SUMMARY Software products are increasingly complex, so it is becoming more difficult to find and correct bugs in large programs. Software developers rely on bug reports to fix bugs; thus, bug-tracking tools have been introduced to allow developers to upload, manage, and comment on bug reports to guide corrective software maintenance. However, the very high frequency of duplicate bug reports means that the triagers who help software developers in eliminating bugs must allocate large amounts of time and effort to the identification and analysis of these bug reports. In addition, classifying bug reports can help triagers arrange bugs in categories for the fixers who have more experience for resolving historical bugs in the same category. Unfortunately, due to a large number of submitted bug reports every day, the manual classification for these bug reports increases the triagers’ workload. To resolve these problems, in this study, we develop a novel technique for automatic duplicate detection and classification of bug reports, which reduces the time and effort consumed by triagers for bug fixing. Our novel technique uses a support vector machine to check whether a new bug report is a duplicate. The concept profile is also used to classify the bug reports into related categories in a taxonomic tree. Finally, we conduct experiments that demonstrate the feasibility of our proposed approach using bug reports extracted from the large-scale open source project Mozilla.

key words: bug report classification, concept profile, duplicate detection, support vector machine, software maintenance

1. Introduction

Vast numbers of software products are being developed, and their size and complexity are increasing. Bug fixing consumes large amounts of project funding each year and the cost is becoming too high [1]. Thus, software maintenance is an important and challenging task.

Software maintenance activities rely on bug reports to correct defects in the source code files [2]. Bug-tracking systems are used widely to handle the large number of bugs in programs [3]. Many open-source projects, such as Eclipse [4] and Mozilla [5], use bug-tracking systems to track any bugs that arise during software development. For example, Bugzilla [6] is a well-known open source web-based bug-tracking tool, which allows developers to submit, describe, and comment on bug reports at any time. These bug reports are then used to guide corrective software maintenance activities, which help to produce more reliable software systems.

However, most software projects that use bug-tracking tools receive several hundred bug reports each day, but these projects lack the developmental resources to triage every bug report. The large number of bug reports being duplicates has become a serious problem. Duplicate bug reports occur when more than one developer submits a similar description for the same defect [7]. Previous studies find that over 36% of bug reports are duplicates [8]. The identification of duplicate bug reports in large repositories is a burdensome task. In general, the triagers responsible for performing bug triage tasks allocate large periods of time to identifying duplicates. Moreover, classifying the numerous bug reports is a necessary step for bug fixing because it can help triagers assign bugs in a category to the appropriate developers who are more familiar with fixing bugs in the same category. Unfortunately, the manual classification is a time-consuming task for the triagers because of so many submitted bug reports each day. Therefore, it is necessary to develop an algorithm for executing automatic duplicates detection and classification of bug reports.

In the present study, we developed a novel technique using a support vector machine (SVM) [12, 13], a concept profile (CP) [9], and taxonomy [10], which facilitates duplicate detection and the classification of bug reports. A major component of our proposed approach is the use of a smoothed unigram model (UM) and KL-divergence [11] to compute similarity measures between bug reports, which are then used as inputs to train a discriminatory model with the SVM to identify duplicate bug reports. A CP with taxonomy (CPwT) algorithm is used to classify each new bug report into an appropriate category. This new technique is expected to reduce the time and effort allocated by triagers to bug fixing.

We tested our novel method using bug reports extracted from the large-scale open source repository Mozilla. Experiments and analyses demonstrated that our novel classification technique is more effective and accurate than other methods. The most important contributions of this study are as follows:

- An SVM trained a classifier to answer the question: “Is this bug report the duplicate of another bug report?” and this classifier was used to check whether a bug report was a duplicate.
- We developed the CPwT algorithm to classify bug reports in appropriate categories, which allows developers to find related bug reports easily and rapidly.
- We tested the proposed technique using bug reports
collected from Mozilla and demonstrated the superior effectiveness of our novel method compared with other approaches.

The paper is organized as follows. Section 2 reviews research related to duplicate bug report identification and bug report classification. Section 3 presents the data model of bug reports used by our proposed duplicate detection and bug report classification technique. Section 4 explains the detection of duplicate bug reports and their classification using the CPwT algorithm. Section 5 presents the experimental evaluation of our method, while threats to the validity of our approach are discussed in Sect. 6. In the last section, we conclude this study and indicate directions for future research.

2. Related Work

2.1 Duplicate Detection and Classification of Bug Reports

One of the pioneer studies on duplicate bug report identification was performed by Runeson et al. [14], who developed a prototype tool for evaluating and analyzing bug reports. Their experimental results showed that about two-thirds of the duplicates could be found using natural language processing (NLP) techniques. Sureka and Jalote also proposed a method that used a character N-gram-based model for duplicate bug report identification [15]. This approach differed from word-based duplicate bug report identification methods because they investigated the usefulness of low-level features based on characters, which have many advantages such as natural language independence and robustness against noisy data. The empirical results and evaluation showed that this method was effective. Both of these methods are useful for the identification of duplicate bug reports, but they did not classify the bug reports. Therefore, it is not easy to find related bug reports. Our study addressed this problem using a new bug report classification technique, CPwT.

Most techniques used to detect duplicate bug reports only employ natural language information for detection or identification. Wang et al. presented a new method that combined natural language information and execution information [16]. After a new bug report was submitted, its natural language information and execution information were compared with existing bug reports and the most similar bug report was retrieved. This helped the developers to determine whether a new bug report was a duplicate of an existing bug report. Experiments showed that this method could detect 67–93% of duplicate bug reports in the Firefox bug repository. The number of bug reports detected was higher than that using a method based only on natural language information. However, although the execution information helped to improve the accuracy of duplicate bug report detection or identification, the authors think there are costs (e.g., more time-consuming for calculating the execution-information-based similarities) to achieve this effectiveness. Our duplicate detection method does not utilize the execution information in order to reduce the cost, but it can also achieve the better performance as shown in Fig. 9.

Several studies have been performed to classify duplicate bug reports and retrieve appropriate bug reports. For example, Jalbert and Weimer proposed a system for the automatic classification of duplicate bug reports that saved developer time [7]. They used surface features, texture semantics, and graph clustering to identify duplicate bug reports. They found that the inverse document frequency (IDF) was not useful for detecting duplicate bug reports, whereas the title similarity was the most important element during the duplicate bug report classification process. They found that their system reduced the development costs by filtering out 8% of duplicate bug reports. Cunha et al. presented a tool based on information visualization techniques to support developers during the analysis and identification of bug reports [17]. In an experimental evaluation, they simulated a real software development environment and compared the proposed tool with a well-known bug tracker, Bugzilla. The results showed that the visual bug report analysis and search tool was effective. Different from these previous studies, we utilize a machine learning algorithm SVM to check whether a new bug report is duplicate.

