semantic selection. Prior studies have shown that feature hashing is a robust approach to achieve fast similarity search [22, 23, 43]. If we use a feature selection technique other than the feature hashing technique in this study, it will select a subset from the original features and reduce the parameter vector's size, but still, we need to map from string to integers. Nevertheless, if we apply feature hashing, it will automatically do the mapping into a hash function; thus, there is no need for mapping strings to integer. Performing text hashing increases efficiency and scalability in the classification of big data analytics.

3.4 Data Classification

The primary objective of the classification step is to train a binary classifier that classifies whether a bug report is accessibility or non-accessibility after learning from the bug report that we have identified as an accessibility bug reports of all the different projects. Data obtained from Bugzilla and Monorail archives for accessibility bug reports was highly imbalanced; precisely, the number of non-accessibility of bug reports is much higher than the accessibility of bug reports. Imbalanced data restricts the standard deviation in the majority of the classification approaches from operating well with the lower classes (i.e., the class containing significantly lower data). There are different machine learning methods designed to solve this issue [21]—for instance, data resampling techniques, gradient boosting techniques (tree-based models), and ensemble techniques. Similar costing methods become computationally costly to sample approaches, such as the Synthetic Minority Over-sampling (SMOTE) [13]. Therefore, the main reason for selecting the ensemble learning techniques in our study is that we can combine several different classification methods to overcome imbalanced data.

3.5 Machine Learning Algorithms

Choosing a suitable classifier that can provide an optimal identification for our study purpose is not a straightforward task [17]. Our study addresses a binary classification (two-class) problems, as our dataset is being classified into two classes, accessibility bug reports, and non-accessibility bug reports. Since we have a dataset already labeled as two classes, our methodology depends on supervised machine learning algorithms to allocate each bug report into one of the defined classes. We evaluated five different machine learning algorithms to observe which one offers the most successful outcomes for the classification of accessibility and bug reports. Specifically, Decision Tree (DT), Random Forest (RF), Decision Jungle (DJ), Support Vector Machine (SVM), and Neural Network (NN). We selected these algorithms because they are widely used in the literature of software defect classification [20, 28, 36, 42, 49], also they are stated to work well with the imbalance datasets, and NLP in literature [19, 25]. To enable replication of our findings, we present the chosen key parameters for the selected machine learning algorithms techniques, as described in the Table 4.

