Front. Comput. Sci.
DOI 10.1007/s11704-015-4409-2

RESEARCH ARTICLE

# Source code fragment summarization with small-scale crowdsourcing based features

**Najam NAZAR[1], He JIANG (✉)[1,2], Guojun GAO[1], Tao ZHANG[3], Xiaochen LI[1], Zhilei REN[1]**

1 Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, School of Software, Dalian University of Technology, Dalian 116621, China
2 State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China
3 Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China



**Abstract** Recent studies have applied different approaches for summarizing software artifacts, and yet very few efforts have been made in summarizing the source code fragments available on web. This paper investigates the feasibility of generating code fragment summaries by using supervised learning algorithms. We hire a crowd of ten individuals from the same work place to extract source code features on a corpus of 127 code fragments retrieved from Eclipse and NetBeans Official frequently asked questions (FAQs). Human annotators suggest summary lines. Our machine learning algorithms produce better results with the precision of 82% and perform statistically better than existing code fragment classifiers. Evaluation of algorithms on several statistical measures endorses our result. This result is promising when employing mechanisms such as data-driven crowd enlistment improve the efficacy of existing code fragment classifiers.

**Keywords** summarizing code fragments, supervised learning, crowdsourcing

## 1 Introduction

During software maintenance, software developers correct faults in the source codes to improve the performance of the system. While doing so, they either skim the code to find the relevant parts or read the related documentation. In Ref. [1] the authors indicated that the developers often read the header of a class or a method, leading comments (when available), and parameters to understand the source code. However, in most instances such leading comments are either missing (or incomplete), or headers that contain the irrelevant words that make it difficult for a developer to understand the source code clearly. Thus, for understanding the source code, developers have no other choice except to read the implementation of the source code or documentation, which in-turn requires significant time and effort [1]. Automatic summary generation of a source code overcomes this problem, leading to better understanding of a source code.

According to one estimate [2], developers spend 40% of time on web for searching relevant source code examples from source code documentations or code manuals or online resources. Current research in software engineering has mostly focused on the retrieval accuracy aspect but little on the presentation aspect of code examples, e.g., how code examples are presented in a result page or in a documentation [3]. Code examples or more formally code fragments are partial programs that serve the purpose of demonstrating the usage of an application programming interface (API) [3].

In recent years, researchers have proposed a wide variety of studies, ranging from simple text retrieval (TR) techniques (e.g., [1, 4–6]) to complex heuristic based techniques (e.g., [7]), for generating source code summaries. On the basis of available literature, we can classify these studies into four different categories. The first category includes studies

Received September 12, 2014; accepted October 15, 2015

E-mail: jianghe@dlut.edu.cn

based on text retrieval methods [1, 4–6] that generates 5–10 words summaries, while the second contains natural language processing studies such as Refs. [8–10]. The third and the fourth categories deal with code concern summaries generated through program analysis such as Ref. [7] and source to source summaries of code examples, e.g., Ref. [3] respectively.

Being new, unique, and promising, we select the fourth category, i.e., source to source summaries of code examples as our research direction. Figure 1 illustrates one code fragment taken from the NetBeans FAQ "how can I add support for Lookups on nodes representing my file type?"[1] — the generated summary for this FAQ is marked in a bold text. This code example describes the simplest way to create a mutable lookup by using InstanceContent and AbstractLookup elements of Netbeans project.

```
1: public SomeObject implements Lookup.Provider {
2:     private InstanceContent content = new InstanceContent () ;
3: private final AbstractLookup 1kp = new AbstractLookup(content) ;
5:
6:   public someMethod()
7:         ic.set (someCollection…) ;
8:   }
9:
10:     public Lookup getLookup () {
11:          return 1kp;
12:     }
13: }
```

**Fig. 1** A code fragment summary (in bold)

In this paper, we investigate the feasibility of summarizing code fragments for better presenting a code example using supervised machine learning algorithms [11]. Our approach consists of following steps. First, we collect a corpus of code fragments, containing 127 code fragments, extracted directly from the Eclipse[2] and NetBeans[3] Official FAQs (Section 3). Second, we hire human annotators to suggest summary lines, i.e., gold summary lines (annotation). Third, we introduce crowdsourcing (data-driven) as a problem solving model in software artifact summarization paradigm, as it has not been employed before for software artifact summarization, for extracting code features. Fourth, we train two classifiers namely support vector machines (SVM) and naive bayes (NB) on code fragments (Section 5). Next, we evaluate the effectiveness of these classifiers on different statistical measures such as Accuracy, Precision, Recall, F-Measure, True Positive Rate (TPR), False Positive Rate (FPR), Receiver Operator Characteristic (ROC), and Area under Curve (AUC) curves (Section 6). In the end, we perform feature selection analysis to rank and determine the importance of selected features.

Our SVM classifier outperforms the NB classifier and achieves the precision of 82%. This result is promising as summaries with this level of precision achieve the same level of agreement as human annotators with each other [3]. Furthermore, our classifiers improve the performance of the existing classifiers [3] with higher precision and accuracy.

