International Journal of Software Engineering and Knowledge Engineering

A Unified Framework for Bug Report Assignment

Yuan Zhao[1], Tieke He[1]*, Zhenyu Chen[1]
[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
*hetieke@gmail.com

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

It is typically a manual, time-consuming, and tedious task of assigning bug reports to individual developers. Although some machine learning techniques are adopted to alleviate this dilemma, they are mainly focused on the open source projects, which use traditional repositories such as Bugzilla to manage their bug reports. With the boom of the mobile internet, some new requirements and methods of software testing are emerging, especially the crowdsourced testing. Unlike the traditional channels, whose bug reports are often heavyweight, which means their bug reports are standardized with detailed attribute localization, bug reports tend to be lightweight in the context of crowdsourced testing. To exploit the differences of the bug reports assignment in the new settings, a unified bug reports assignment framework is proposed in this paper. This framework is capable of handling both the traditional heavyweight bug reports and the lightweight ones by (i) first preprocessing the bug reports and feature selections, (ii) then tuning the parameters that indicate the ratios of choosing different methods to vectorize bug reports, (iii) and finally applying classification algorithms to assign bug reports. Extensive experiments are conducted on three datasets to evaluate the proposed framework. The results indicate the applicability of the proposed framework, and also reveal the differences of bug report assignment between traditional repositories and crowdsourced ones.

Keywords: Bug Report Assignment; Crowdsourced Bug Report; Traditional Bug Report.

1. Introduction

Bug reports are crucial to software development. Typically, bug reports contain a detailed description of a failure and occasionally hint at the location of the fault in the code [1]. As a consequence, software developers spend a significant portion of their resources dealing with user-submitted bug reports. For some open source software projects, the number of bug reports normally outstrips the resources available to triage and assign them. In that way, some reports may be handled slowly or not at all [2].

In addition, with the advent of the mobile internet era, new trends of software testing such as crowdsourced testing and mobile application testing are emerging. On the one hand, this adds up to the number of bug reports to a large scale, as more and more users are presenting feedback on various aspects of software. On the other hand, bug reports in these new channels are often lightweight or "non-informative". It is found that only 35.1% of these bug reports can directly help developers improve their apps according to a

survey [3]. Traditional reporting channels such as Bugzilla[a], GNATS[b] and JIRA[c] provide heavyweight bug reports. These reports have different kinds of pre-defined fields, free-form text, attachments and dependencies, which lead to a huge difference in processing different bug reports.

To sum, the research topics as dealing with the sizeable number of bug reports in software projects as well as the issue of lightweight bug reports are attracting more and more attention of researchers, in both software engineering and data mining communities.

However, existing methods are focused on some large open source software developments such as eclipse, whose bug reports are typically submitted by people with professional background, in other words, these bug reports cannot be referenced for other scenes. We also notice that, all these methods deal with only part of the whole process of bug management, i.e., either the preprocessing of bug reports or the vectorization of bug reports, or the classification of bug reports. The challenges lying in the management of bugs are that, the framework shall handle both the sizeable traditional heavyweight bug reports and the lightweight ones in the new channels, such as crowdsourced testing and mobile application testing, and that the framework must take the process as a whole, which means it shall consist of the preprocessing and vectorization, as well as the classification. In the meantime, it has been a trend that many companies are outsourcing their testing jobs to third-party, which means such third-party will come across various types of bug reports. Also, the different compositions of the traditional heavyweight bug reports and lightweight bug reports will call for different techniques, and manually selection would be unsuitable.

To the best of our knowledge, there are few studies that were focused on dealing with both heavyweight bug reports and lightweight ones, as well as forming a unified framework that considers preprocessing, vectorization and classification as a whole. This paper formally formulates this as a new research problem. Specifically, to address this challenging problem, we propose a unified framework for bug management. Generally speaking, given a bunch of bug reports, the proposed framework first preprocesses them by stemming and feature selection, then vectorizes the bug reports through a tuning parameter which decides the ratio of choosing different vectorizing methods, and finally chooses the classification algorithm which is most suitable for these bug reports. The main difference between our framework and other traditional frameworks lies in its adaptability and compatibility.

We validate the efficacy of our framework by conducting experiments on three datasets, including bug reports from the traditional channel (*Bugzilla*), the new channels crowdsourced testing and the mobile application testing. Empirical results have validated the effectiveness and efficiency of our proposed framework. In short, this paper makes the following main contributions:

- We formulate a new problem which aims to manage bugs by handling both heavyweight and lightweight bug reports, as well as forming the process as a whole;

[a] www.bugzilla.org/
[b] www.gnu.org/software/gnats/
[c] www.atlassian.com/software/jira/

- We present a new framework to deal with this new problem, which is a unified framework;
- We evaluate our framework on three different channels of bug reports, from which the results indicate that it is effective and promising.

The rest of the paper is organized as follows: Section 2 reviews related work; Section 3 gives the problem statement and introduces some notations; Section 4 details the proposed framework; Section 5 presents the experiments; finally Section 6 concludes this work and outlines the future work.

2. Related Works

In this section, we review a number of existing works in the areas of crowdsourced testing, mobile application testing and automated bug assignment.

Crowdsourced Testing. Crowdsourced testing is gaining more and more attention from the industry. Meanwhile, it has also become a new trend in software engineering research community. Crowdsourcing is emerging as the new online distributed problem solving and production model in which networked people collaborate to complete a task. As a result, it is becoming increasingly popular and has been studied as a usability engineering method [4]. Liu et al. [5] applied crowdsourced testing in usability testing. They studied both methodological differences for crowdsourcing usability testing and empirical contrasts to results from more traditional, face-to-face usability testing. Pastore et al. [6] studied whether it is possible to exploit crowdsourcing to solve the oracle problem: generated test input depends on a test that oracle requires human input in one form or another. Also, in a most recent work, Feng et al. [7] proposed the *DivRisk* strategy to prioritize sizeable test reports in crowdsourced testing, which can help developers inspect a wide variety of test reports as well as avoid duplicates. Different from all these earlier works that are focused on the strategies and applicability of applying crowdsourced testing, we mainly address the problem of handling the sizeable and lightweight bug reports.