Sun et al. utilized SVM to build a discriminatory model or classifier based on a set of labeled vectors [13], where they treated duplicate bug report retrieval as a binary classification problem. When a new bug report was submitted, they classified it as a duplicate or non-duplicate report. They computed 54 types of textual similarities in different reports and used them as features for training and classification purposes. Using different machine learning methods to classify the bug reports, they found that SVM had the best classification accuracy. Bettenburg et al. [18] also used SVM and Naive Bayes to predict whether a bug report needed to be triaged by developers by merging duplicate bug reports. Sun et al. [19] proposed a retrieval function (REP) to identify the duplicate bug reports accurately. In this study, BM25F$\alpha\omega$, an extension version of BM25F, was used to compute the textual similarity between bug reports. Nguyen et al. [20] introduced DBTM, which was a duplicate detection approach that employed IR-based features and topic-based features. Even if we also utilize SVM to train a classifier for discriminating whether the given bug report is duplicate, a major difference is the use of a smoothed UM and KL-divergence to compute similarity measures between bug reports. We adopt them instead of traditional VSM and cosine similarity measure, and use the resulted textual similarity between bug reports as the input of SVM. Discriminatory model and textual similarity measure in REP and DBTM are different from them in our work.

In the present study, we extended our previous work [21], which was a preliminary investigation. The aim of our present study was also duplicate detection and the classification of bug reports, but there are some differences. First, we used smoothed UM instead of a traditional vector space model (VSM) to represent bug reports as vectors and
KL-divergence to compute the similarity measure between the bug reports, which differ from the use of a cosine similarity measure and REP. According to previous studies [11], [22], smoothed UM performs better than VSM, which was used in previous studies. Therefore, the combination of UM and SVM to discriminate duplicate bug reports may deliver greater precision than other methods. Next, we describe the use of CP for classifying bug reports into appropriate categories in a taxonomy tree, which differs from rule-based classification and topic-based models.

2.2 Automatic Bug Triage

The method used to send a bug report to an appropriate developer is the key component of triaging incoming bug reports. Matter et al. proposed an approach that automatically suggested developers with related expertise [23]. This method was executed by comparing the vocabularies in the source code contributions and bug reports of developers. They trained a vocabulary-based expertise model to evaluate the performance of the proposed method. The results showed that this model delivered rates of approximately 33% for top-1 precision and 71% for top-10 recall. Developer feedback was used to improve the performance in this study. Xuan et al. [24] modeled developer prioritization in the Eclipse and Mozilla bug repositories using a social network technique. This approach analyzed the communication among developers based on their comment activities and ranked the developers based on their ability to accomplish three tasks, one of which was bug triage. In our previous study [9], we used CP to find the corresponding bug concept in a new bug report. We also utilized CP to classify bug reports during the clustering process in the present study, but we used smoothed UM and KL-divergence instead of VSM and a cosine similarity measure to calculate the textual similarity between bug reports. In addition, a taxonomy algorithm was used to classify the bug reports into their corresponding categories.

2.3 Other Duplicate Document Detection Problem

Recently, several studies have considered methods for searching documents that are similar to a query document. Jiang and Sun [25] proposed a novel semi-supervised hashing method for high-dimensional data similarity search. This algorithm learned the optimal feature weights based on prior knowledge to retrieve similar data with similar hash codes. Sood and Loguinov [26] proposed a new dimensionality reduction technique for detecting similar document pairs in large-scale collections. They showed that this method delivered 95% recall, which was better than previous studies. Lin et al. [27] used a SVM to learn a discriminant function to determine whether a document was a near-duplicate to the input document based on their shared degree of similarity. Likewise, the researchers including us in software engineering area think that it is great significance to discriminate the duplicate bug reports. Automatic duplicates detection can reduce the triagers’ time and improve the accuracy. Therefore, in our work, we focus on how to utilize the machine learning algorithm (SVM) to train a classifier for verifying the duplicates.

3. Model of Bug Reports

In this section, we present the model of bug reports used by the classification technique. This model describes how the bug reports are structured to explore the new classification technique. Using features extracted from bug reports to build this model, the proposed technique computes the textual similarity between bug reports, discriminates duplicate bug reports using the SVM, and classifies the bug reports using the CPwT algorithm. Figure 1 shows the formal model. This model includes Core, Content, Context, and Category properties. Thus, we refer to the model as the 4C Model [21].

In the 4C Model, a bug report and its id is based on four properties. The Core property represents the basic information in a bug report such as the reporter and the date created. Developer feedback was used to improve the performance in this study. Xuan et al. [24] modeled developer prioritization in the Eclipse and Mozilla bug repositories using a social network technique. This approach analyzed the communication among developers based on their comment activities and ranked the developers based on their ability to accomplish three tasks, one of which was bug triage. In our previous study [9], we used CP to find the corresponding bug concept in a new bug report. We also utilized CP to classify bug reports during the clustering process in the present study, but we used smoothed UM and KL-divergence instead of VSM and a cosine similarity measure to calculate the textual similarity between bug reports. In addition, a taxonomy algorithm was used to classify the bug reports into their corresponding categories.

Fig. 1 4C model for bug report.
4. Duplicate Detection and Classification of Bug Reports

This section describes the approach used to identify duplicate bug reports and to classify new bug reports. First, we outline the execution process used by the proposed technique. Next, we describe the computation of the similarity measure between bug reports. We then explain the duplicate bug report discrimination process based on a SVM. Finally, we present the method used to classify bug reports with the CPwT algorithm.

4.1 Overview

Figure 2 shows the execution process used by the proposed method. There are five steps in the process, as follows.

The first step, extracting bug reports, extracts the bug reports according to the products and components in the taxonomy tree. The second step, pre-processing, applies standard NLP methods such as tokenization, stemming, and stop word removal. The third step, computing the textural similarity between reports, is performed using smoothed UM and KL-divergence. The fourth step, discriminating duplicate bug reports, trains a discriminatory model to classify duplicate and non-duplicate bug reports with an SVM. The fifth step, classifying bug reports with the CP and taxonomy method, allocates the bug reports to their appropriate categories. Finally, the classification results are expected to be provided to the triagers who organize the developers to resolve the bugs. We describe the details of each step in the following subsections.

4.2 Bug Reports Extraction and Pre-Process

We extracted the bug reports from Mozilla for the products and components in the taxonomy tree described in Sect. 4.5.