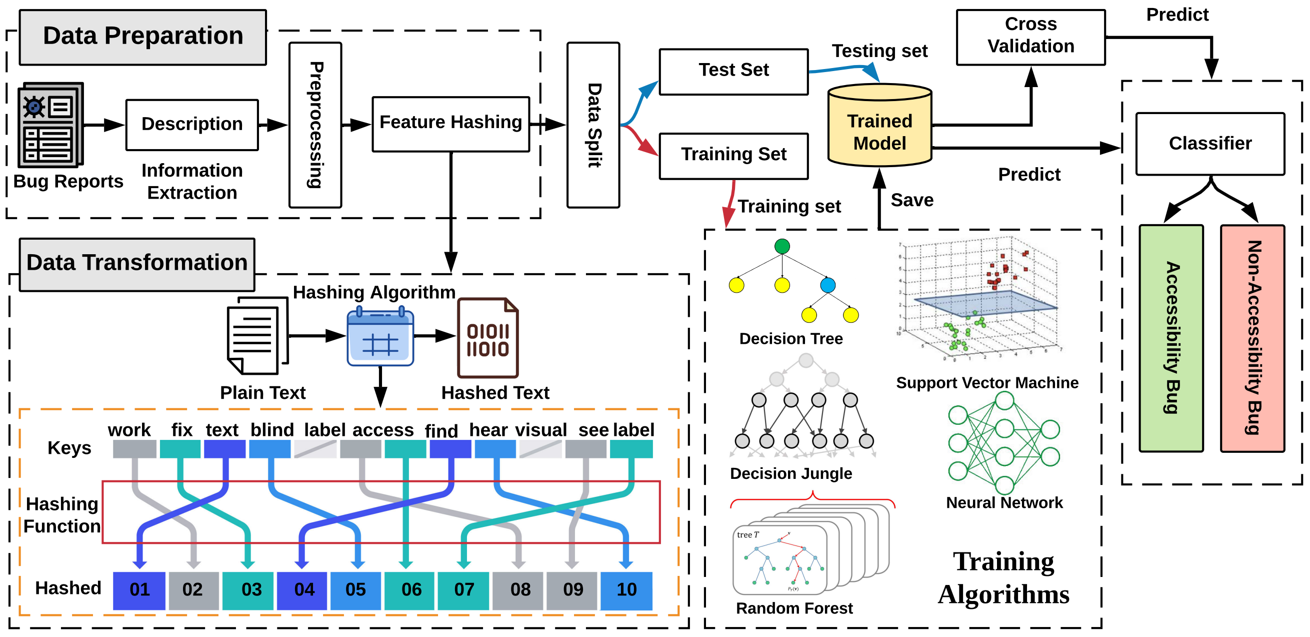

Figure 2: Overview approach of data preparation and data transformation.

3.6 Evaluation Metrics

This study used a 10-fold cross-validation method to train the model to assess the variability and reliability. For individual models, we distributed our dataset in 10 folds of the same size bug reports. Afterward, we performed 10 tests with separate data sets, during which 9 folds were utilized in each assessment as training sets and the remaining fold used as test sets. Then, we evaluate these machine learning models' performance in terms of accuracy, precision, recall, and F_1 score. These evaluation parameters are mostly used in binary classification problems, as in our case [20, 48]. For each evaluation metric, the score rank is between 0.0 and 1.0, where 0.0 represents the classifier's lowest performance, while the 1.0 score represents the classifier's highest performance.

3.6.1 Accuracy provides the score, which shows that how much a classifier is accurate. It can be defined as the total number of correct predictions divided by the total number of predictions. It works well on a balanced dataset.

$$Accuracy = \frac{total\ number\ of\ correct\ predictions}{total\ number\ of\ predictions} \tag{2}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

- **TP** is the classification made by a classifier as "Yes" against an example, and the actual label of the example is also "Yes".
- **TN** is the classification made by a classifier as "No" against an example, and the actual label of the example is also "No".
- **FP** is the classification made by a classifier as "Yes" against an example, but the actual label of the example was "No".
- **FN** is the classification made by a classifier as "No" against an example, but the actual label of the example was "Yes".

3.6.2 Precision is also known as a positive predictive value, which is the fraction of relevant examples among the retrieved examples. It tells us the number of correct positive classifications from the classifier's total number of positive classifications. It can be calculated as:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

3.6.3 Recall is also known as sensitivity. It tells us how many correct positive predictions are made by a classifier from the total number of actual positive predictions. It can be calculated as:

$$Recall = \frac{TP}{TP\ + FN} \tag{5}$$

3.6.4 F-score is also known as the F_1 score or F measure. It is a harmonic mean of precision and recall. It can be calculated as:

$$F - score = 2 * \frac{Precision\ * \ Recall}{Precision\ + Recall} \tag{6}$$

3.6.5 AUC stands for the area under the curve, AUC is a performance measurement for classification problems. It tells us about the successful classification rate of a classifier.

4 Study Results

RQ1: *What is the accuracy of different models in detecting bug reports?*

Approach. In this research question, we double the number of the accessibility bug report for each project to run the experiment of RQ1. For instance, the Firefox platform contains 250 accessibility bug reports (ABR), so we double the number of non-accessibility bug reports (Non-ABR X2) to become 500 bug reports as shown in

Table 4: Summary of the hyperparameter in machine learning algorithm.

Algorithm	Hyperparameter	Default	Description
	max_n_leaf	20	The maximum number of leaves per tree
Decision Tree	min_samples_leaf	10	The minimum number of samples per leaf node
	learning_rate	0.2	Learning rate
	n_tree	100	The number of trees constructed
	n_estimators	8	The number of decision trees
Decision Forest	max_depth	32	The maximum depth of the decision trees
	n_samples_leaf	125	The number of random splits per node
	min_samples_split	1	The minimum number of samples per leaf node
	n_estimators	8	The number of decision directed acyclic graphs
Decision Jungle	max_depth	32	The maximum depth of the decision directed acyclic graphs
	max_width	128	The maximum of the decision directed acyclic graphs
	n_optimiz	2048	The number of optimization steps per decision directed acyclic graphs layer
	n_iter	1	The number of iterations
SVM	Lambda	0.001	The Lambda
	n_nodes	100	The number of hidden nodes
Neural Network	learning_rate	0.1	Learning rate
	n_learning_rate	100	The Number of learning iterations
	learning_rate_weights	0.1	The initial learning weights diameter
	momentum	0	The momentum

Table 5: Distribution of the number of non-accessibility bug reports dataset divided in ten iterations.