Mobile Application Testing. Mobile application testing refers to app marketplace analysis as well as the mobile platform based bug management. With the rocketing development of mobile applications, app marketplace has been drawing more and more attention among researchers within and outside the software engineering community. However, there are three challenges in processing, analyzing and mining user feedback. 1). App marketplace or mobile testing platforms include *a large quantity* of reviews, which we use as lightweight bug reports. For these reviews, a large amount of effort is required to manually analyze and process them. 2). These reviews or feedbacks are normally in the form of *unstructured text* that is difficult to parse and analyze, which adds to the burden of developers and analysts [3]. 3). The *quality* of these lightweight bug reports varies greatly, from professional suggestions on improving specific aspects of apps to generic praises and complaints (e.g. "*Great, Addictive & Fun*", "*I love it!*", "*waste of money*"). Some works studied the quality of apps, from perspectives of both developers and users. Agawal et al. studied how to better diagnose unexpected app behavior [8]. Khalid [9] focused on analyzing specific app reviews to understand why users complain about apps from the user perspective. The work of Mu-

dambi et al. showed that reviews play a key role in the purchasing decision of products at the online retailer Amazon[d] [10, 11]. Moreover, there are several previous studies that are similar to our work on mining user reviews in app marketplace. In [12], Pagano and Maalej conducted an exploratory study to analyze the user reviews, by studying 1). the usage and impact of feedback through statistical analysis, and 2). the content of feedback via manual content analysis. Iacob et al. [13] presented the *MARA* prototype which uses a list of linguistic rules to automatically retrieve feature requests from online user reviews. Galvis Carreño [14] adopted the Aspect and Sentiment Unification Model (*ASUM*) proposed in [15] to extract topics for requirements changes.

Automated Bug Assignment. The problem of automated bug assignment has been previously addressed by both the software engineering and data mining research communities [16, 17, 18, 19]. Murphy et al. [16] used a text categorization approach which was based on the *Naive Bayes* algorithm, to tackle the problem of automated bug assignment for the Eclipse project, and their approach achieved 30% precision. Much similar to their work, Canfora et al. [17] used an information retrieval approach to automate the process of bug assignment, and they reported recall levels of around 20% for the Mozilla project. Mockus and Herbsleb [20] used source code change data from a version control repository to determine appropriate experts to work on given elements of software projects. Unlike the above mentioned works, our framework determines developer expertise based on their history of past bug-solving tasks extracted from the bug repository data. Podgurski et al. [21] applied a machine learning algorithm to bug reports, in their case, the algorithm was applied to cluster function call profiles from automated fault reports. The clusters were then used to prioritize software faults and to help diagnosis their cause rather than to assign reports to appropriate developers. As a matter of fact, the bug assignment problem is similar to the problem of recommending experts in particular parts of the system to assist with the development process. Anvik et al. [19] proposed an approach to automate bug assignment using machine learning techniques, in which they recommended potential resolvers for the bug by mining history bug reports that the developers have been assigned to and resolved for the system. However, the approach is semi-automated because of the triager. The team lead for example, uses the recommendation list to select a potential developer who can fix the bug and makes the final decision on assigning this bug to the appropriate developer, considering the team's current workloads and schedules.

3. The Problem Statement

For ease of presentation, we define the key data structures and notations used in this paper. Table 1 lists the relevant notations used in this paper.

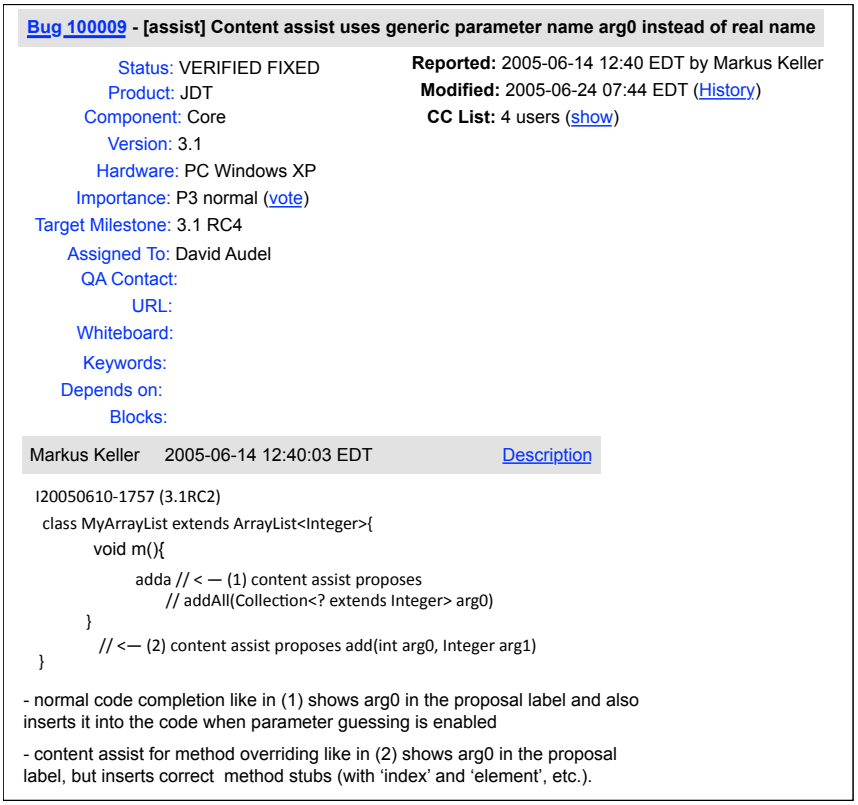
The "bug report assignment" task is extremely important in software development. In this paper, we formulate it as a new research problem, which aims to facilitate developers to deal with the sizeable bug reports in software projects, as well as to handle the issue of lightweight bug reports, which meets the problem of short text and unstructured data.