To compute the similarity measure between bug reports, we pre-process the bug reports using NLP techniques [28]. In general, this process includes tokenization, stemming, and stop word removal [29]. Tokenization is a process where a character stream is parsed into a sequence of word tokens by splitting the stream based on delimiters. Stemming is a process that transforms words to their ground forms. For example, a stemmer can transform “likes” and “liking” to “like.” Stop words are insignificant words in the information retrieval task. These words include pronouns such as “it” and “he,” and link verbs such as “is” and “am.” It is necessary to remove these stop words before determining the textural similarity measure.

After pre-processing the bug reports, we measure the similarity between two bug reports. The title and description represent the major textual information in the bug reports. Therefore, we need to convert the title and description of each bug report into vectors according to the smoothed UM. The fifth step, classifying bug reports with the CP and taxonomy method, allocates the bug reports to their appropriate categories. Finally, the classification results are expected to be provided to the triagers who organize the developers to resolve the bugs. We describe the details of each step in the following subsections.

![Flowchart](image.png)

Fig. 2 The work flow of proposed bug reports classification technique.
Definition 2: Similarity Measure between Bug Reports
\[
sim(\vec{\omega}_a, \vec{\omega}_b) = 1 - KL(p_{\text{uni}}(\omega | \vec{\omega}_a), p_{\text{uni}}(\omega | \vec{\omega}_b))
\]
\[
= 1 - \sum_{i} p_{\text{uni}}(\omega_i | \vec{\omega}_a) \times \log \frac{p_{\text{uni}}(\omega_i | \vec{\omega}_a)}{p_{\text{uni}}(\omega_i | \vec{\omega}_b)}
\]
(2)

where
- \(p_{\text{uni}}(\omega_i | \vec{\omega}_a)\) is the smoothed Unigram representation of bug report \(a\) according to Definition 1. Bug report \(b\) also has the same representation.

4.3 Discriminating Duplicate Bug Reports Using SVM

SVM is a form of supervised learning that analyzes data and recognizes patterns [30]. SVM is based on a set of labeled vectors. Given a set of training patterns, where some are marked as belonging to a positive class and others are marked as belonging to a negative class, SVM builds a discriminatory model that assigns new examples to appropriate categories. In the present study, we used a SVM to classify duplicate and non-duplicate bug reports. This method can be used to classify other unknown data points in vector representations and label them as duplicates or non-duplicates.

During the duplicate detection process, we selected a linear kernel for the SVM because it was efficient and effective for duplicate detection. The linear kernel performed better than other kernel functions such as polynomials and radial basis functions, as described in [31].

Given a set of bug reports classified as duplicates and non-duplicates, this dataset can be used for training to build a discriminatory model or a classifier that answers the question: “Is a bug report a duplicate of another bug report?” Thus, we can determine whether a new bug report is a duplicate or a non-duplicate. This process is performed as follows.

- Creating training patterns: we create the patterns for the duplicate class that includes the pairs \((b_i, D)\), where \(b_i\) represents a master bug report (the oldest bug report submitted to the repository) and \(D\) represents a set of duplicate bug reports. We also create patterns for the non-duplicate class that includes the pairs \((b_i, ND)\), where \(ND\) stands for a set of non-duplicate bug reports.
- Feature extraction: we compute the similarity between the bug reports according to Definition 2, and use the value as a feature. The feature represents a triple \((b_i, b_j, S_{im_{ij}})\).
- Training model: before training, all of the feature values are normalized to set within the range \([-1, 1]\) to avoid the case where some features in a higher range dominate those in a smaller range. The same normalization is also applied when the model is used to classify unknown bug reports. In our study, we used a linear kernel for the SVM when executing the classification process. This is because we were interested in determining the likelihood that two bug reports are duplicates. To achieve this goal, we use the probability estimation functionality of the SVM to train a discriminatory model that classifies duplicate and non-duplicate bug reports.

When a new bug report arrives, we utilize the trained model to discriminate the bug report. The details of the algorithm are shown in Fig. 3.

In our approach, we divide all of the bug reports into sets that include a master bug report and related duplicates, where a master bug report (the oldest report) represents a distinct defect described by both of the reports. Thus, each duplicate must have a corresponding master. After a new bug report is submitted to the Duplicate Discriminating Algorithm (DDA), it returns a Boolean variable that discriminates whether the bug report is a duplicate or a non-duplicate. First, this algorithm moves through all the pairs between \(b_{\text{new}}\) and all of the reports in set \(B_i\). According to Definition 2, we compute the similarity between the new bug report and all of the reports in \(B_i\) to find the highest similarity value. In this algorithm, the discriminative model is used to predict the probability of the new bug report being a duplicate. SVM Predict is an expression used by our discriminative model. In Line 5 in Fig. 3, SVM Predict returns the probability value. In Line 6, the highest value of the current maximum and the probability is returned. If the value is

```
Duplicates Discriminating Algorithm (DDA)
Begin
Input:
\(b_{\text{new}}\): new bug report
\(\theta\): threshold value
\(B_i\): \((b_{i_1}, b_{i_2}, ..., b_{i_n})\) where \(B_i, (1 \leq i \leq n)\) includes a set of duplicate bug reports or a non-duplicate bug report.
Output:
duplicate: true/false where true means that a new bug report is a duplicate and false means that it is not a duplicate.

Body:
1: duplicate = false;
2: max = 0; the initial probability of which the reports are duplicates.
3: foreach \(B_i\), 1 \leq i \leq n,
4: find the highest similarity s between \(b_{\text{new}}\) and any bug report in \(B_i\).
5: \(p = \text{SVM Predict}(s)\); return a probability of the pair of bug reports in \(B_i\) being duplicate by using SVM.
6: max = MAX (max, p); find the current maximum probability.
7: if (max > \(\theta\))
8: duplicate = true;
9: \(B_i = B_i \cup b_{\text{new}}\);
10: break;
11: end
12: if (duplicate == false)
13: make a new set \(B_{\text{new}}\) with \(b_{\text{new}}\).
14: add \(B_{\text{new}}\) to \(B_i\).
End

Fig. 3 Duplicate discriminating algorithm.
```
more than a pre-defined threshold $\theta$, duplicate is set to true and $b_{new}$ is added to $B_i$. This means that the new bug report is a duplicate report in $B_i$. Otherwise, the bug report is not a duplicate and it is added to a new report set $B_{n+1}$ as a master report.

4.4 Classifying Bug Reports Using CPwT

In our previous study [9], we demonstrated that CP is an effective method for classifying bug reports. The definition of CP is presented as follows.

Definition 3: Concept Profile (CP)

A CP is formed by a pair $(C, R)$ where $C$ denotes a bug concept with the topic terms, and $R$ denotes a set of clustered bug reports related to the bug concept.