Platforms	#ABR	#Non *ABR X2*	#Non *ABR X10*	#Non *ABR X20*	#Non *ABR X30*	#Non *ABR X40*	#Non *ABR X50*	#Non *ABR X60*	#Non *ABR X70*	#Non *ABR X80*	#Non *ABR X90*	#Non *ABR X100*
Firefox	**250**	**500**	**2,500**	**5,000**	**7,500**	**10,000**	**12,500**	**15,000**	**17,500**	**2,000**	**22,500**	**25,000**
Core	**599**	**1,198**	**5,990**	**11,980**	**17,970**	**23,960**	**29,950**	**35,940**	**41,930**	**47,920**	**53,910**	**59,900**
Mac	**307**	**614**	**3,070**	**6,140**	**9,210**	**12,280**	**15,350**	**18,420**	**21,490**	**24,560**	**27,630**	**30,700**
Windows	**442**	**884**	**4,420**	**8,840**	**13,260**	**17,680**	**22,100**	**26,520**	**30,940**	**35,360**	**39,780**	**44,200**
Chrome	**412**	**824**	**4,120**	**8,240**	**12,360**	**16,480**	**20,600**	**24,720**	**28,840**	**32,960**	**37,080**	**41,200**
Android	**347**	**694**	**3,470**	**6,940**	**10,410**	**13,880**	**17,350**	**20,820**	**24,290**	**27,760**	**31,230**	**34,700**
NetBeans	**210**	**420**	**2,100**	**4,200**	**6,300**	**8,400**	**10,500**	**12,600**	**14,700**	**16,800**	**18,900**	**21,000**
Total	**2,567**	**5,134**	**25,670**	**51,340**	**77,010**	**102,680**	**128,350**	**154,020**	**179,690**	**205,360**	**231,030**	**256,700**

the Table 5. Then, we conducted a 10-folds cross-validation [24] procedure to split our data into training data and evaluation data on the five machine learning models. We use cross-validation because the dataset we used in this study is an imbalanced dataset. Therefore cross-validation is a more appropriate approach as compared to the conventional train test split approach. To evaluate our result in RQ1, we used our performance evaluation accuracy, precision, recall, F score, and AUC, which are described in detail in Section 3.6.

Results. The result of all machine learning models show in Table 6. The model performs differently in different scenarios, such as when we apply the machine learning model for the Firefox project bug report classification. Neural networks outperform all other models in accuracy, precision, recall, F-score, and AUC by achieving the 0.91, 0.94, 0.88, 0.90, and 0.97. Decision Tree is the only model with the same accuracy, recall, and F score as a Neural Networks. However, Neural Network achieves high precision and AUC score than all other models; thus, Neural Networks is significant in the case of the Firefox project.

Decision Tree outperforms all other models in terms of all evaluation parameters in the Core project bug report classification. The Decision Tree achieves 0.92 accuracy, precision, recall, F score, and 0.96 ACU score, while SVM performs poorly in this case with a 0.87 accuracy score. Decision Tree also performs well in the Windows project and achieves the 0.89 accuracy score, same as in Core project bug reports classification, and SVM is the worst performer than all other Windows project models.

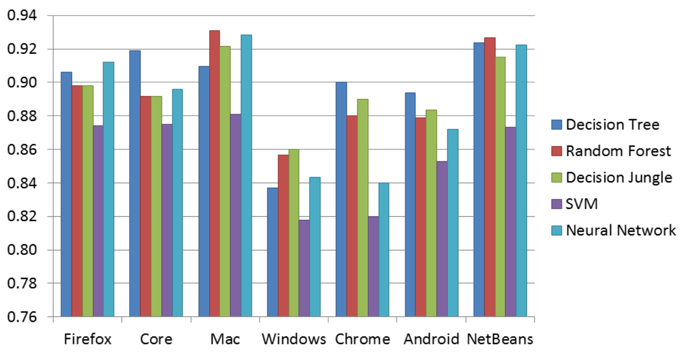

Figure 3: Distribution of classification accuracy metric in all classifiers.