[d]www.amazon.com

Table 1: Notations used in the paper

SYMBOL	*DESCRIPTION*
D_p	bug reports of project p
v	the identifier of a bug report
$c[v]$	the textual description of bug report
$d[v]$	the developer responsible for bug report
$W[v][]$	result of word segmentation of bug report
$RSW[v][]$	result after filtering stop words
$F[v][]$	the result of feature selection
$wV[v][]$	word vector of bug report
$tV[v][]$	topic vector of bug report
$V[v][]$	final vector of bug report
αF	feature selection algorithm
βFR	feature selection ratio
γMR	vectorization ratio
δC	classification algorithm
K	the number of bug reports

Bug Report. Traditionally, a bug report contains information such as software version, summary of the bug, component where the bug occurs, developer who is responsible and so on. In our framework, we are dealing with bug reports that vary greatly, which are from traditional channel, crowdsourced testing, as well as mobile application testing scenarios. Examples of these bug reports are illustrated in Figure 1, Table 2, and Figure 2, respectively. Figure 1 shows an example of a traditional bug report (Eclipse project in *Bugzilla*), from which we can find a summary of the bug, as well as a detailed description that points out the product, component, the operation system and when the bug happens and so on. Table 2 shows an example of the crowdsourced testing bug report, in which the reporters use Chinese to report bugs (for ease of understanding, we've translated the example into English). It mainly differs from the traditional bug report in that the form of bug report is free, and varies greatly in length and textual description. Figure 2 presents a mobile application testing bug report, in which ordinary users can submit bug reports through a mobile testing platform we developed (*Mubug*). Characteristics of such bug reports are: 1) people submit bug reports through mobile devices. As a result, the textual description are normally short; 2) text in these bug reports tend to be casual and informal; 3) in addition to the text form, novel forms such as voice message are also supported in such platforms. To note, this project is also in the circumstance of Chinese (we also translated it into English). As a matter of fact, a bug report has three attributes in our framework: identifier, textual description and developer. We use v to represent a bug report identifier, $c[v]$ to denote its corresponding textual description, and $d[v]$ is used to indicate the developer who is responsible for bug report v.



Bug 100009 - [assist] Content assist uses generic parameter name arg0 instead of real name

Status: VERIFIED FIXED
Product: JDT
Component: Core
Version: 3.1
Hardware: PC Windows XP
Importance: P3 normal (vote)
Target Milestone: 3.1 RC4
Assigned To: David Audel
QA Contact:
URL:
Whiteboard:
Keywords:
Depends on:
Blocks:

Reported: 2005-06-14 12:40 EDT by Markus Keller
Modified: 2005-06-24 07:44 EDT (History)
CC List: 4 users (show)

Markus Keller 2005-06-14 12:40:03 EDT Description

```
I20050610-1757 (3.1RC2)
class MyArrayList extends ArrayList<Integer>{
    void m(){
        adda // < — (1) content assist proposes
            // addAll(Collection<? extends Integer> arg0)
    }
      // <— (2) content assist proposes add(int arg0, Integer arg1)
}
```

- normal code completion like in (1) shows arg0 in the proposal label and also inserts it into the code when parameter guessing is enabled

- content assist for method overriding like in (2) shows arg0 in the proposal label, but inserts correct method stubs (with 'index' and 'element', etc.).

Fig. 1: A bug report instance of Eclipse

Table 2: Bug report instances of Baidu Input

Bug ID	Test Case	Bug Description
1	1004	Metro style layout, the displays are not consistent in Portrait orientation and Landscape orientation
2	1014	The input characters will disappear when rotate the screen, in unfinished typing status
3	1005	There will be disorders when input numbers under the input of Uyghur, as the read is from right to left. Test Phone: SAMUNG I9100. Test Environment: Short Text Message
4	1021	When using the default theme, float effect disabled, under landscape orientation, the input area is blocked. Test Phone: Mi 2A, Meizu M2. Test Environment: Mobile QQ2012

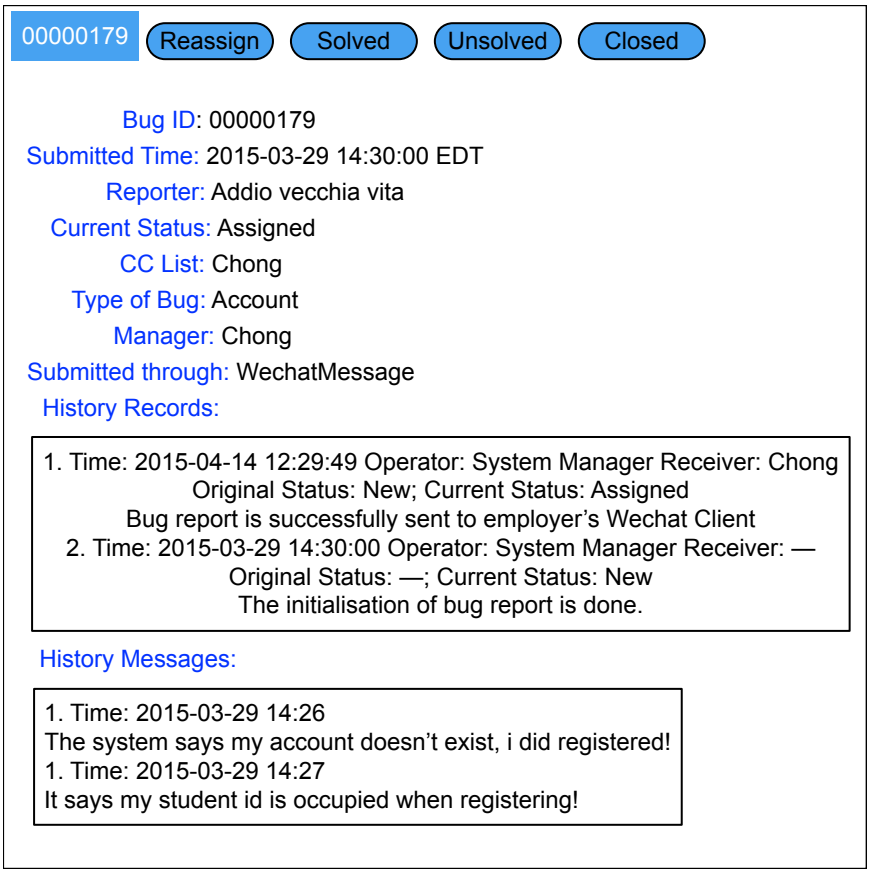


Fig. 2: A bug report intance of Mooctest

Given a dataset D as the union of a collection of bug reports for a project, we aim to infer developers for bug reports in D. We formulate our problem that takes into account all the three scenarios in a unified fashion as follows.

Problem 1. *Management of Bugs* Given a bug management dataset D that contains both history bug solving records and new coming bug reports, our goal is to automatically assign appropriate developer for a target bug report v with our framework. We also aim to process all the bug reports in the format of feature vector for further usages, such as retrieval and cluster, as well as vectorization of the bug reports, in which way we can store them in the structured format.

4. Our Framework

In this section, we first give an overview of the proposed framework that aims at dealing with the problem stated in Section 3, and then present each step of our framework in detail.