Building the CP comprises two steps: clustering bug reports and extracting topic terms. We use the k-means clustering algorithm [32] to cluster the bug reports in the training set into appropriate categories. In this process, we define the similarity measure between bug reports as the distance between data points. We refer to this category as a bug concept. After clustering the bug reports in the training set, we extract the topic terms that appear at a high frequency in the bug reports, which belong to the same bug concept. We convert the frequency to a weight value by normalization and select the topic terms for each bug concept according to the threshold of weight values. We decide this threshold according to the result of performance evaluation using the different threshold values, which is described in Sect. 5.2.

Figure 4 shows examples of CPs. Each CP includes a bug concept (e.g., $C_1$ and $C_{11}$) with related topic terms and the bug reports associated with the corresponding bug concept. We assume that the threshold of weight values is 0.07, so we extract 6 topic terms for $C_1$, and 4 topic terms for $C_{11}$, respectively. For example, bug concept $C_1$ has the following 6 related topic terms: ‘unable’, ‘not’, ‘no’, ‘access’, ‘screen’ and ‘reader’. The corresponding weight values are 0.08539, 0.08237, 0.07954, 0.07217, 0.07089 and 0.07089, respectively. We note that ‘screen’ and ‘screen’ have the same weight values, it means that the two topic terms may co-occur in the bug reports in $C_1$. The bug concept $C_1$ and the set of clustered bug reports (e.g., $B_{1-1}$) comprise a CP. The total frequency is shown in the second column of the right sub-table, which indicates the overall frequency of the topic terms that occur in a bug report. This helps us to verify the appropriate category for a new bug report. Next, we describe how this process was implemented.

We introduce a taxonomy algorithm that allocates a newly submitted bug report to an appropriate class. A taxonomy [33], [34], or taxonomy scheme, is a specific type of classification where the elements are arranged in a hierarchical tree. Thus, we collected bug reports with common domains (e.g., client software and server software), products (e.g., Firefox and Mozilla services) and components (e.g., disability access and search) from the Mozilla bug repository. We clustered the bug reports in the training set and extracted the topic terms to build the CPs. Using these CPs, we allocated the reports to their corresponding categories. Figure 5 shows the taxonomy tree, which includes 11 bug types.

**Taxonomic Classification for Bug Types**

- **Mozilla Bugs**
  - **Client Software**
    - **Disability Access**
      - Type-1: ‘not’, ‘access’, ‘unable’, ‘no’, ‘screen’, ‘reader’
    - **Search**
      - Type-2: ‘search’, ‘engine’, ‘plugin’, ‘searchbar’, ‘update’
  - **Mozilla Services(HTML, XML)**
    - Firefox Sync: UI
      - Type-3: ‘weave’, ‘Sync’, ‘bar’, ‘UI’
  - **Components**
    - Core(C++)
      - Security
        - Type-4: ‘threat’, ‘link’, ‘warning’, ‘failure’
      - Networking
        - Type-5: ‘localhost’, ‘http’, ‘URL’, ‘protocol’, ‘proxy’, ‘DNS’
    - Testing
      - Type-6: ‘test’, ‘controller’, ‘remove’, ‘error’, ‘fail’, ‘run’
  - **Server Software**
    - Bugzilla(Perl)
      - Database
        - Type-7: ‘MySQL’, ‘Oracle’, ‘PostgreSQL’, ‘table’
    - User Interface
      - Type-8: ‘error’, ‘button’, ‘page’, ‘link’, ‘bug’
  - **Webtools(XUL)**
    - Graph Server
      - Type-9: ‘graph’, ‘server’, ‘graphserver’
  - **Graveyard(C+++, XUL, Javascript)**
    - Core Graveyard
      - **Embedding: ActiveX Wrapper**
        - Type-10: ‘ActiveX’, ‘plugin’, ‘embed’
    - Toolkit Graveyard
      - Data Collection/Metrics
        - Type-II: ‘metric’, ‘date’, ‘collect’, ‘event’

- **Others**
types. In our study, a bug type refers to a bug concept in the CPs. For example, the bug concept $C_1$ in Fig. 4 is also denoted by the bug type $Type = 1$ in Fig. 5. The corresponding domains, products, and components also appear in the taxonomy scheme. Specifically, we mark the program language with respect to the relevant products. For example, the product ‘Firefox’ has been developed using C++ and Javascript.

For each bug report in the testing set, we need to classify them to their corresponding category. First, we calculate the sum frequency of the topic terms for each bug type in the given bug report. If we detect the highest frequency for a bug type, we confirm that the given bug report belongs to that bug type. For example, for a given bug report $b_{new}$, we assume that the number of occurrences of the topic terms ‘metric’, ‘collect’, ‘data’ and ‘event’ related to $Type = 11$ are 14, 17, 28 and 25 respectively. Thus, the total frequency of these terms appearing in $b_{new}$ is as follows: $84 (14 + 17 + 28 + 25 = 84)$. If the total frequency is the highest in all bug types, $b_{new}$ should be allocated to the category ‘Disability Access’ (Fig. 5). Moreover, there are two special cases in this process, as follows.

- If the total frequency of the topic terms related to a bug type in the given bug report is the same as one or more different bug types and higher than others, we can compute the total frequency of the $(k + n)$ topic terms associated with the corresponding bug types until we obtain the highest. For example, we assume that the total frequency of the 6 topic terms related to $Type = 1$ occurring in the given bug report $b_{new}$ is the same as the total frequency of the 4 topic terms associated with $Type = 11$ and higher than the other bug types. However, if we find that the total frequency of 7 $(6 + 1)$ topic terms associated with $Type = 1$ in $b_{new}$ is higher than the total frequency of 5 $(4 + 1)$ topic terms related to $Type = 11$, we consider that $b_{new}$ belongs to $Type = 1$.

- If the total frequency of the topic terms that appear in the given bug report for each bug type is 0, we allocate the bug report to a special category ‘Others,’ as shown in Fig. 5.

In our study, a bug report and its duplicates (or master) are allocated to the same bug type in the taxonomy tree. This new technique can check whether a newly submitted bug report is a duplicate and allocate it to an appropriate category. This technique should help to reduce the workload for triagers and improve the efficiency of bug triage.

5. Experiment

In this section, we describe how we implemented our proposed bug report classification technique and demonstrate its practical feasibility. First, we collected bug reports from the Mozilla project, which is a large-scale open source project. Next, we performed duplicate discrimination using the given bug reports in the test set via the SVM and compared the detection results with those obtained using previous methods. Finally, we classified the bug reports in the test set and obtained the evaluation results.

5.1 Experiment Setup

We collected bug reports related to different domains, products, and components that appeared in the taxonomy tree of the Mozilla project (https://bugzilla.mozilla.org). Table 1 shows the details of our data set.