In Netbeans and Android bug reports classification, Random Forest and Neural Networks perform significantly than other models and achieve equal accuracy. In terms of the AUC neural network, lead the table with a 0.92 score. Mac project bug reports classification is the only case where Decision Jungle achieves high accuracy. In this case, Random Forest also achieves the same accuracy as Decision Jungle, so both share their Mac case's significant performance.

Discussion. According to the result, all tree-based ensemble models such as Decision Tree, Random Forest, and Decision Jungle perform better than the linear model SVM, except the case on the Chrome project. The reason for the better performance of the tree-based ensemble model is that when the number of base learners work on a single problem, it performs better than an individual learning model. Random Forest and Decision Jungle are ensemble models that make a final prediction based on their numbers of decision tree predictions using voting criteria. Random Forest is better than Decision Jungle in some cases where data is more imbalanced because Random Forest controls the over-fitting problem on imbalanced data more efficiently [41]. After all, each tree in Random Forest is constructed on a bag, and each bag is a uniform random sample from the original dataset with the replacement of samples, that the reason tree in Random Forest is biased in the same direction and magnitude (on average) by class imbalance.

On the other linear model, SVM shows poor performance in all cases except chrome project in term of the accuracy, as shown in Figure 3 because its kernel trick is not to consider more suitable to boost the performance on the small and imbalanced dataset as compare to a tree-based model that can perform better also on small data size. Neural Network is also performed better in all cases and beats the SVM, where it reaches 0.95 accuracy in the Chrome project. There is no significant difference in the tree-based model and Neural Network model performance. To compare all classifiers results, Random Forest and Decision tree are a more fitting model as compare to others. In the case of Chrome, it achieves the highest accuracy of all this study 0.97.

*RQ*1 Summary

We find that tree-based and Neural Networks classifiers perform better than linear model (SVM) classifier when classifying accessibility bug reports. However, *Decision Tree*'s performance significantly outperforms all other classifiers in terms of evaluation parameters. In terms of projects, NetBeans and Android bug reports are more correctly classified in comparison with other projects.

RQ2: *What is the size of the training dataset needed for the classification to effectively identify accessibility bug reports?*

Approach. This question aims to investigate the size of the dataset needed for the classifiers to classify the accessibility bug reports. To examine this, we performed the RQ2 by incrementally increase the dataset size step by step. We apply this approach to ten iterations. For the first iteration, we randomly selected 10 accessibility bug reports, 100 non-accessibility bug reports. Then we used the Random Forest classifier to examine the outputs of the study experiment. We performed the same approach in the second iteration, but we increased the dataset (double size) as the first iteration. We randomly selected 20 accessibility bug reports and 200 non-accessibility bug reports. We apply this method until we reach the ten iterations with 100 accessibility bug reports and 1000 non-accessibility bug reports. We separately examined each project to find out if different projects needed less or more dataset to classify. For the evaluation parameters of RQ2, we used F1-Score, since accuracy is not considered the best parameters because we have an imbalanced data issue by incrementally increasing each iteration by adding the non-ABR reports.

To assess this RQ2, we performed 10-folds cross-validation techniques. We collected all the results of the F1-Score for all the ten iterations, as shown in Figure 4. When the F1-Scores present stability in the works, we consider the number of accessibility bug reports needed for classification to classify the accessibility bug reports.

Results. The machine learning model performance depends on the size of data and the feature correlation with the target class. In this study, the experiment performs on different project dataset using machine learning algorithms to analyze the impact of the dataset size on model performance. The Figure 4 show the performance random forest on the different project data. As in Figure 4 on the X-axis number show the iterations, we increase the size of data for each class after each iteration. In the first iteration, when we train random forest, the first project dataset contains ten records for the ABR and 100 for non-ABR, and on the second iteration, there are 20 records for ABR and 200 records for non-ABR, and this procedure applied to all of the ten iterations. If we analyze random forest performance in each project, we can see that it is more consistent as we increase the dataset size. Random forest performance evaluates using the F1 score because the ratio of target classes (ABR & non-ABR) is unequal in the dataset. After all, the F1 score can better interpret the machine learning model performance [8].