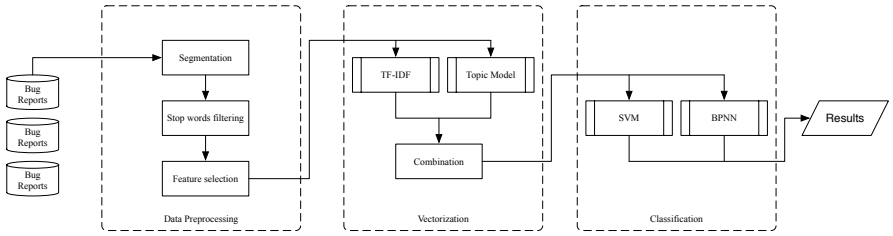


Fig. 3: An instantiation of the proposed *Unified Framework for Bug Report Assignment*

4.1. *Overview*

Figure 3 presents an overview of the process flow diagram of our proposed bug report management framework. The first step preprocesses the raw bug reports into well-structured format to facilitate the following steps (Section 4.2). The second step combines different vectorization strategies to further format the bug reports so as to suit bug reports from different scenarios (Section 4.3). The last step tests different machine learning algorithms in order to get the best performance of assignment from the proposed framework (Section 4.4).

4.2. *Preprocessing*

As described in Definition 3, bug reports vary vastly in different channels. Following the traditional process of text categorization, we perform the data preprocessing as segmentation, filtering the stop words and feature selection. We left the option of stemming because previous research [22] shows that it has less impact.

The general form of data preprocessing is shown in Algorithm 1. Segmentation is first performed on the textual description. For bug reports in English, the segmentation won't be a problem, as it is intrinsically separated by blanks, and we adopt the Stanford NLP [23]. On the contrary, for the Chinese bug reports, the results of segmentation greatly affect the effect of the whole task. We compared a series of Chinese lexical analyzer and finally choose the *ICTCLAS* [24] to do the segmentation.

Once we get the result of segmentation, we need to remove the stop words. Stop words are the set of common words such as *the*, *is*, *an* etc., these words are not important for our task of bug management and should be removed. Mainly, we adopt the *SMART* [25] stop word list. An original feature space is obtained for each bug report in this step.

One of the major characteristics of textual tasks is high dimensionality. Thus, dimension reduction which attempts to reduce the size of feature space without sacrificing the performance has been a critical step. Feature selection that selects a subset from original feature space according to evaluation criteria, is the most commonly used dimension reduction method in this field. In order to facilitate our following steps, we need to do the feature

Algorithm 1 Data Preprocessing

Input: Bug reports dataset D of a project;
Output: Bug reports as a list of feature words, $F[][]$;

```
for i from 1 to K do
    Segment c[i] to get W[i][];
end for
for i from 1 to K do
    Remove stop words from W[i][] to get RSW[i][];
end for
for i from 1 to K do
    for αF in [IG, CHI] do
        for βFR in [20%, 40%, 60%, 80%, 100%] do
            Process RSW[i][] with αF and βFR to get F[i][];
        end for
    end for
end for
```

selection in our data preprocessing. There are three classes of feature selection methods [26]: 1) the embedded approach where process of the feature selection is embedded in the induction algorithm; 2) the wrapper approach where the evaluation function is used to select the feature subset as a wrapper around the classifier algorithm; 3) filtering approach that uses the evaluation function to select the feature subset, which is independent of the classifier algorithm. In this paper, we use the filtering approach which suits best. There are many efficient and effective filtering feature selection methods that have been applied to text processing, such as Information Gain (IG) [27], Chi-square statistics (CHI) [28], Mutual Information (MI) [29], Document Frequency (DF) [27], improved Gini index (GINI) [30], and so on. As we are attempting to obtain a framework that is adaptable, we compare different feature selection methods in this step. In combination with related work, to best demonstrate our objective and without loss of generality, we adopted the IG and CHI methods in this paper. In detail, we choose between the two methods, as well as the different ratios of applying certain feature selection methods, as can be seen from Algorithm 1, in which 100% means no feature selection is adopted.

4.3. *Vectorize the Bug Reports*

Before we can apply machine learning algorithm to the free-form text found in the summary and description, the text must be converted into a feature vector. As we have obtained the feature words for each bug report, we further normalize the frequency based on certain models, such as document length, intra-document frequency and inter-document frequency outlined by Rennie et al. [31].

The general form of vectorizing the bug reports is shown in Algorithm 2. Traditionally, vector space models are adopted to vectorize the bug reports, but they are limited to the source of bug reports, that is to say, for a single set of bug reports, one certain vector space model is chosen to perform the vectorization. While in our case we aim to deal with bug reports from various channels, we shall meet the requirement of compatibility.

Vector space model and topic model are two sorts of methods that are most adopted in text categorization and text mining. Vector space models get relatively good performance in that it only considers the feature words, as a result, it has a low computational cost. Given a collection of bug reports, topic model assumes that a bug report is about the proportion over all the topics, and each topic is about the proportion of all the feature words. Topic model is perfect for cases where the corpus are very large, as Dirichlet posterior parameters are introduced in both the topic layer and word layer, and the number of topics are fixed during the training process.

The main idea behind this step is that, we propose a method that combines the vector space model with topic model. The new method takes advantage of both models to well represent the bug reports, through which it can deal with bug reports from various channels. Specifically, to best illustrate our idea in this step and without loss of generality, we choose TF-IDF for the vector space model. *Term Frequency-Inverse Document Frequency* [32] is a numerical statistic model which is intended to show how important a word is to a document in a collection or corpus. For the topic model, we choose the original *Laten Dirichlet Allocation* [33] model.

As shown in Algorithm 2, we first compute the TF-IDF for all the words. That is, after choosing the feature words in the previous step, we should identify the weight of them, as different words vary in importance to bug reports, a numerical value is required to indicate such difference. Normally, words that are important to bug reports shall have a large value, and those less important will have a small value. In this paper, we adopt the TF-IDF to compute the numerical value for the feature words. The TF-IDF not only considers the frequency of words, but also takes into account the influence of the frequency of documents. The formula is defined as follows:

$$w_{ik} = tf_{ik} * \log \frac{N}{df_i}$$

in which, tf_{ik} is the frequency of feature word in bug report k, N is the number of bug reports, and df_i is the frequency of feature word in all the bug reports. In this way, we fulfill the process of vector space model for all the bug reports.