In our experiment, 6,787 bug reports including 641 duplicates are collected from the Mozilla project. Due to the fewer number of bug reports in some product/components (e.g., “Data Collection/Metrics”), we used 10-fold cross validation over our data set in order to guarantee the appropriateness of our experiment results. In detail, we divided the bug reports in our data set into 10 folds equally. Thus, at each cross validation step, we have 9 folds of bug reports as the training set and one remaining fold of bug reports as the testing set. This process finishes when every fold has become a testing set one by one. We averaged the evaluation results of all turns as the final result for reducing the error. For the duplicates detection, each of 9 folds includes 678 bug reports and the last one includes 685 bug reports; for the bug reports classification, to analyze the influence of different size of data in each product/component, we divided the bug reports belonging to each product or component into 10 folds, respectively. For example, for a component “Networking”, each of 9 folds includes 114 bug reports and the last one includes 117 bug reports; while dividing the bug reports in “Security”, each of 9 folds includes 25 reports and the last one include 26 reports.

Figure 6 shows the training and testing process using our proposed technique. After applying pre-processing and vector conversion using smoothed UM to all of the bug reports in the training and testing sets, we determined the textual similarity between each pair of bug reports based on the KL-divergence. Next, we used the bug reports in the training set to build a discriminatory model with the SVM. This model determined whether a test bug report was a duplicate. The bug reports in the training set were also used to build CPs and the CPs were arranged in a taxonomy tree. Our proposed algorithm allocated each new test bug report to an appropriate category.

In order to demonstrate that the proposed technique can effectively execute duplicate detection and classification of bug reports, we need to evaluating the performance of previous studies and our method. These evaluation methods include F measure [35] and statistical test. F measure is defined as follows.

**Definition 4: F Measure**

\[
\text{Precision}(q) = \frac{TP}{TP + FP} \quad (3)
\]

\[
\text{Recall}(q) = \frac{TP}{TP + FN} \quad (4)
\]

\[
F(q) = 2 \times \frac{\text{Precision}(q) \times \text{Recall}(q)}{\text{Precision}(q) + \text{Recall}(q)} \quad (5)
\]
Table 1 Summary of our data set.

| Product/Component       | # of Bug reports | # of Duplicates | Period          |
|-------------------------|------------------|-----------------|-----------------|
| Disability Access       | 176              | 23              | 2004/08/19–2012/08/07 |
| Search                  | 304              | 55              | 2003/05/10–2013/05/23 |
| Firefox Sync: UI        | 240              | 33              | 2008/07/01–2013/02/08 |
| Security                | 251              | 27              | 1999/05/27–2013/05/13 |
| Networking              | 1,143            | 178             | 1999/08/13–2013/06/02 |
| Testing                 | 3,491            | 191             | 2006/06/27–2013/06/06 |
| Database                | 176              | 18              | 2001/09/04–2013/03/19 |
| User Interface          | 724              | 100             | 1999/08/14–2013/05/24 |
| Graph Server            | 128              | 8               | 2006/08/15–2013/04/16 |
|_embedding: ActiveX Wrapper | 81              | 5               | 1999/07/01–2010/08/30 |
| Data Collection/Metrics | 73               | 3               | 2006/02/10–2010/12/04 |
| All                     | 6,787            | 641             |                 |

Fig. 6 Training and testing process of the classification technique.

where

- **TP** indicates the true positive instances. For duplicates detection, these instances are the number of bug reports identified as duplicates correctly; for bug reports classification, these instances are the number of bug reports classified correctly.
- **FP** represents the false positive instances. For duplicates detection, these instances are the number of bug reports identified as duplicates incorrectly; for bug reports classification, these instances are the number of bug reports classified incorrectly.
- **FN** denotes the false negative instances. For duplicates detection, these instances are the number of actual duplicate bug reports, which are not detected as duplicate using the duplicates detection algorithms; for bug reports classification, these instances are the number of actual bug reports in the corresponding category, which are not allocated to the correct class using the classification methods.

Statistical test can help us understand whether there is a statistically significant difference between two results (e.g., CPwT and LDA) that we want to compare. We use the Wilcoxon signed-rank test [36] for this statistical test since it can be applied on pairs of data, and is able to compare the difference against zero. At the 95% confidence level, **p-value** that is smaller than 0.05 indicates that the difference between two algorithms is statistically significant. Otherwise, **p-value** that is 0.05 or larger indicates that the difference is not statistically significant.

5.2 Parameter Settings

To conduct the experiments for evaluating our approach, we need to decide on the values of the parameters in this study. For duplicates detection, these parameters include the weight factor of smoothed UM \( \mu \) (Definition 1) and the threshold of SVM Predict \( \theta \) showed in Fig. 3. Because the different values of \( \mu \) and \( \theta \) may affect the performance of duplicates detection, we utilize F measure to evaluate the influence on the performance. Table 2 and Fig. 7 show the evaluation result when selecting the different values of \( \mu \) and \( \theta \). In Table 2, we note that when the value of \( \mu \) is less than or equal to 0.4, the values of F measure are the highest. When it is more than 0.4, the F-measure values begin to decrease. Moreover, when the value of \( \theta \) is set to 0.85, the values of F measure are higher than others. We can also observe this result in Fig. 7. For this figure, the data on the X-axis represent the precision rate, and the data on Y-axis denote the
Table 2  F measure for duplicates detection when selecting the different values of parameters.

| F measure (%) | θ = 0.70 | θ = 0.75 | θ = 0.80 | θ = 0.85 | θ = 0.90 |
|---------------|----------|----------|----------|----------|----------|
| μ ≤ 0.4       | 77.0     | 84.3     | 88.9     | **91.2** | 89.9     |
| μ = 0.5       | 69.4     | 70.5     | 73.1     | 76.9     | 72.7     |
| μ = 0.6       | 59.0     | 63.3     | 65.9     | 67.8     | 65.4     |
| μ = 0.7       | 51.2     | 53.4     | 55.8     | 58.1     | 54.6     |
| μ = 0.8       | 40.4     | 44.5     | 46.3     | 48.3     | 46.3     |
| μ = 0.9       | 29.2     | 31.1     | 32.1     | 34.6     | 33.4     |

Fig. 7  Performance influence in different values of parameters.