Table 6: The results of the classifiers.

Project	Classifier	Mean				
		Accuracy	Precision	Recall	F-Score	AUC
Firefox	Decision Tree	0.91	0.92	0.88	0.90	0.95
	Random Forest	0.90	0.94	0.85	0.89	0.96
	Decision Jungle	0.90	0.93	0.86	0.87	0.93
	SVM	0.87	0.88	0.86	0.87	0.93
	Neural Network	0.91	0.94	0.88	0.90	0.97
Core	Decision Tree	0.92	0.92	0.92	0.92	0.96
	Random Forest	0.89	0.90	0.88	0.89	0.96
	Decision Jungle	0.89	0.91	0.87	0.89	0.94
	SVM	0.87	0.87	0.88	0.87	0.94
	Neural Network	0.90	0.90	0.89	0.89	0.95
Mac	Decision Tree	0.84	0.83	0.85	0.84	0.92
	Random Forest	0.86	0.88	0.82	0.85	0.91
	Decision Jungle	0.86	0.92	0.79	0.85	0.91
	SVM	0.82	0.84	0.80	0.81	0.89
	Neural Network	0.84	0.86	0.83	0.84	0.90
Chrome	Decision Tree	0.90	0.90	0.90	0.90	0.95
	Random Forest	0.80	0.89	0.87	0.88	0.94
	Decision Jungle	0.89	0.93	0.84	0.88	0.93
	SVM	0.82	0.84	0.80	0.82	0.90
	Neural Network	0.84	0.85	0.83	0.84	0.92
Windows	Decision Tree	0.89	0.89	0.89	0.89	0.95
	Random Forest	0.88	0.90	0.85	0.87	0.94
	Decision Jungle	0.88	0.92	0.84	0.88	0.94
	SVM	0.85	0.86	0.85	0.85	0.92
	Neural Network	0.87	0.88	0.87	0.87	0.94
Android	Decision Tree	0.92	0.92	0.93	0.92	0.96
	Random Forest	0.93	0.92	0.93	0.93	0.96
	Decision Jungle	0.92	0.93	0.89	0.91	0.95
	SVM	0.87	0.88	0.87	0.87	0.93
	Neural Network	0.92	0.92	0.93	0.92	0.96
NetBeans	Decision Tree	0.91	0.92	0.91	0.91	0.96
	Random Forest	0.93	0.96	0.90	0.93	0.97
	Decision Jungle	0.92	0.94	0.90	0.92	0.96
	SVM	0.88	0.91	0.86	0.88	0.96
	Neural Network	0.93	0.94	0.92	0.93	0.98