This paper makes the following contributions:

- It reports on the creation of 127 code fragments from two well-known open source projects.
- It introduces data-driven crowd enlistment (small-scale crowdsourcing) for extracting source code features manually.
- It shows that our classifiers trained on code fragments can generate good summaries.
- It demonstrates that our classifiers outperform the state-of-the-art code fragment classifiers.
- It shows that our classifiers can improve the performance of existing code fragment classifiers.

The remainder of this paper is organized as follows. Section 2 provides overview of related work in summarizing software artifacts and how our work differs from the existing work. Section 3 provides a complete detail about corpus construction and annotation, while Section 4 discusses features extraction through crowdsourcing. Sections 5 and 6 provide details about our classifiers i.e., SVM and NB, and their statistical comparison and evaluation respectively. Section 7 concludes our paper and discusses future directions.

## 2 Related work

Summarizing software artifacts is a well researched area and wide varieties of techniques are applied to software artifacts such as bug reports and source codes. Other emerging phenomenon, such as crowdsourcing, has been effectively used in text summarization. Some of the related efforts to our work are given below.

• **Summarizing source code fragments** To the best of our knowledge, Ying et al. [3] first conducted a feasibility study, using supervised learning approach for summarizing code fragments, and focusing on presentation aspect of code

[1] http://wiki.netbeans.org/DevFaqLookupForDataNode, verified 29-07-2014
[2] http://wiki.eclipse.org/index.php/Eclipse, verified 24-09-2014
[3] http://wiki.netbeans.org, verified 29-07-2014

examples. They defined code fragments as partial programs that served the purpose of demonstrating the usage of an API. Any line in the summary is more important in the context of a query and a syntax than any other line in a code fragment, and thus a code fragment summary is shorter in size. As an initial investigation, they exploited syntactic and query features of a code fragment by applying machine learning techniques and trained SVM and NB classifiers. Using these classifiers, they achieved the precision of 71%.

Our work differs in following ways:

- We employ data-driven crowd enlistment for feature extraction in an ad-hoc way by enrolling participants from the same institution. As Ying et al. did not select features by enlisting individuals in their work, our selected features are substantially different and unique in this aspect.
- Our corpus size contains 127 code fragments extracted from Eclipse and Netbeans Official FAQs. This corpus size is larger than the corpus size in Ref. [3].
- We utilize the feature selection analysis to determine the discriminability of the selected features. It also helps us find important features and their effect on generating summary lines.
- Our SVM and NB classifiers trained on code fragments attain the precision of 82%, which outperforms the existing classifiers.

Table 1 provides a summary of differences between ours and the existing study in code fragments.

**Table 1** Summary of difference between ours and the existing work

| Our work | Existing work |
|---|---|
| Corpus size: 127 code fragments | Corpus size: 70 code fragments |
| Subjects: Eclipse and NetBeans Official FAQs | Subjects: Eclipse Official FAQs |
| Feature extraction method: data-driven small-scale crowdsourcing | Feature extraction method: unknown |
| Precision: 82% | Precision: 71% |
| Total features: 21 | Total features: 49 claimed, 17 described in study |

• **Summarizing bug reports** Other efforts in software artifact summarization have used bug reports as a repository. Rastkar et al. [12, 13] proposed supervised machine learning methods for summarizing bug report conversations. They built an extractive summary by selecting some sentences from the original bug report. Their approach employed a logistic regression classifier trained on a corpus of 36 bug reports extracted from Mozilla, Eclipse, Gnome and KDE opensource projects. They achieved the precision of more than 62% for the BRC classifier trained for bug reports.

In contrast, Mani et al. [14] employed unsupervised methods on the same set of bug reports used by Ref. [12]. They proposed a heuristic based noise reducer that automatically classify sentences into investigative, question, or a code snippet for generating better summaries. They applied four general purpose well-known unsupervised algorithms namely: Centroid [15], Maximum Marginal Relevance (MMR) [16], Grasshopper [17], and DivRank [18]. With this noise reducer, they found that the summaries generated using four unsupervised algorithms produced at least the same quality of summaries as by supervised approach. In other efforts, Lotufo et al. [19] proposed a page ranking algorithm for generating bug report summaries considering the evaluation links, titles and description similarity as important factors, on the same set of corpus used by Ref. [12].

As bug repositories contain substantial knowledge about a software development, there has been recent interest in improving the use and management of this information. For Example, correctly assigning a developer in a new bug and addressing a problem of data reduction in bug triage [20], and extracting instances from bug reports [21].

• **Crowdsourcing in text summarization** Recently, there is an increasing trend of using crowdsourcing as a model to solve issues and challenges in different domains of software engineering. However, in text summarization corwdsourcing has not been applied much. Lloret et al. [22] first utilized crowdsourcing to retrieve relevant information about place or an object in the form of sentence to create short summaries for a tourist — in the context of tourism industry. Hong et al. [23] developed a model of extracting key contextual terms from unstructured data, especially from documents, with crowdsourcing — using term selection by frequency and sentence building. In the same way, Mizuyama et al. [24] applied crowdsourcing as a mean to generate abstractive summary of a document. This clearly shows that very few efforts in text summarization have employed crowdsourcing as a problem solving model.