Following, we generate the representation of bug reports with topic model. LDA is a very popular topic model, which is adopted in various use cases. As for our case, the model assumes that there are K hidden topics among bug reports, every bug report is about the multinomial of all the topics, and each topic is about the multinomial of all the feature words. The reason we choose LDA is that it is very simple and intuitive, and this generative model ensures that each bug report is the proportion of all the topics, which facilitates the training process of classification. Furthermore, we employ Gibbs sampling to perform approximate inference in this paper. Through which, we first get a set of training parameters, and then get the distribution of all the bug reports over all the topics.

After we have adopted the vector space model and LDA on the bug reports, we combine the two according to different ratios. As a matter of fact, in the experiment we set the ratio of vector of topic space to 0, 10%, and so on, till 100%. Here, 0 means solely using the words vector space, and 100% means using the vector obtained from topic model. To this

point, the vectorization of the bug reports is completed.

Algorithm 2 Vectorization

Input: Bug reports in the format of $F[][]$;
Output: Bug reports vectorization $V[][]$;
for i from 1 to K **do**
  Compute TF-IDF of $F[i][]$ to get $wW[i][]$;
end for
Perform Gibbs sampling for $F[][]$;
for i from 1 to K **do**
  Obtain $tV[i][]$ from the topic proportion;
end for
for i from 1 to K **do**
  for vMR in $[0\%, 10\%, 20\%, 30\%, 40\%, 50\%, 60\%, 70\%, 80\%, 90\%, 100\%]$ **do**
    $V[i][] = (1 - vMR) * wV[i][] + vMR * tV[i][]$;
  end for
end for

4.4. *Assigning Bug Reports*

The key question in this step of the assignment is to choose the classification algorithm to train a classifier from the training set. Then we can apply the classifier to the new coming tasks of assigning bug reports. Classification has always been a hot topic in data mining. Many mature classification algorithms have been introduced in this area, such as Naive Bayes [34], KNN [35], C4.5 [36], SVM [37], Neural Network [38], and so on. In this paper, we choose two representative algorithms to demonstrate the effectiveness of our framework, which are SVM and Back Propagation Neural Network. Together with the previous two steps we get various kinds of combinations, respectively for the two algorithms. Finally, a most suitable combination will stand out for a given set of bug reports.

SVM deals with classification tasks by constructing hyperplanes in a multidimensional space, which separates cases of different class labels. It supports both regression and classification tasks and can handle multiple continuous and categorical variables. One characteristic that makes SVM an excellent algorithm is that its ability to learn can be independent of the dimensionality of the feature space. The complexity of SVM is that its hypotheses based on the boundary with which they separate the data, not the number of features.

Back propagation is a method that monitors learning. It utilizes the method of mean square error and gradient descent to realize the modification to the connection weight of network. The modification to the connection weight of the network is aimed at achieving the minimum error sum of squares. The initial weight of the network is generally generated at random in certain interval, and the training starts with an initial point and reaches gradually to a minimum of error along the slope of error function.

In this paper, we adopt the three-layer back propagation neural network. Suppose its input node is x_i, the node of hide layer is y_j, and the node of output layer is z_h. Also

suppose that weight value of network between the input node and node of hide layer is w_{ji}, and weight value of network between the nodes of hide layer and output layer is v_{hj}, and the expected value of the output node is t_h, $f(\cdot)$ is the active function. The details of the computational formula of the model can be referred in [39].

Algorithm 3 Classification

Input: Bug reports vectorization $V[][]$;
Output: Classification Algorithm;
for δC in [SVM, BPNN] **do**
  for i from 1 to 10 **do**
    Randomly divide $V[][]$ into two parts,the training set and the test set, with ratio $2:1$;
    Adopt algorithm δC to get the precision;
  end for
  Compute the avarage precision of algorithm δC;
end for
Return the algorithm with the highest precision.

The general form of classification is presented in Algorithm 3. Firstly, an algorithm is chosen, either SVM or the Back Propagation Neural Network. Then, a three fold cross validation experiment is conducted. In each round, the bug reports are randomly divided into training set and test set, according to the ratio $2:1$, followed by the training process and computation of precision. After all the rounds, an average precision of the specific algorithm is computed for the certain bug reports dataset. Finally, the algorithm that generates the highest precision is returned for that dataset.

To sum up, the proposed framework deals with bug reports in three steps. In the data preprocessing step, feature selection algorithm αF and the ratio of different algorithms βFR shall be decided. In the vectorization step, the combination ratio γMR of the vector space model and the topic model will be assigned. In the assigning step, the classification algorithm δC will finally be chosen for that specific dataset.

5. Experiments

Based on the framework proposed in Section 4, we conduct an extensive set of experiments to empirically study how different combinations of all the variables will affect the results of bug report assignment. Specifically, we use three datasets, i.e., the Eclipse Bugzilla bug reports from the traditional channel, the Baidu Input bug reports from the crowdsourced testing, and the Mooctest bug reports from the mobile application testing. We begin this section by first introducing the details of these three datasets, and then presenting the metrics we adopt, followed by evaluating various aspects, as well as the comparison among the three datasets.

5.1. *Datasets*

The first dataset we choose is the Bugzilla bug reports of the Eclipse project. For a bug report in Bugzilla, there are various status, such as "*New*", "*Assigned*", "*Resolved*", "*Closed*" and so on. For our experiments, we only choose bug reports with the status "*Resolved*" and "*Closed*", because in these bug reports we can find the developer who finally fixed the bug, and further we will use this information to label the bug report.

We choose the bug reports between id 100001 and 105000 to conduct the experiments. After filtering with status "*Resolved*" and "*Closed*", we get $2,533$ bug reports. For the ease of training, we also filter out developers that handled less than 10 bug reports, and remove the corresponding bug reports. In this way, we finally get a dataset of Eclipse project bug reports, with $1,171$ bug reports, and 78 developers, which forms the ground truth dataset. Table 3 shows the distribution of the number of developers that fixed bugs, from which we can see most developers fixed bugs between 11 and 20.

Table 3: Number of Bugs Developers Fixed

#Fixed Bugs	#Developers
[11-20]	48
[20-30]	17
[30-40]	7
[40-50]	1
[50-60]	3
[60-70]	1
[70-80]	0
[80-90]	1

For this final set of bug reports, we utilize the summary and description of the bug report of the following experiments. The average length of the bug reports is 795.35 words.