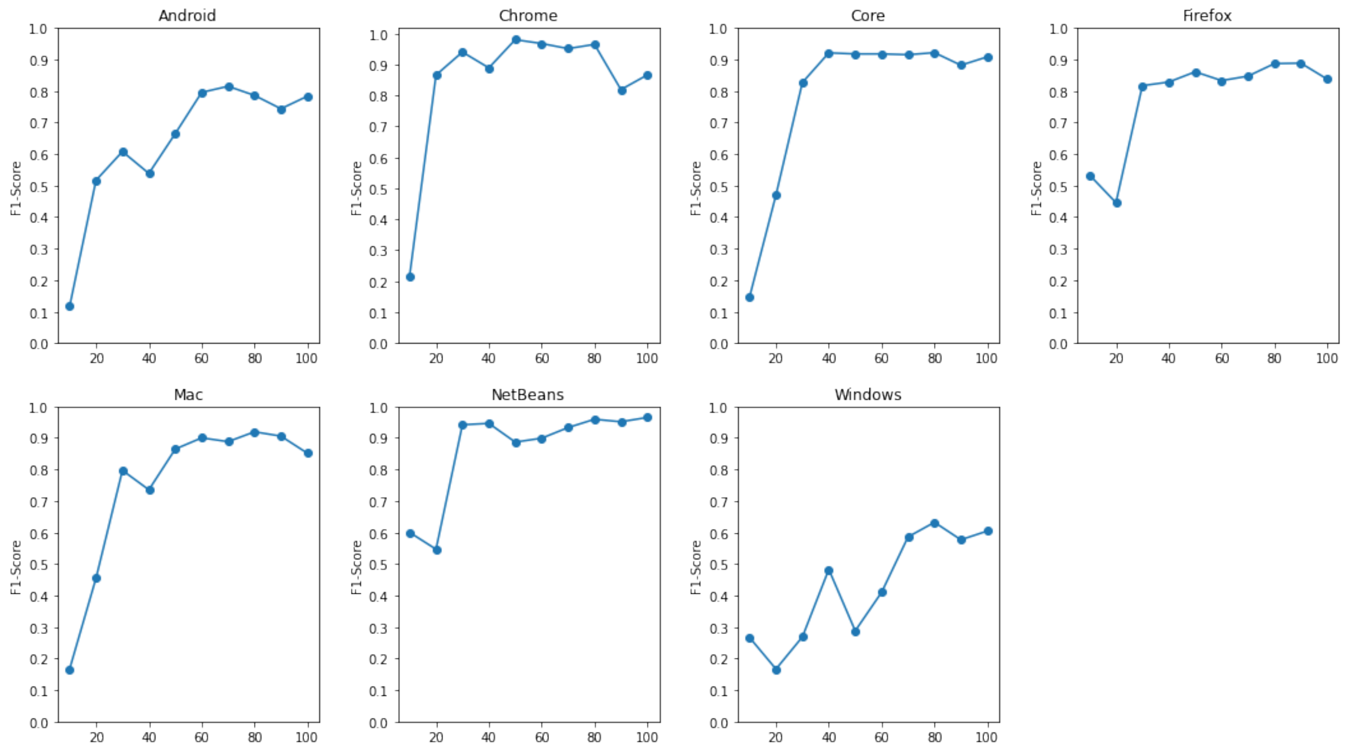

Figure 4: Distribution of classification F1-score in random classifier when incrementally increase accessibility bug reports in ten iterations.

The Android project model performs very poorly in the first iteration when there are only ten records in the dataset, but as the dataset size increases model performs gradually, and after six iterations, it becomes more consistent, as shown in Figure 4. Random forest performance on the Chrome project is different as compared to Android. Random forest performs extraordinary only after one iteration and achieves the 0.87% score, which is the highest score on the second iteration compared to the other models. Random forest perform more accurately in chrome second iteration even on a small dataset, and its reason can be a good correlation between the features and the target class in the chrome dataset. There is little fluctuation in the ninth and 10th iteration in the F1 score on the chrome dataset, as illustrated in Figure 4. Random forest becomes consistent in performance after the fourth iteration in Core and Mac dataset and maintains its consistency until the tenth iteration.

NetBeans and Firefox datasets also contain more better-correlated features for target classes because random forest performs very well on these two projects only after three iterations and becomes a consistent performer after the third iteration. The performance of random forest is different on Windows project data as compare to the others. The Window project dataset model performed very poorly and achieved the highest 0.63% F1 score from all ten iterations, but the model becomes consistent in the score after the seventh iteration, which shows that the model is more accurate when dataset size becomes large.

RQ2 Summary

We find that to achieve a performance equivalent to 93% of the high F-measure score, only one fold of bug reports is required for the training of the binary classifier. In terms of projects, NetBeans and Android bug reports seem to contain the highest number of discriminative keywords, yielding in better accuracy of the classification, in comparison with other projects.

5 Threats to Validity

This section presents various threats to the validity of our study. Threats are divided into three main categories: sampling bias, and external validity.

Sampling Bias. Any classified experiment is challenged because what works in one field may not work in a different one. To generalize our findings, we use several projects in various domains with labeled bug reports. In particular, we performed our classifier on mobile and desktop applications. Another potential concern of bias is the choice of classifiers. Despite the numerous learning algorithms existing, there are still many to consider. Our choice is directed by the objective to manage a reasonable balance between existing and creative techniques. To limit this threat, the subset algorithms selected for this research are based on a comparative analysis of 22 algorithms [26], which indicated that the overall seventeen algorithms' results were not significantly different. We utilized five of the seventeen algorithms. Furthermore, we also chosen algorithms widely used in the literature of the software classification [20, 28, 36, 42, 49], and operate well with the imbalance datasets and NLP in literature [19, 25]. We believe that our study contains the representation of various fields, such as machine learning, topic modeling, and text mining.