We are the first to explore crowdsourcing in a data-driven manner as a problem solving model on a smaller scale employing ten participants, specifically for summarizing source code, and more specifically for code fragments, consequently, which makes our study unique and innovative.

## 3 Code fragment corpus

We need a corpus of code fragments to train and judge the ef-

fectiveness of our classifiers. Optimally, such a corpus would have been made available before or created by experts in the field of code fragment summarization. Previously, Ying et al. [3] had collected a corpus of code fragments containing 70 code fragments from Eclipse Official FAQ. Since last year, additional 8 code fragments have been added to Eclipse FAQ, making the total number of code fragments to 78. As the code fragments are increased, we can not decide what 70 code fragments are employed by Ref. [3].

In view of these observations, we decide to build a comparatively larger corpus by extracting more code fragments not only from Eclipse but from NetBeans Official FAQs. Our corpus, as a whole, contains 78 code examples from Eclipse FAQ and 49 code fragments from NetBeans FAQ, making 127 code fragments in total. There are 57 additional code fragments in our corpus from Ref. [3], i.e., 8 code fragments from Eclipse and remaining 49 from NetBeans.

### 3.1 Corpus selection

We choose two authors to build a suitable corpus of code fragments. Overall, both authors have two to three years of experience in software development and working with software repositories. Besides, one of the authors has a programming experience in industry and over three years of research experience in academia.

For selecting a suitable code fragment corpus, we investigate different open source projects about the availability of code fragments. During this collection process, we become aware of that some open source projects such as Mozilla[4)] have very few code examples available on web at same location. Most of these examples are scattered over the developer's wiki on different pages which makes it harder for authors to collect. Therefore, as an initial investigation, we focus more on collecting the examples easily accessible and available at a single uniform resource locator (URL) over the web. After the extensive research, we select two projects, Eclipse and NetBeans, which have code examples available on FAQ pages, not distributed at different pages over the official websites. In addition, we also discard those code fragments which are written in programming languages other than Java such as extensible markup language (XML) or hypexText markup language (HTML).

Our corpus, in total, contains 2 262 lines of code (LOC). This corpus size is comparable to the size of corpora in other domains, for example, the BRC corpus [12] used to train a classifier to summarize bug reports, and contains 36 bug reports and 2 361 sentences. Our selected code fragments vary in length: where the longest code fragment consists of 58 LOC while the shortest has 6 LOC. Out of total 127 code fragments, 79 code fragments (62.2%) have between 6 to 15 LOC; the remaining 48 fragments (37.8%) have more than 15 LOC. Table 2 shows the distribution of code fragments w.r.t. the number of LOC in our corpus. We employ all 127 code fragments for the evaluation of summary classifiers.

**Table 2** The distribution of code fragments in the corpus

| Lines of code | Number of code fragments |
|---|---|
| 6 to 15 | 79 |
| 16 to 30 | 30 |
| 31 to 45 | 12 |
| 46 to 58 | 6 |

### 3.2 Code fragments annotation

The main reason of applying the annotation process is to extract gold summaries from the set of code fragments. For this purpose, we recruit four postgraduate students from School of Software in Dalian University of Technology, China. These annotators have three to four years of software development experience on different projects. All four annotators are currently pursuing postgraduate degrees from the same institution and possessing extensive domain knowledge in the areas of software engineering and mining software repositories (MSR).

Each annotator is assigned the whole set of code fragment corpus and we ask the annotators to select the prospective summary lines by assigning "Yes" or "No" against each line in a code fragment. After collecting the results from annotators, we assign each sentence a value from zero to four, based on the number of times the lines which have been assigned "Yes" by annotators. For each sentence the score is zero, when it has not been selected by any annotator and is four when all four annotators have selected it as a potential candidate line for summary. For each code fragment, the set of lines with a score two or more (a positive line) is called the gold summary line (GSL). For our corpus, GSLs contains 601 lines, which are 26.5% of all lines in a code fragment corpus. On average, each code fragment contains 17.81 lines, and 4.73 lines per code fragment for gold summaries. The standard deviation value of total number of lines in a code fragment is 11.28 whereas, the standard deviation value for number of lines in a summary is 2.18.

There is a common problem in annotation, i.e., the annotators usually do not agree on the same summary. This reflects

[4)] mozilla.org verified 15-01-15

the fact that summarization is a subjective process and there is no any better summary for a repository [12]. To mitigate this problem, we perform a Cohen's kappa test (K-value) [25] to measure the level of agreement among annotators. For our corpus, the kappa K-value is 0.434, showing a moderate level of agreement among annotators [26].