Table 3  Performance evaluation for bug reports classification when selecting the different threshold values.

| Threshold | Precision Rate (%) (Weighted Mean) | Recall Rate (%) (Weighted Mean) | F Measure (%) (Weighted Mean) |
|-----------|------------------------------------|---------------------------------|-------------------------------|
| θw = 0.060| 82.7                               | 81.9                            | 82.3                          |
| θw = 0.065| 90.2                               | 89.3                            | 89.7                          |
| θw = 0.070| 93.2                               | 93.2                            | **93.2**                      |
| θw = 0.075| 88.1                               | 87.2                            | 87.6                          |
| θw = 0.080| 84.2                               | 80.6                            | 82.4                          |

recall rate. The six curves stand for the precision-recall rate when selecting the different values of μ. The data points on each curve represent the precision-recall rate when choosing the different values of θ. The values of μ and θ in Fig. 7 are corresponding to Table 2. It can be seen that the values of precision and recall rate are higher than others when μ ≤ 0.4. When the value of θ is more than 0.85, the graphs change sharply. It shows that the recall rate gets worse. In this situation, even if the precision is improved, the F measure gets worse. The major reason is that a small number of bug reports are identified as duplicates due to the higher threshold. According to this evaluation result, we set the weight vector of smoothed UM μ to 0.4 and the threshold of SVM Predict θ to 0.85 in the following experiments.

For bug reports classification, it is necessary to predefine the threshold of weight values for determining the number of topic terms k in each bug concept. We also adopt F measure to evaluate the performance of our classification technique when selecting the different threshold values. Table 3 shows the weighted mean of precision, recall and F measure when the threshold θw changes from 0.060 to 0.080 (the step is 0.005). We adopt weighted mean instead of arithmetic mean. In the performance evaluation for the proposed classification technique, arithmetic mean is not a fair due to the different size of data fold for each product or component. For example, in a component “Networking”, there are 114 bug reports in each fold averagely. Its contribution is much better than “Data Collection/Metrics” which includes only 7 bug reports in each fold averagely. The weighted mean is denoted by \( \frac{1}{n} \sum_{i=1}^{n} w_i \), where \( w_i \) means the ratio of the average number of bug reports in each fold for each product or component to the total number of testing bug reports in all 11 products and components, and \( x_i \) represents the precision, recall and F measure for each product or component respectively. It can be seen that when the threshold is set to 0.070, the weighted mean of precision, recall and F measure is higher than others. It is easy to understand, if θw is set to a higher value, the number of topic terms is reduced, it is not enough to verify the class a new bug report belongs. Otherwise, if θw is set to a lower value, the number of topic terms is increased, some unimportant terms may impact the performance of classification. Therefore, we use 0.070 as the threshold of weight values for deciding the number of topic terms in our classification technique.

5.3 Evaluation Result

5.3.1 Duplicate Bug Reports Discrimination

We use the SVM to train the discriminatory model to find duplicate bug reports in the testing set. In the first experiment, we evaluated the performance of this discriminatory model with the test bug reports. We used F measure to compare the performance of the SVM with Naive Bayes [37]. We also evaluate two different similarity measure methods (smoothed UM-KL divergence and VSM-cosine similarity) to validate why smoothed UM combined with KL-divergence was selected to measure the textual similarity in our study.

Naive Bayes is a supervised classifier like SVM, but it is a probabilistic classifier based on Bayes’ theorem with strong independence. In this experiment, we used 10-cross validation described in Sect. 5.1 to train and test our data set with SVM and Naive Bayes. Figure 8 shows the precision rate, recall rate and F measure for SVM (\( SVM(U−KL) \), \( SVM(V−Cos) \)), Naive Bayes (\( NB(U−KL) \) and \( NB(V−Cos) \)), where \( U−KL \) means that the classifiers used smoothed UM and KL divergence as the textual similarity measure, whereas \( V−Cos \) indicates that the classifiers used VSM and cosine similarity to compute the textual similarity between bug reports.

Figure 8 shows clearly that SVM outperformed Naive Bayes based on its higher precision rate (89.3% for \( U−KL \) and 81.7% for \( V−Cos \)), recall rate (93.1% for \( U−KL \) and 88.3% for \( V−Cos \)) and F measure (91.2% for \( U−KL \) and 84.9% for \( V−Cos \)). In addition, \( U−KL \) performed better than traditional \( V−Cos \), i.e., \( SVM(U−KL) \) shows the higher precision rate, recall rate and F measure than \( SVM(V−Cos) \).
We compared the performance of two different classification models, i.e., CPwT and Latent Dirichlet Allocation (LDA) \cite{38}, in the evaluation experiment. In LDA, each document is assumed to be a mixture of a small number of topics and each topic is created using a bag of words. As a topic-based model, LDA extracts N topics to determine the distributions of topics in each document. If the research objective is a set of bug reports, LDA considers each bug report as a document that comprises words after pre-processing the bug report. Each topic is represented as the probabilities of words that occur in the bug report. In this evaluation experiment, we computed the probabilities of words in each test bug report and selected the topic with the most similar probability distribution. CPwT differs from LDA because the topic terms are extracted after clustering the bug reports. Therefore, an advantage of the CPwT model is that it reduces the size of the word corpus for each bug type and improves the accuracy of classification. To ensure a fair comparison, we set the number of LDA topics to 11, which was equal to the number of bug types in the CPwT model. Table 4 shows the comparison of the two different classification models.

In Table 4, when the size of the testing set was large, CPwT performed much better than LDA. For example, for the product “Testing”, The F-measure was 96% for CPwT, which was much higher than that for LDA (80%). Otherwise, the experiment produced different results. For example, for the components “Embedding: ActiveX wrapper” and “Data collection/Metrics,” the precision, recall, and F-measure were higher with LDA than CPwT because of the smaller size of the testing set (only seven and eight testing bug reports respectively). We can easily observe this result in Fig. 10. For this figure, the numbers from 1 to 10 on the X-axis represent the ranking of the size for data fold in each product/component. For example, ’1’ stands for the minimum size of the data fold (7 bug reports per fold averagely in “Data collection/Metrics”); ’10’ denotes the maximum size of the data fold (349 bug reports per fold averagely in “Testing”). From the values of F measure on the Y-axis, it can be seen that LDA is better than CPwT when the size of the data fold is very small. However, with the increasing of the size of the data fold, CPwT shows the much better performance than LDA. We explain the reason in Sect. 5.4. According to the weighted Mean of Precision, Recall and F measure showed in the last row of Table 4, we can get a conclusion, CPwT is performed better than LDA, especially on the large scale projects.