External Validity. Our build classifier is trained and evaluated in English-language bug reports. The proposed approach is not sufficient or does not work in other languages for the bug reports. Furthermore, we applied our classifier on open-source systems due to its availability. It could generalize the findings if we can use commercial/industrial projects since the quality level of bug reports varies.

6 Conclusion

In this paper, we tackled the detection of accessibility bug reports as a binary classification problem. We challenged various classifiers using a large set of reports, exported from multiple open-source projects. Our experiments show that the *Decision Tree*'s performance significantly outperforms all other classifiers in terms of evaluation parameters.

In the future, we plan to study the applicability of our approach to other projects developed in different programming languages, and to other domains. Another potential research direction is to use the current findings to build a model that handles the class imbalance problem, in the context where the number of accessibility bug reports becomes a minority class, which hinders the learning of its discriminative features.

References

[1] Afnan A Al-Subaihin, Atheer S Al-Khalifa, and Hend S Al-Khalifa. 2013. Accessibility of mobile web apps by screen readers of touch-based mobile phones. In *International Conference on Mobile Web and Information Systems*. Springer, 35–43.

[2] Wajdi Aljedaani and Yasir Javed. 2018. Bug reports evolution in open source systems. In *5th International Symposium on Data Mining Applications*. Springer, 63–73.

[3] Wajdi Aljedaani, Yasir Javed, and Mamdouh Alenezi. 2020. Lda categorization of security bug reports in chromium projects. In *Proceedings of the 2020 European symposium on software engineering*. 154–161.

[4] Wajdi Aljedaani, Yasir Javed, and Mamdouh Alenezi. 2020. Open Source Systems Bug Reports: Meta-Analysis. In *Proceedings of the 2020 The 3rd International Conference on Big Data and Education*. 43–49.

[5] Wajdi Aljedaani, Meiyappan Nagappan, Bram Adams, and Michael Godfrey. 2019. A comparison of bugs across the iOS and Android platforms of two open source cross platform browser apps. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 76–86.

[6] Wajdi Aljedaani, Furqan Rustam, Stephanie Ludi, Ali Ouni, and Mohamed Wiem Mkaouer. 2021. Learning Sentiment Analysis for Accessibility User Reviews. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 239–246.

[7] Bader Alkhazi, Andrew DiStasi, Wajdi Aljedaani, Hussein Alrubaye, Xin Ye, and Mohamed Wiem Mkaouer. 2020. Learning to rank developers for bug report assignment. *Applied Soft Computing* 95 (2020), 106667.

[8] Eman Abdullah AlOmar, Wajdi Aljedaani, Murtaza Tamjeed, Mohamed Wiem Mkaouer, and Yasmine N El-Glaly. 2021. Finding the needle in a haystack: On the automatic identification of accessibility user reviews. In *Proceedings of the 2021 CHI conference on human factors in computing systems*. 1–15.

[9] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1323–1334.

[10] BBC. 2020. The BBC Standards and Guidelines for Mobile Accessibility. https://www.bbc.co.uk/guidelines/futuremedia/accessibility/mobile.

[11] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An empirical analysis of bug reports and bug fixing in open source android apps. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 133–143.

[12] Bugzilla. 2020. Bugzilla Issue Tracker). https://www.bugzilla.org/.

[13] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[14] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[15] Marcelo Medeiros Eler, Leandro Orlandin, and Alberto Dumont Alves Oliveira. 2019. Do Android app users care about accessibility? an analysis of user reviews on the Google play store. In *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*. 1–11.

[16] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. 2021. On the classification of bug reports to improve bug localization. *Soft Computing* 25, 11 (2021), 7307–7323.

[17] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we need hundreds of classifiers to solve real world classification problems? *The journal of machine learning research* 15, 1 (2014), 3133–3181.

[18] Joseph L Fleiss, Bruce Levin, Myunghee Cho Paik, et al. 1981. The measurement of interrater agreement. *Statistical methods for rates and proportions* 2, 212-236 (1981), 22–23.