## 4 Features extraction using data-driven crowd enlistment

We apply crowdsourcing mechanism to extract features. As the crowdsourcing is done on a small-scale, involving ten participants only, we call our crowdsourcing process data-driven. Crowdsourcing is one of the emerging Web 2.0 based phenomenon and in recent years has attracted great attention from both practitioners and researchers [27]. It is used as a platform for connecting people, organizations, and societies to increase mutual cooperation among each other. In 2006, Howe [28][5)] first coined the term crowdsourcing and according to him crowdsourcing is based on the notion that virtually everyone can contribute a valuable information or participate in an activity online through an open call. Academic scholars from different disciplines have examined various issues and applied crowdsourcing to resolve different issues and challenges [29]. A typical crowdsourcing process works in the following way. An organization identifies tasks and releases them online to a crowd of outsiders who are interested in performing these tasks. A wide variety of individuals then offer their services to undertake the tasks individually or collaboratively. Upon completion, the individuals submit their work to an organization which later evaluates it [28, 30, 31].

The subsequent subsections explain our application of data-driven crowdsourcing for extracting source code features and also provide brief details of some of the important features.

### 4.1 Features extraction

For our study, we organize the crowdsourcing activity on a small scale in the form of an open call on web (Intranet of our institution). Altogether, ten individuals respond our call and express their willingness to participate in performing feature extraction through crowdsourcing. On average, these individuals have three to four years of software development and research experience, which is useful for understanding the code snippets and extracting the required features from these code fragments. Further, they have been provided with a detailed help document explaining the tasks and the requirements for feature extraction. We ask participants to read every line of code fragment and provide details about the features contained in every line. We set aside 50 code fragments from our corpus to motivate the participants for feature extraction through corwdsourcing. They are requested to reply back individually in seven days, with a document containing line numbers, names and reasons for selecting a specific feature from a line in a code fragment. After a week, nine out of ten individuals hand in their work (answers) the form of a document. We evaluate the answers on the basis of available literature in source code summarization as well as our experience in software development. In total, they extract 26 features from the given set of code fragments.

Our crowdsourcing platform works in an ad-hoc fashion, i.e., extracting features through an open call in a manual way by sharing data with individuals who want to participate the selection process. As crowdsourcing requires the cognitive power, which supplies through an online open call, our crowdsourcing platform fully follows the true essence of crowdsourcing.

While evaluating answers, we notice that some features are repeated i.e., either having different wordings for describing the same features or interpreted differently by participants. For instance, "throws" keyword or a "try-catch block" are selected as separate features by different participants during features extraction. As both keywords serve the same purpose, we consider these features as a single feature, i.e., "contains an exception", rather than two separate features. Similarly, "extends" and "implements" keywords deal with inheritance. We consider these keywords as a single feature as well, i.e., "extends/implements keywords".

We use the following criteria in selection of features. For example, if a line contains a "throw" keyword such as "throw new IllegalArgumentException (command)", we mark it as a feature, i.e., "contains an exception". Similarly, if a line contains "public final class AddActionActions implements ActionProvider", and participants mark it as a "public declaration of a class", we consider it a feature too. After evaluation and selection, we finalize 21 out of total 26 features provided by participants.

As stated by [3], if a line contains a certain type of syntactic construct, for instance, a line containing a try-catch block or throwing keyword is likely to be in a summary of a code fragment, whereas a line containing an if-else condition tends not to be in a summary. Thus, we compare our

[5)] archive.wired.com/wired/archive/14.06/crowds_pr.html verified 14-01-15

resulted features with the features provided by existing research [3] and find some similar features in both studies. We find that only five features are common in both studies. These common features are "type is public", "anonymous declaration", "instantiation", "contains a comment", and "contains an assignment".

As Ying et al. [3] did not provide the whole list of features, we could not compare all features with their. However, we believe that most of our features are different from the exiting study. Table 3 provides a short description of the features extracted through crowdsourcing and range of each feature.

**Table 3** Feature numbers, names, and range

| ID | Description | Range |
|---|---|---|
| 1 | Contains a class declaration | 0 or 1 |
| 2 | Contains a comment | 0 or 1 |
| 3 | Constructor call | 0 or 1 |
| 4 | Contains a method invocation | 0 or 1 |
| 5 | Contains an exception or a try-catch block | 0 or 1 |
| 6 | Contains instantiation | 0 or 1 |
| 7 | Mutator (setter methods) | 0 or 1 |
| 8 | Accessors (getter methods) | 0 or 1 |
| 9 | Assertion | 0 or 1 |
| 10 | Return statements | 0 or 1 |
| 11 | Annotations | 0 or 1 |
| 12 | Anonymous declaration | 0 or 1 |
| 13 | Part of method signature | 0 or 1 |
| 14 | Method parameters | 0,1,2,3,...,*n* |
| 15 | The length of current line of code in a fragment | 1,2,3,4,...,*n* |
| 16 | Public type signature of a method or a class | 0 or 1 |
| 17 | Assignment | 0 or 1 |
| 18 | .class literal | 0 or 1 |
| 19 | Array | 0 or 1 |
| 20 | Current location of code in a fragment | 0 or 1 |
| 21 | Extends/implements keywords | 0 or 1 |