For the crowdsourced bug reports, we choose the Baidu Input project testing tasks, which is from a collaboration between us and the industry. As it is relatively difficult for us to acquire the information of exactly who finally fixed the reported bugs, we use the function modules as the classes of bug reports. In fact, for the obtained Baidu Input crowdsourced bug reports we totally have 10 function modules, which are "*Interface*", "*Word Input*", "*Cloud Input*", "*Voice Recognition*", "*Contacts Import*", "*Input Switch*", "*Account*", "*Dictionary*", "*Input Panel Height Adjust*" and "*Hand Input Model*". The average length of bug reports in this dataset is 78.69 words. Table 4 shows the number of bug reports of every module.

We choose the bug reports that were submitted by users through the *Mubug* platform of mobile application testing (*Mooctest*) [e], between 2014-12-15 and 2015-04-16. After removing some meaningless reports, we are left with 211 bug reports. The average length of these bug reports is 40.76 words. For these mobile application testing bug reports, there

[e] mooctest.net

Table 4: Number of Bugs in Function Module

Function Module	#Bug Reports
Interface	13
Word Input	40
Cloud Input	7
Voice Recognition	40
Contacts Import	14
Input Switch	17
Account	86
Dictionary	18
Input Panel Height Adjust	50
Hand Input Model	27

are 5 developers in charge of different aspects, specifically, "*Account Problems*", "*Frontpage Problems*", "*Quiz Problems*", "*Client Problems*" and "*Python Counting Problems*". As each developer is responsible for one sort of problems, we further use this information as the label of classification. The number of bug reports each developer is in charge of is presented in Table 5.

Table 5: Number of Bugs Developers In Charge of

Developers In Charge	#Bug Reports
D_1-*Account Problems*	77
D_2-*Frontpage Problems*	35
D_3-*Quiz Problems*	51
D_4-*Client Problems*	29
D_5-*Python Counting Problems*	19

Obviously, there are various differences among these three datasets, including the language (English Versus Chinese), formal and informal text, and the length of the bug reports, which is the biggest difference. All these indicate that the bug report assignment could be different in traditional channel, crowdsourced channel and mobile application testing.

5.2. *Key Steps & Performance Metrics*

We have already introduced the key variables in our framework, i.e., the feature selection algorithm αF and feature selection ratio βFR in the data preprocessing phase, the combination ratio of the vector space model and the topic model γMR in the vectorization phase, and the choosing of different classification algorithm δC. To find the best combination of these variables on specific datasets, we conduct the empirical study with ground truth. For the feature selection, we choose the Information Gain (IG) [27] and Chi-square statistics (CHI) [28], and set the feature selection ratio to 20%, 40%, 60%, 80% and 100%. For the vectorization, the combination ratios we set for the vector space model and the topic model

are 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100%. For the classification algorithm, we choose two representative algorithms, SVM and Back Propagation Neural Network, which are said to have the best performances in classification tasks.

We would like to formalize such steps as follows:

1. Perform the Information Gain (IG) and Chi-square statistics on the dataset, and set the feature selection ratio to 20%, 40%, 60%, 80% and 100%, in which 100% means no feature selection algorithm is adopted. Through this step, we can have 9 groups of experiments.
2. For the 9 groups of experiments generated from the previous step, we further choose the ratio between the vector space model and the topic model with 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100%, in which 0 means solely using the vector space model, while 100% means using the topic model only. In this way, we have 99 groups of experiments by far.
3. Finally, we run the SVM and Back Propagation Neural Network on 99 groups of experiments we obtained from previous steps, which means we have 198 combinations to compare with in total.

Then, we evaluate the proposed framework on the three datasets with the 198 groups of experiments. In the evaluation, to best reduce the influence of over-fitting during training process, we adopt 3-fold cross validation, that is, we randomly choose $2/3$ of the all the bug reports as the training set, and the rest $1/3$ as test set, and repeat this process 10 times to take a mean accuracy as the result.

For the evaluation, we use the metric *Accuracy*. The computation of *Accuracy* proceeds as follows. We define *hit* for a single test case as either the value 1 if the class in the test case equals the ground truth, or the value 0 if otherwise. The overall *Accuracy* is defined by averaging all test cases:

$$Accuracy = \frac{\#hit}{|S_{test}|} \tag{1}$$

where *#*hit denotes the number of hits in the test set, and $|S_{test}|$ denotes the number of all test cases. As we are attempting to evaluate the whole framework other than single cases, this metric well suits our scenario.

5.3. *Experimental Results*

In this part, we will present the detailed results of our experiments. From Section 5.2 we know that, for each dataset, we have 198 groups of experiments, which means in that case we will have too many experimental results, so we mainly focus on four aspects to present and analyze the results. First, we show how much the ratio of feature selection affects the result of classification. Second, we compare different ratios of combinations between the vector space model and the topic model. Third, we present the difference of choosing SVM and Back Propagation Neural Network. Fourth, we report the overall combinations of the variables that get the best performance.

5.3.1. *Feature Selection*

We demonstrate the experimental results by the datasets. As mentioned in the previous part, we choose two feature selection algorithms, and set the selection ratio to 20%, 40%, 60%, 80% and 100%, in which 100% means no feature selection algorithm is adopted. As a result, we have 9 groups after preprocessing. To compare between them, we use the mean *Accuracy* of all the possible combinations as the result of the according to feature selection decision.

Figure 4 shows the mean *Accuracy* of different combinations of feature selection algorithms over the three datasets. From Figure 4 we can see that, on the Eclipse dataset, it is effective to adopt the feature selection. Compared to ratio 100% (35.57%), which means no feature selection is conducted, the mean *Accuracy* slightly increases when choosing the *IG* algorithm and set ratios 80% (37.19%) and 60% (37.02%). While for the *CHI*, except when the ratio is 20%, the other 3 cases all outperform the case when no feature selection is adopted. We can conclude that, for the Eclipse bug reports dataset, *CHI* is more suitable.