To verify whether the difference between CPwT and LDA is statistically significant, we use the Wilcoxon signed-rank test for this statistical test. We formulated the null hypothesis ($H_{10}$) and alternative hypothesis ($H_{1_a}$) are as follows:

- $H_{10}$: The classification performance of CPwT and LDA have no acceptability difference from each other.
- $H_{1_a}$: The classification performance of CPwT is more acceptable than LDA.
Table 4 Performance comparison of CPwT and LDA.

| Product/Component       | CPwT      | LDA       |
|-------------------------|-----------|-----------|
|                         | Precision | Recall    | F-Measure | Precision | Recall    | F-Measure |
| Disability Access       | 84.2%     | 88.9%     | 86.5%     | 66.7%     | 77.8%     | 71.8%     |
| Search                  | 91.5%     | 90.0%     | 90.7%     | 75.8%     | 83.3%     | 79.4%     |
| Firefox Sync: UI        | 84.6%     | 91.7%     | 88.0%     | 77.8%     | 87.5%     | 82.4%     |
| Security                | 85.2%     | 92.0%     | 88.5%     | 81.5%     | 88.0%     | 84.6%     |
| Networking              | 95.5%     | 93.9%     | 94.7%     | 74.8%     | 83.3%     | 78.8%     |
| Testing                 | 96.3%     | 95.7%     | 96.0%     | 78.1%     | 81.9%     | 80.0%     |
| Database                | 78.1%     | 88.9%     | 83.2%     | 71.4%     | 83.3%     | 76.9%     |
| User Interface          | 96.3%     | 88.9%     | 92.5%     | 70.5%     | 76.4%     | 73.3%     |
| Graph Server            | 72.8%     | 92.3%     | 81.4%     | 73.3%     | 84.6%     | 78.5%     |
| Embedding: ActiveX Wrapper | 66.7%  | 75.0%     | 70.6%     | 82.7%     | 87.5%     | 85.0%     |
| Data Collection/Metrics | 62.5%     | 71.4%     | 66.7%     | 80.4%     | 85.7%     | 83.0%     |
| Weighted Mean           | 93.2%     | 93.2%     | 93.2%     | 76.3%     | 82.1%     | 79.1%     |

Table 5 Result of the Wilcoxon signed-rank test.

| null hypothesis | p-value  | alternative hypothesis |
|-----------------|----------|------------------------|
| \(H_{0}\)       | p = 0.0336 | \(H_{1a}\): Accept    |

We run CPwT and LDA on the bug reports in the testing set so that these bug reports are arranged into the appropriate categories. Then we compute F measure (Table 4) as a feature for executing the Wilcoxon signed-rank test. This result is given in Table 5.

In Table 5, it can be seen that the p-value of \(H_{10}\) is 0.0336 which is smaller than 0.05 (95% confidence level), therefore, we reject the null hypothesis \(H_{10}\) and accept the alternative hypothesis \(H_{1a}\). This result indicates that the difference between CPwT and LDA is statistically significant. By executing the performance comparison (Table 4 and Fig. 10) and statistical test (Table 5) between CPwT and LDA, we can draw a conclusion: CPwT performed better than LDA.

5.4 Discussion

Duplicate detection and the classification of bug reports are major problems in software maintenance, so many duplicate detection and classification methods have been developed to enhance the performance of maintenance activities. The evaluation experiments conducted in the present study showed that our technique delivered better results than previous studies. We express the possible reasons for the following questions.

Q1: What is the major reason that SVM(U-KL) performs better than other duplicates detection methods?

In our study, we utilized smoothed UM instead of traditional VSM as the representation of bug reports. As a statistical language model, smoothed UM is different from VSM which represents the bug reports as numeric vectors. It transforms the bug reports into the probability vectors. Using the probability vectors as the representation of bug reports improves the accuracy of textual similarity measures. As an evidence, S. Rao and A. Kak in [11] demonstrated that smoothed UM performed better than VSM when executing the Information retrieval-based algorithm in bug localization. Therefore, smoothed UM improved the quality of features (textual similarity between bug reports) used to train the discriminatory model via SVM for detecting the duplicate bug reports. We believe this is the major reason why SVM(U-KL) performs better than other duplicates detection methods which used VSM to represent the bug reports.

Q2: Why the performance of CPwT classification technique is better than the performance of LDA when the data set is bigger?

LDA considers each bug report as a document that comprises words. Due to the probabilities of these words occurring in the bug report, the bug report arranges into one or more topics. CPwT is different from LDA in that the topics are extracted after clustering the bug reports. Therefore, it can produce the higher accuracy of classification than LDA which does not cluster the bug reports. However, if the size of data set is so small, the advantage of CPwT is not obvious. In Table 4, we note that LDA performed better than CPwT when only 7 bug reports are included in the testing sets for “Embedding: ActiveX Wrapper” and “Data Collect-
tion/Metrics.” The main explanation for this is the small size of the testing set, which reduced the effectiveness of clustering using CPwT. As a result, the classification performance of CPwT is better than that of LDA when the data set is bigger.

Based on the result of our experiments, it is expected that this technique can be applied to bug report management using large-scale bug repositories, where it should reduce the workload for triagers and improve the efficiency of bug fixing.

6. Threats to Validity

This section considers the threats to the validity of our approach.

Lack of Generalization: In our study, we captured bug reports from Mozilla to test the proposed technique. Thus, we cannot claim that our results can be generalized to bug reports from other projects. In addition, we only validated our technique using open source projects. Thus, our method might not perform as well with commercial software. However, we consider that our proposed duplicate detection and bug report classification technique could be applied to other projects. The threat to generalization is lower compared with rule-based bug classification techniques [21] because we did not need to pre-define bug rules and specific bug types for bug reports in the target bug repositories. Depending on the specific products and components used in each project, the bug types (bug concepts) and topic terms can be produced automatically using the CP approach. In the future, we will verify the effectiveness of this technique using other open source and commercial projects.

Limited Categories: We collected 6,787 bug reports from the major products and components in Mozilla and organized them into 11 categories based on the different bug types, but the number of categories was not sufficiently large. In the future, we will extend the number of bug types to classify more bug reports.

Empirical Evaluation: We analyzed the evaluation results and explained why our detection and classification methods outperformed previous approaches. However, other external factors (e.g., the quality of bug reports and the selected product/component) may affect the experimental results. In our future studies, we will capture other features that affect the results during duplicate detection and classification so we can improve the current algorithm.

7. Conclusion

In this study, we developed a new technique for detecting duplicates and for classifying new bug reports. We used smoothed UM and KL-divergence to compute the textual similarity between bug reports, and built a discriminatory model based on an SVM to determine whether a new bug report was a duplicate. Moreover, we used CPwT to categorize bug reports to their corresponding categories. The evaluation results showed that our novel technique delivered superior performance in duplicate detection and bug report classification compared with other methods.

In the future, we plan to extend the number of bug types to classify more bug reports and to evaluate our technique using other open source projects. In addition, we will investigate other features to enhance the current method.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (No.2012-0007149) and by Seoul Creative Human Development Program funded by Seoul Metropolitan Government (No.HM120006).