## 4.2 Features quality

Features acquisition is an essential step in training supervised classifiers. Though manual extraction is considered time consuming and expensive, the possibility of extracting features through internet is an appealing task and typically reduces the over all cost [32]. However, it requires quality control as participants could be experts or non-experts. To achieve quality features we employ the "expert (on site)" feature acquisition strategy partly inspired by Hsueh [32]. First, we hire experts, belonging to the same institution as expertise of participants affect the crowdsourcing results [33, 34]. Secondly, we provide participants with 50 code fragments only as mentioned in Section 1. This has greatly reduced the participants maliciousness, i.e., dishonesty in performing the required task [34].

## 4.3 Features detail

We categorize 21 features into four major groups.

- *Keyword* features are related to the keywords of the source code in the code fragments. Examples include keywords such as "extends", "public", and "return" keywords.
- *Length* features are related to the length and location of a code in a code fragment. Some examples are "current location of a code in a fragment", "the length of the current line of a code in a fragment", and "part of a method signature".
- *Declaration* features are belonged to instances, declarations and assignments. Some of the features included are "constructors", "assignments", and "anonymous declarations".
- *Other* features contain features such as comments in a source code, mutators, and accessors. They further include the structure and method calls in a code fragment.

Among the selected features, parts of them are similar in the sense that they are used for referencing current or immediate objects, for instance, "this & super" keywords. Likewise, there are two more features, getting and setting keywords, dealing with accessing and mutating the values of a field. Some other features such as "public" declaration of a class or method, "the length of current line of code in a fragment", "part of a method signature" and "current location of code in a fragment" are some of the unique and different features from the existing research.

One special feature, "annotations", also becomes a part of our selected features. We consider annotations as a separate feature for the reason that annotations are special elements that provide messages to the Java virtual machine (JVM) and can be useful for the readers in understanding the source code. According to one of the individuals who have participated in the corwdsourcing mentions that only those annotations are suggested which deal with the reflection mechanism of JVM. For instance, *@ActionID*[6] taken from the NetBeans FAQ[7] deals with the retention (how long annotations are to be retained in a system) and target of a code.

[6] http://bits.netbeans.org/dev/javadoc/org-openide-awt/org/openide/awt/!ActionID.html, verified 29-07-14
[7] http://wiki.netbeans.org/DevFaqActionAddJavaPackage, verified 29-07-14

Other features such as, the ".class literal" return the instance of a class that represents the type of a class object while "class keyword" is related to class declarations. We have also analyzed the comments written in the code fragments. In code fragments, the examples are written by experts, and thus, the code fragments contain significant and expedient comments. These comments can assist in extracting extra information of the line in a code example. "Call to the constructor", "method invocation", "anonymous declaration", "method parameters", and "arrays" are the remaining features extracted for our research. Figure 2 shows an example of a code fragment taken from the NetBeans FAQ. Some of the extracted features, shown in the example are marked in bold.

```
@ServiceProvider (service=FileSystem. class)
public class DynamicLayerContent extends MultiFileSystem
    private static DynamicLayerContent INSTANCE;
    public DynamicLayerContent () {
        // will be created on startup, exactly once
        INSTANCE = this;
        setPropagateMasks (true); // permit *_hidden masks to be
    }
    static boolean hasContent () {
        return INSTANCE. getDelegates (). length > 0;
    }
    static void enable() {
        if (!hasContent ())
            try {
                INSTANCE. setDelegates (new XMLFileSystem (
                        DynamicLayerContent. class. getResource (
                        "dynamicContent. xml")));
            } catch (SAXException ex) {
                Exceptions.printStackTrace(ex);
            }
        }
```

**Fig. 2** Some selected features from a code example (in bold)

The code fragment corpus, list of extracted features, and the source code for classifiers is publicly available[8].

## 5 Summarizing code fragments

The code fragment corpus provides a basis for experimentation, which leads to producing code summary lines in turn. The 21 source code features we have extracted from code fragments are discrete variables with a binary value depending on if a line contains a feature — if a line contains a feature, we assign a value 1 otherwise 0. We conduct experiments with two separate classifiers, support vector machine (SVM) and naive bayes (NB).

We plan to investigate following two research questions (RQs):

1) Can we produce good summaries with our classifier?
2) Can our classifiers outperform the results produced by existing classifiers?

The answer to the first question will give us valuable information about the summaries generated by our classifiers and the learning method for producing these summaries. The answer to the second question aims at comparing the results generated from our classifiers with existing classifiers and baseline classifiers to evaluate the quality of our classifiers.

**Generation of summaries with classifiers** As we have only the source code fragment corpus available for both training and testing the source code fragment classifiers, we use a leave-one-out cross validation procedure.