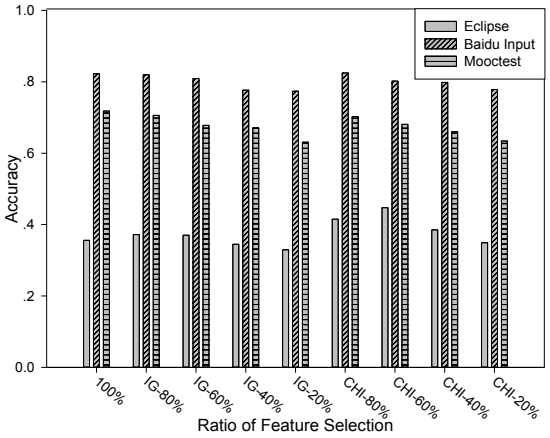


Fig. 4: Results of Feature Selection

While for the Baidu Input dataset and Mooctest dataset, the results are much better than that of the Eclipse dataset. This is because the bug reports are submitted by a limited group of people, which leads to the similarity of the textual description of bugs. Regarding the feature selection algorithm, it is clear that we get no better performance when including the *IG* or *CHI* algorithm. For both datasets, we observe that for either *IG* or *CHI*, when the ratio decreases (from 80% to 20%), the mean *Accuracy* of classification decreases. This is due to the short length of the crowdsourced bug reports and mobile application testing bug reports, in which cases when lower the ratio of feature selection, we suffer from the information loss. This phenomenon is more obvious in the Mooctest dataset, as the average length of Mooctest (40.76) bug reports is shorter than the Baidu Input (78.69).

5.3.2. *Vector Space Model vs. Topic Model*

We endeavor to show the influence of different choices over the vector space model and the topic model on various datasets. We choose the combination ratio as 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100%, in which 0 means solely using the vector space model, while 100% means using the topic model only. In this way, we have 11 groups of experimental results. We follow the approach in Section 5.3.1 to show the results.

Figure 5 illustrates the mean *Accuracy* of classification of different combinations of vector space model and the topic model.

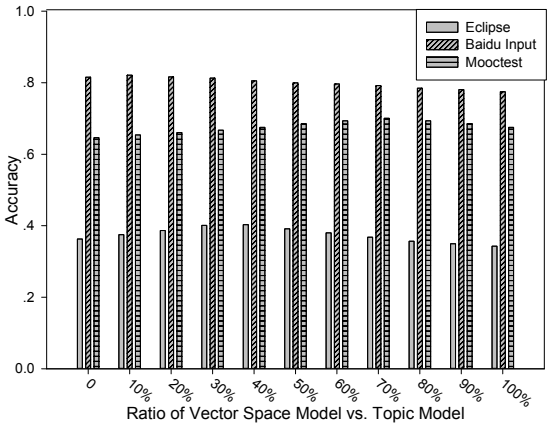


Fig. 5: Results of VSM vs. Topic Model

From Figure 5 we can see that the combination of these two methods is effective on the Eclipse dataset. When the ratio is 40%, the framework gets the best performance with the mean *Accuracy* 40.29%, which exceeds the results when only uses the vector space model (36.27%), and when only uses the topic model (34.29%). For the Baidu Input dataset, the framework gets the best performance when the ratio is 10%, which is 82.11%. Compared to the result of Eclipse, this dataset favors the vector space model more, which is also due to the textual similarity of bug reports, as vector space model is mostly based on the words. While for the Mooctest dataset, the framework has its peak when the combination ratio is 70%, and the mean *Accuracy* is 70.07%. This outperforms the results of solely using the vector space model (64.66%) and only using the topic model (67.53%). We also notice that for the Mooctest dataset, the framework leans more against the topic model, as the bug reports in mobile application testing are more semantic related.

5.3.3. *Classification Algorithms*

Figure 6 demonstrates the mean *Accuracy* of the two classification algorithms on the three datasets. From which we can infer that, in the traditional channels of bug reports (Eclipse), SVM model performs better than the Back Propagation Neural Network. While in the new

fashioned channels, such as crowdsourced (Baidu Input) and mobile application testing (Mooctest), the Back Propagation Neural Network can be a better choice.

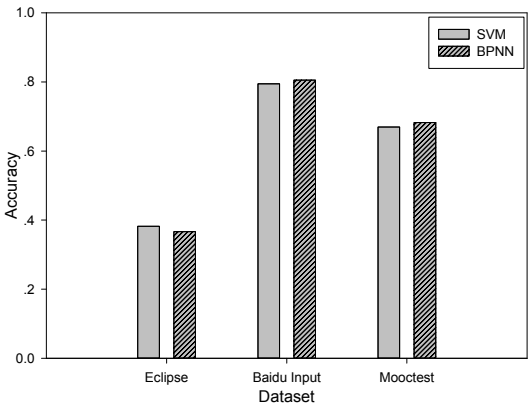


Fig. 6: Results of Classification Algorithms

5.3.4. *Overall Results*

In the previous three parts, we compared the results according to each component of the framework. As mentioned before, we have 198 combinations of the variables in total. Following, we will present the top 10 combinations of variables for each dataset that get the best performance.

Table 6: Top 10 of Eclipse

Rank	αF	βFR	γMR	δC	*Accuracy*
1	*CHI*	60%	40%	*SVM*	49.22%
2	*CHI*	60%	30%	*SVM*	48.88%
3	*CHI*	60%	50%	*SVM*	47.81%
4	*CHI*	60%	30%	*BPNN*	47.73%
5	*CHI*	60%	20%	*SVM*	47.17%
6	*CHI*	60%	60%	*SVM*	46.6%
7	*CHI*	60%	40%	*BPNN*	46.26%
8	*CHI*	60%	10%	*SVM*	46.24%
9	*CHI*	60%	50%	*BPNN*	45.89%
10	*CHI*	60%	70%	*SVM*	45.77%

Table 6 shows the top 10 combinations of variables for the Eclipse dataset that get the best performance. From which we can see that when the αF is *CHI*, the βFR is 60%, the γMR is 40% and chooses the SVM model, the framework gets the best performance with *Accuracy* 49.22%.

Table 7 shows the top 10 combinations of variables for the Baidu Input dataset that get the best performance. From which we can see that when no feature selection is performed,

the γMR is 20% and chooses the BPNN model, the framework gets the best performance with *Accuracy* 85.99%. From the results we can conclude that the framework is capable of finding combinations that beat the traditional classification model.

Table 7: Top 10 of Baidu Input

Rank	αF	βFR	γMR	δC	*Accuracy*
1	*None*	100%	20%	*BPNN*	85.99%
2	*None*	100%	30%	*BPNN*	85.52%
3	*CHI*	80%	10%	*BPNN*	85.42%
4	*CHI*	80%	0%	*BPNN*	85.41%
5	*None*	100%	10%	*BPNN*	85.25%
6	*None*	100%	0%	*BPNN*	85.13%
7	*CHI*	80%	10%	*SVM*	85.03%
8	*None*	100%	10%	*SVM*	85.02%
9	*None*	100%	40%	*BPNN*	84.88%
10	*CHI*	80%	20%	*BPNN*	84.28%

Table 8 shows the top 10 combinations of variables for the Mooctest dataset that get the best performance. From which we can see that when no feature selection is performed, the γMR is 70% and chooses the BPNN model, the framework gets the best performance with *Accuracy* 74.89%.