References

[1] A. April, J.H. Hayes, A. Abran, and R. Dumke, “Software maintenance maturity model (SMmm): The software maintenance process model,” J. Software Maintenance and Evolution: Research and Practice, vol.17, pp.197–223, 2005.
[2] J. Anvik and G.C. Murphy, “Determining implementation expertise from bug reports,” Proc. 4th International Workshop on Mining Software Repositories, pp.1–8, 2007.
[3] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?,” Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.308–318, 2008.
[4] Eclipse, http://www.eclipse.org
[5] Mozilla, https://bugzilla.mozilla.org
[6] Bugzilla, http://www.bugzilla.org
[7] N. Jalbert and W. Weimer, “Automated duplicate detection for bug tracking system,” Proc. 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp.52–61, 2008.
[8] J. Anvik, L. Hiew, and G.C. Murphy, “Who should fix this bug?,” Proc. 28th International Conference on Software Engineering, pp.361–370, 2006.
[9] T. Zhang and B. Lee, “An automated bug triage approach: A concept profile and social network based developer recommendation,” Proc. 8th International Conference on Intelligent Computing, LNCS 7389, pp.505–512, 2012.
[10] L. Anderson, D. Krathwohl, and P. Airasian, A Taxonomy of Educational Objectives, Abridged Edition, Available at: www.amazon.ca, Allyn&Bacon, 2000.
[11] S. Rao and A. Kak, “Retrieval from software libraries for bug localization: A comparative study of generic and composite test models,” Proc. 8th International Working Conference on Mining Software Repositories, pp.43–51, 2011.
[12] S. Tong and D. Kolber, “Support vector machine active learning with applications to text classification,” J. Machine Learning Research, vol.2, pp.45–66, 2002.
[13] C. Sun, D. Lo, X. Wang, J. Jiang, and S.C. Khoo, “A discriminative model approach for accurate duplicate bug report retrieval,” Proc. 32th ACM/IEEE International Conference on Software Engineering, pp.45–54, 2010.
[14] P. Runson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language process,” Proc. 29th International Conference on Software Engineering, pp.499–510, 2007.
[15] A. Sureka and P. Jalote, “Detecting duplicate bug report using character N-gram-based features,” Proc. 17th Asia Pacific Software Engineering Conference, pp.366–374, 2010.
[16] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, “An approach to
detecting duplicate bug reports using natural language and execution,” Proc. 30th ACM/IEEE International Conference on Software Engineering, pp.461–470, 2008.
[17] C.E.A. da Cunha, Y.C. Cavalcanti, P.A. da Mota Silveira Neto, E.S. de Almeida, and S.R. de Lemos Meira, “A visual bug report analysis and search tool,” Proc. 22nd International Conference on Software Engineering and Knowledge Engineering, pp.742–747, 2010.
[18] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, “Duplicate bug reports considered harmful…but really?,” Proc. 24th International Conference on Software Maintenance, pp.337–345, 2008.
[19] C. Sun, D. Lo, S.C. Khoo, and J. Jiang, “Towards more accurate retrieval of duplicate bug reports,” Proc. 26th International Conference on Automated Software Engineering, pp.253–262, 2011.
[20] 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,” Proc. 27th International Conference on Automated Software Engineering, pp.70–79, 2012.
[21] T. Zhang and B. Lee, “A bug rule based technique with feedback for classifying bug reports,” Proc. 11th IEEE International Conference on Computer and Information Technology, pp.336–343, 2011.
[22] T. Zhang and B. Lee, “A hybrid bug triage algorithm for developer recommendation,” Proc. 28th Annual ACM Symposium on Applied Computing, pp.1088–1094, 2013.
[23] D. Matter, A. Kuhn, and O. Nierstrasz, “Assigning bug reports using a vocabulary-based expertise model of developers,” Proc. 6th IEEE Working Conference on Mining Software Repositories, pp.131–140, 2009.
[24] J. Xuan, H. Jiang, Z. Ren, and W. Zou, “Developer prioritization in bug repositories,” Proc. 34th IEEE Conference on Software Engineering, pp.25–35, 2012.
[25] Q. Jiang and M. Sun, “Semi-supervised SimHash for efficient document similarity search,” Proc. 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, pp.93–101, 2011.
[26] S. Sood and D. Loguinov, “Probabilistic near-duplicate detection using Simhash,” Proc. 20th ACM International Conference on Information and Knowledge Management, pp.1117–1126, 2011.
[27] Y.S. Lin, T.Y. Liao, and S.J. Lee, “Detecting near-duplicate documents using sentence-level features and supervised learning,” Expert System with Application, vol.40, no.5, pp.1467–1476, 2012.
[28] S. Friedman, P. Kra, H. Yu, M. Krauthammer, and A. Rzhetsky, “GENIES: A natural-language processing system for the extraction of molecular pathways from journal articles,” Bioinformatics, vol.17, pp.S74–S82, 2001.
[29] L. Zhang, J. Zhu, and T. Yao, “An evaluation of statistical spam filtering techniques,” ACM Trans. Asian Language Information Processing, vol.3, pp.243–269, 2004.
[30] D. Fradkin and I. Muchnik, “Support vector machines for classification,” DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol.70, pp.13–20, 2006.
[31] B. Nallapati, “Discriminative models for information retrieval,” Proc. 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp.64–71, 2004.
[32] M.J. Li, M.K. Ng, Y.M. Cheung, and J.Z. Huang, “Agglomerative fuzzy K-means clustering algorithm with selection of number of clusters,” IEEE Trans. Knowl. Data Eng., vol.20, no.11, pp.1519–1534, 2008.
[33] S. Ponzetto and R.Navigli, “Large-scale taxonomy mapping for restructuring and integrating Wikipedia,” Proc. 21st International Joint Conference on Artificial Intelligence, pp.2083–2088, 2009.
[34] S. Ponzetto and M. Strube, “Deriving a large scale taxonomy from Wikipedia,” Proc. 22nd Conference on the Advancement of Artificial Intelligence, pp.1440–1445, 2007.
[35] C. Goutte and E. Gaussier, “A probabilistic interpretation of precision, recall and F-score, with implication for evaluation,” Lecture Notes in Computer Science, vol.3408, pp.345–359, 2005.
[36] A. Choudhury, “Wilcoxon signed rank test,” Retrieved from Explorable.com: http://explorable.com/wilcoxon-signed-rank-test, 2009.
[37] D.R. Pontin, S. Schliebs, S.P. Worner, and M.J. Watts, “Determining factors that influence the dispersal of a pelagic species: A comparison between artificial neural networks and evolutionary,” Ecological Modelling, vol.222, pp.1657–1665, 2011.
[38] J. Park, M. Lee, J. Kim, S. Hwang, and S. Kim, “Cos TRIAGE: A cost-aware triage algorithm for bug reporting systems,” Proc. 25th AAAI Conference on Artificial Intelligence, pp.139–144, 2011.