In leave-one-out cross validation procedure, the classifier employed for creating summary lines for a particular code fragment is trained on the remainder of code fragment corpus. To form the summary, we select one code fragment from all code fragments and predict it over the classifier trained with the remaining code fragments i.e., 126 code fragments. We repeat this procedure until all fragments are trained and predicted. The SVM classifier we employ is implemented using LIBSVM toolkit[9] [35]. For implementing the NB classifier, MATLAB toolkit[10] (NaiveBayes.fit) is used. Figure 3 illustrates our code fragment summarization process. First, the corpus is created and 50 code fragments are set aside to be used for crowdsourcing activity. These 50 code fragments are passed to crowdsourcing individuals for feature extraction suggestions. Our classifiers (SVM and NB), are trained using a training set and a test set is predicted based on the training set. At the same time the annotators generate the gold summaries. The summaries generated through classifiers are evaluated using statistical measures later.

**Algorithm 1** Pseudo-code for SVM algorithm

**Input:** Code fragments $\{C_1, C_2, C_3, ..., C_{127}\}$
**Output:** Classifier model $M$
1: Initialize $C_i$: test set, $C_o$ (others except $C_i$): train set
2: **for all** $C_i$ such that $1 \leqslant C_i \leqslant 127$ **do**
3: Use method *svmTrain.main*() and $C_i$ to train the model $M$
4: Use method *svmPredict.main*() and $C_o$ to train the model $M$
5: end for
6: **return** model $M$

**Performance of classifiers** The existing code fragment classifiers we choose to investigate are trained on code fragments and developed by Ying el al. [3]. We compare their

[8] http://oscar-lab.org/CFS/, verified 31-01-15
[9] http://www.csie.ntu.edu.tw/cjlin/libsvm/, verified 24-07-14
[10] http://www.mathworks.com.au/help/stats/naivebayes-class.html, verified 29-07-14

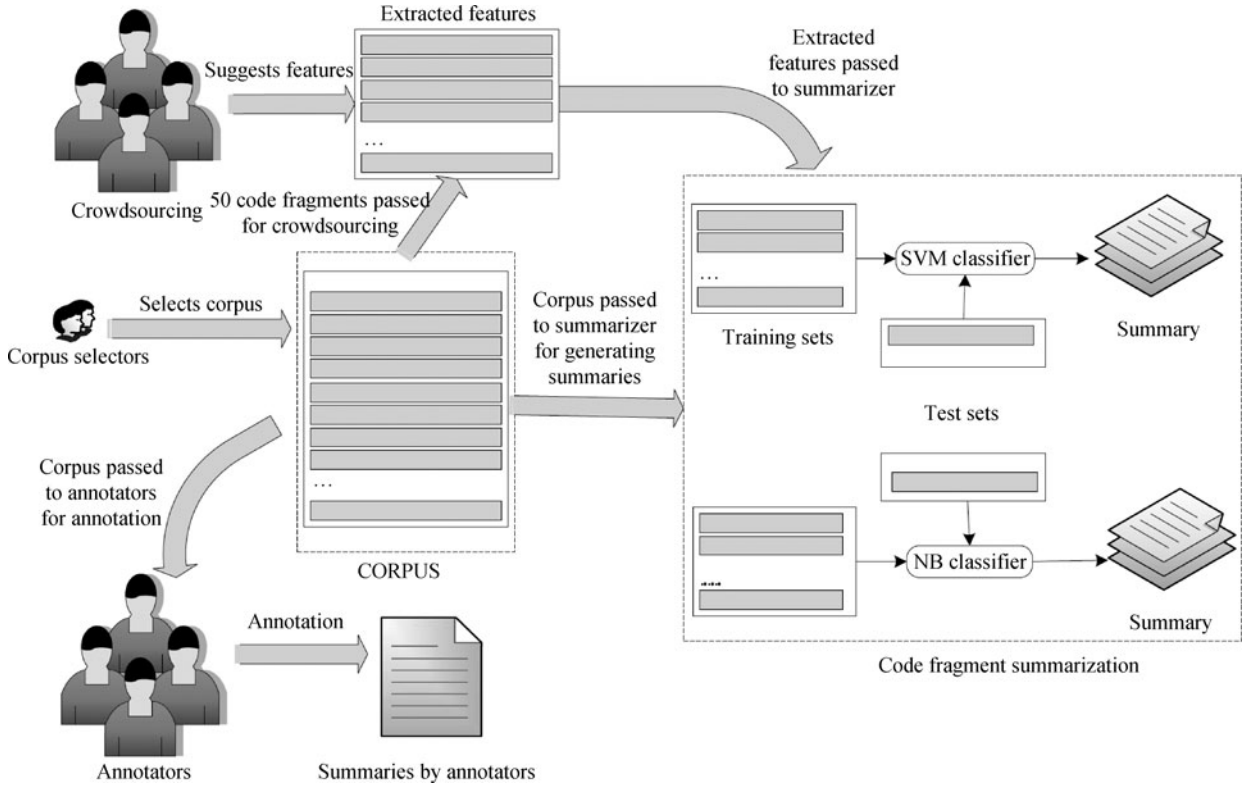



**Fig. 3** Our code fragment summarization process

performance with ours so that we can verify whether the proposed classifiers can work better.