Table 8: Top 10 of Mooctest

Rank	αF	βFR	γMR	δC	*Accuracy*
1	*None*	100%	70%	*BPNN*	74.89%
2	*None*	100%	70%	*SVM*	74.67%
3	*None*	100%	80%	*BPNN*	74.37%
4	*IG*	80%	70%	*BPNN*	74.34%
5	*None*	100%	90%	*BPNN*	74.32%
6	*IG*	80%	80%	*BPNN*	73.76%
7	*None*	100%	60%	*BPNN*	73.73%
8	*None*	100%	60%	*SVM*	73.54%
9	*CHI*	80%	60%	*BPNN*	73.54%
10	*CHI*	80%	70%	*BPNN*	73.53%

From the overall results, we notice that the average *Accuracy* improves about 5% over the traditional classification model, which proves the effectiveness of our proposed framework. In the future, when new bug reports dataset comes, we can refer to the proposed framework to build up a most suitable bug reports assignment model.

6. Conclusion and Future Work

This paper presents a unified framework for bug report assignment. We also conduct an empirical study of automatic bug report assignment in various channels, including the tra-

ditional standardized bug reports, the crowdsourced bug reports and the mobile application testing. We found encouraging results that differentiate the tasks of bug report assignment in different channels. In the future, we would like to compare more heavyweight methods to ensure not lagging behind in the traditional way, and investigate whether more iterations will have a further impact on the classification effect. We also discuss some limitations along with threats to validity in this work, and plan to address them in our future work.

Acknowledgements

The work is supported in part by the National Key Research and Development Program of China (2016YFC0800805) and the National Natural Science Foundation of China (61472176, 61772014).

References

[1] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT*, pages 308–318, 2008.
[2] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *ASE*, pages 34–43, 2007.
[3] Ning Chen, Jialiu Lin, Steven CH Hoi, Xiaokui Xiao, and Boshen Zhang. Ar-miner: mining informative reviews for developers from mobile app marketplace. In *ICSE*, pages 767–778, 2014.
[4] Aniket Kittur, Ed H Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *SIGCHI*, pages 453–456, 2008.
[5] Di Liu, Randolph G Bias, Matthew Lease, and Rebecca Kuipers. Crowdsourcing for usability testing. *JASIST*, 49(1):1–10, 2012.
[6] Fabrizio Pastore, Leonardo Mariani, and Gordon Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *ICST*, pages 342–351, 2013.
[7] Yang Feng, Zhenyu Chen, James A Jones, Chunrong Fang, and Baowen Xu. Test report prioritization to assist crowdsourced testing. 2015.
[8] Sharad Agarwal, Ratul Mahajan, Alice Zheng, and Victor Bahl. Diagnosing mobile applications in the wild. In *HotNets-IX*, page 22, 2010.
[9] Haliyana Khalid. On identifying user complaints of ios apps. In *ICSE*, pages 1474–1476, 2013.
[10] Susan M Mudambi and David Schuff. What makes a helpful review? a study of customer reviews on amazon. com. *MIS Quarterly*, 34(1):185–200, 2010.
[11] In Amazone. Amazon. com: Online shopping for electronics, apparel, computers, books, dvds, & more.
[12] Dennis Pagano and Wiem Maalej. User feedback in the appstore: An empirical study. In *RE*, pages 125–134, 2013.
[13] Claudia Iacob and Rob Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *MSR*, pages 41–44, 2013.
[14] Laura V Galvis Carreño and Kristina Winbladh. Analysis of user comments: an approach for software requirements evolution. In *ICSE*, pages 582–591, 2013.
[15] Yohan Jo and Alice H Oh. Aspect and sentiment unification model for online review analysis. In *WSDM*, pages 815–824, 2011.
[16] Davor ČubraniĆ, Gail C Murphy, Janice Singer, and Kellogg S Booth. Learning from project history: a case study for software development. In *CSCW*, pages 82–91, 2004.
[17] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. *STEP*, page 99, 2005.

[18] John Anvik. Automating bug report assignment. In *ICSE*, pages 937–940, 2006.
[19] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *ICSE*, pages 361–370, 2006.
[20] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE*, pages 503–512, 2002.
[21] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *ICSE*, pages 465–475, 2003.
[22] Davor Čubranić. Automatic bug triage using text categorization. In *SEKE*, 2004.
[23] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL*, pages 55–60, 2014.
[24] Hua-Ping Zhang, Hong-Kui Yu, De-Yi Xiong, and Qun Liu. Hhmm-based chinese lexical analyzer ictclas. In *SIGHAN workshop on Chinese language processing*, volume 17, pages 184–187. Association for Computational Linguistics, 2003.
[25] Chris Buckley, James Allan, and G Salton. Automatic retrieval with locality information using smart. In *TREC*, pages 59–72, 1993.
[26] Avrim L Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1):245–271, 1997.
[27] Yiming Yang and Jan O Pedersen. A comparative study on feature selection in text categorization. In *ICML*, volume 97, pages 412–420, 1997.
[28] Hiroshi Ogura, Hiromi Amano, and Masato Kondo. Feature selection with a measure of deviations from poisson in text categorization. *Expert Systems with Applications*, 36(3):6826–6832, 2009.
[29] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *TPAMI*, 27(8):1226–1238, 2005.
[30] Saket SR Mengle and Nazli Goharian. Ambiguity measure feature-selection algorithm. *JASIST*, 60(5):1037–1050, 2009.
[31] Jason D Rennie, Lawrence Shih, Jaime Teevan, David R Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *ICML*, volume 3, pages 616–623, 2003.
[32] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.
[33] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.
[34] Andrew McCallum, Kamal Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48, 1998.
[35] Thomas M Cover and Peter E Hart. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27, 1967.
[36] J Ross Quinlan. *C4. 5: programs for machine learning*. 2014.
[37] Marti A. Hearst, Susan T Dumais, Edgar Osman, John Platt, and Bernhard Scholkopf. Support vector machines. *Intelligent Systems and their Applications, IEEE*, 13(4):18–28, 1998.
[38] Eric A Wan. Neural network classification: a bayesian interpretation. *IEEE transactions on neural networks/a publication of the IEEE Neural Networks Council*, 1(4):303–305, 1989.
[39] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *IJCNN*, pages 593–605, 1989.