[19] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. 2011. A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 4 (2011), 463–484.

[20] Katerina Goseva-Popstojanova and Jacob Tyo. 2018. Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 344–355.

[21] Guo Haixiang, Li Yijing, Jennifer Shang, Gu Mingyun, Huang Yuanyue, and Gong Bing. 2017. Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications* 73 (2017), 220–239.

[22] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*. 309–320.

[23] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. 2016. Using dynamic and contextual features to predict issue lifetime in GitHub projects. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 291–302.

[24] Ron Kohavi et al. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, Vol. 14. Montreal, Canada, 1137–1145.

[25] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 5, 4 (2016), 221–232.

[26] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[27] Stanislav Levin and Amiram Yehudai. 2019. Towards software analytics: Modeling maintenance activities. *arXiv preprint arXiv:1903.04909* (2019).

[28] Andy Liaw, Matthew Wiener, et al. 2002. Classification and regression by randomForest. *R news* 2, 3 (2002), 18–22.

[29] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.

[30] Amiya Kumar Maji, Kangli Hao, Salmin Sultana, and Saurabh Bagchi. 2010. Characterizing failures in mobile oses: A case study with android and symbian. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 249–258.

[31] Norbert Markus, Szabolcs Malik, Zoltan Juhasz, and András Arató. 2012. Accessibility for the blind on an open-source mobile platform. In *International Conference on Computers for Handicapped Persons*. Springer, 599–606.

[32] Stuart McIlroy, Nasir Ali, Hammad Khalid, and Ahmed E Hassan. 2016. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering* 21, 3 (2016), 1067–1106.

[33] Tom Mitchell. 1997. Introduction to machine learning. *Machine Learning* 7 (1997), 2–5.

[34] Monorail. 2020. Monorail Issue Tracker). https://bugs.chromium.org/p/chromium/issues/list.

[35] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. 2017. On-device bug reporting for android applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 215–216.

[36] Moein Owhadi-Kareshk, Sarah Nadi, and Julia Rubin. 2019. Predicting Merge Conflicts in Collaborative Software Development. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–11.

[37] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2015. How can i improve my app? classifying user reviews for software maintenance and evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 281–290.

[38] Fayola Peters, Thein Tun, Yijun Yu, and Bashar Nuseibeh. 2017. Text filtering and ranking for security bug report prediction. *IEEE Transactions on Software Engineering* (2017).

[39] Nasir Safdari, Hussein Alrubaye, Wajdi Aljedaani, Bladimir Baez Baez, Andrew DiStasi, and Mohamed Wiem Mkaouer. 2019. Learning to rank faulty source files for dependent bug reports. In *Big data: learning, analytics, and applications*, Vol. 10989. International Society for Optics and Photonics, 109890B.

[40] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, and SVN Vishwanathan. 2009. Hash kernels for structured data. *The Journal of Machine Learning Research* 10 (2009), 2615–2637.

[41] R Shreyas, DM Akshata, BS Mahanand, B Shagun, and CM Abhishek. 2016. Predicting popularity of online articles using random forest regression. In *2016 Second International Conference on Cognitive Computing and Information Processing (CCIP)*. IEEE, 1–5.

[42] Qasim Umer, Hui Liu, and Yasir Sultan. 2018. Emotion based automated priority prediction for bug reports. *IEEE Access* 6 (2018), 35743–35752.

[43] Solomon Ogbomon Uwagbole, William J Buchanan, and Lu Fan. 2017. Applied machine learning predictive analytics to SQL injection attack detection and prevention. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 1087–1090.

[44] Christopher Vendome, Diana Solano, Santiago Liñán, and Mario Linares-Vásquez. 2019. Can everyone use my app? An Empirical Study on Accessibility in Android Apps. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 41–52.

[45] W3C. 2020. Web Content Accessibility Guidelines (WCAG) 2.1. https://www.w3.org/TR/WCAG21/.

[46] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*. 1113–1120.

[47] William S Yerazunis. 2003. Sparse binary polynomial hashing and the CRM114 discriminator. In *2003 Cambridge Spam Conference Proceedings*, Vol. 1.

[48] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. 2015. Combining software metrics and text features for vulnerable file prediction. In *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 40–49.

[49] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 914–919.