The second classifier that we choose as a baseline is a random classifier. In a random classifier, an arbitrary choice in the form of a coin toss is used to decide the fate of candidate summary lines. The classifiers are binary classifiers, generating the values of either 0 and 1 or between 0 and 1, depending on the parameter(s) passed during experimentation. It classifies the elements of a given set into two groups on the basis of classification rule.

The results show that our classifiers trained on code fragments produce good summaries with high statistical significance. Our classifiers also outperform random classifier and improve the performance of existing classifiers presented in Ref. [3]. One of the reasons of producing good summaries is the feature selection process rather than manual feature extraction. Feature selection is a major contribution of our work. The RQs mentioned in this section are further explained in Section 6.

## 6 Analytical evaluation

In this section, we perform evaluation of our results including the research subjects described in the previous section.

### 6.1 Statistical evaluation

We perform several statistical measures to evaluate the quality of our classifiers. We achieve this by comparing these classifiers against the existing as well as random classifiers. These measures are TPR, FPR, ROC and AUC. Furthermore, Precision, Recall, and F-Measure measures are applied to analytically evaluate the effectiveness of our classifiers. More details are given in the subsections below.

#### 6.1.1 Comparing base effectiveness

We perform following comparisons in order to evaluate the effectiveness of our classifiers. The first comparison is whether the SVM and NB classifiers produce better summaries than a random classifier. In second comparison we evaluate the performance of our classifiers by comparing whether they outperform two baseline classifiers, namely "the first-N-line classifier" and "the last-N-line classifier". The first-N-Line classifier constructs a summary of length N by selecting the first N lines of a code fragment while, the last-N-line classifier picks the last N lines for summary construction. We plot the ROC curve and the AUC [36] to per-

form these comparisons.

As described in Section 5, the output of a classifier for each sentence is either zero or one. Thus, ROC measure is employed for determining the performance of a classifier at different probability threshold values of TPR and FPR. It works in a way that first, we choose a probability threshold. Next, we generate the summary lines by selecting all sentences with probability values greater than the probability threshold [13]. For summary lines generated in this manner, FPR and TPR are computed, which are then plotted as a point in a graph. In our case, for each summary, TPR measures how many sentences in GSL are actually chosen by a classifier.

The TPR of a code fragment $C$ is computed by Eq. (1) which is given as by:

$$\text{TPR} = \frac{\text{\# lines selected from GSL of C}}{\text{\# lines in GSL of C}}. \quad (1)$$

The FPR of a code fragment $C$ is opposite of TPR and is computed by Eq. (2):

$$\text{FPR} = \frac{\text{\# lines selected not in GSL of C}}{\text{\# lines in C but not in GSL}}. \quad (2)$$

Figure 4 shows the ROC curve of classifiers used in evaluation. Our classifiers, SVM and NB, are depicted with ticker lines and three baselines, random, first-N, and last-N with thinner lines on the graph. The AUC is used as a measure of the quality of a classifier [12]. A perfect classifier has an AUC value of 1, while a random classifier has an AUC value of 0.5. Therefore, a classifier should be considered effective if its AUC value lies between 0.5 and 1. The AUC values for our classifiers, SVM and NB are equal to 0.828 6 and 0.737 9 respectively. The AUC value of the first N lines is 0.6 while about the last N lines AUC value is 0.394. These values indicate that our classifiers perform exceptionally well under the given conditions. In addition, comparing our ROC curve and AUC values with the existing classifiers [3] for code fragments, it clearly shows that our SVM classifier outperforms existing ones. The AUC values for existing classifiers are 0.806 and 0.772 respectively. The AUC value of NB is much closer to the AUC value of an existing NB classifier.

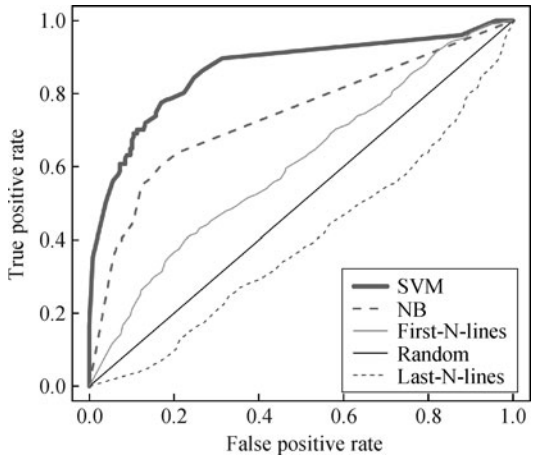



**Fig. 4** ROC plots for classifiers

### 6.1.2 Comparing classifiers

AUC is a measure of the general effectiveness of the classifiers. However, for investigating the quality of SVM and NB, when summaries are generated using predefined procedures (Section 5), we need other statistical measures to compare these classifiers. These measures are Precision, Recall, and F-Measure. Furthermore, feature selection analysis is performed in order to calculate the discrimination and importance among features.

Precision measures how often a classifier chooses a sentence from GSL and is computed as shown in Eq. (3):

$$\text{Precision} = \frac{\text{\# lines selected from GSL}}{\text{\# selected lines}}. \quad (3)$$

Contrary to Precision, Recall measures how many of the sentences present in gold summary lines, are actually chosen by the classifier. For a code fragment summary, the Recall is the same as the TPR used in plotting ROC curves (Section 1) and is computed as in Eq. (4).

$$\text{Recall} = \frac{\text{\# lines selected from GSL}}{\text{\# lines in GSL}} = \text{TPR}. \quad (4)$$

As there is always a quality compromise between Precision and Recall, being desirable but different features, the F-Measure is used as a harmonic mean to counter this problem. F-Measure can be computed by Eq. (5):

$$\text{F-Measure} = 2 \times \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}. \quad (5)$$

Table 4 shows the Precision, Recall, and F-Measure values for both SVM and NB classifiers averaged over all code fragments. These results confirm that the SVM performs much better than NB and baseline classifiers in our case study. These results also demonstrate that both SVM and NB can generate reasonably good summary lines, while SVM produce summaries with better statistical significance.

**Table 4** Evaluation measures

| Classifier | Accuracy | Precision | Recall | F-Measure |
|---|---|---|---|---|
| SVM | **76.33** | **81.75** | **59.75** | **65.92** |
| NB | 71.21 | 69.41 | 43.29 | 50.33 |
| First N Lines | 60.4 | 53.14 | 45.23 | 48.06 |
| Last N Lines | 47.13 | 34.56 | 29.16 | 31.05 |

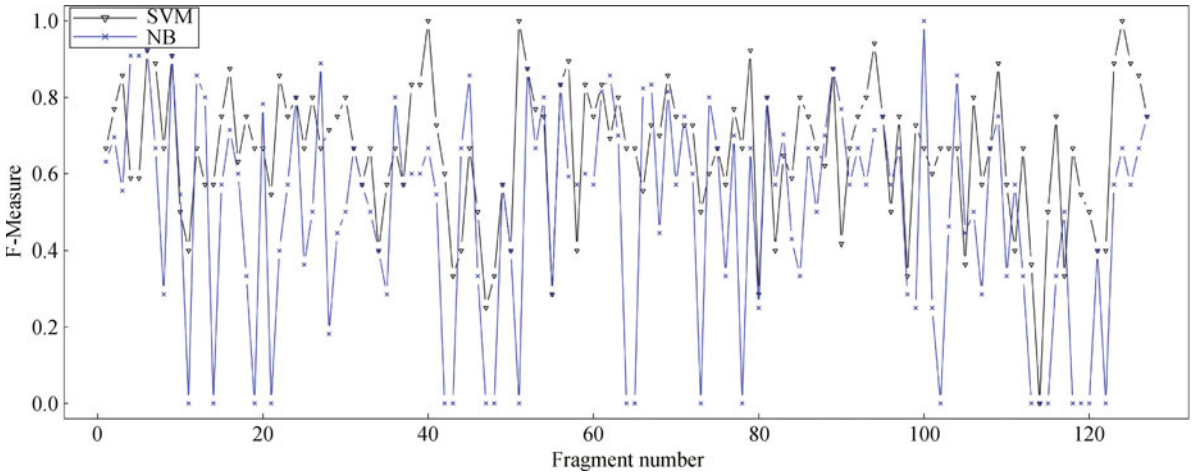



**Fig. 5** F-Measure plot for SVM and NB classifiers

Similarly, Fig. 5 shows the F-Measure values for SVM and NB code fragment classifiers. This figure illustrates that the F-Measure values for SVM are higher than that of the NB. Table 5 gives the standard deviation values for all four classifiers against every evaluation measure.

**Table 5** Standard deviation value for each classifier

| Classifier | Accuracy | Precision | Recall | F-Measure |
|---|---|---|---|---|
| SVM | 0.127 166 | 0.216 34 | 0.213 142 | 0.180 532 |
| NB | 0.155 864 | 0.364 91 | 0.279 189 | 0.284 347 |
| First N Lines | 0.131 503 | 0.224 21 | 0.154 341 | 0.172 62 |
| Last N Lines | 0.131 714 | 0.206 691 | 0.156 31 | 0.167 354 |

We believe that the performance of classifiers may be affected due to the different datasets rather than themselves. For example, Hassan et al. [37] demonstrated that NB classifier performed better than SVM in Wikitology. Therefore, it is necessary to evaluate the performance of different classifiers in our data sets. According to our experimental results, SVM shows better performance than NB classifier, and NB classifier performs better than random classifier.

## 6.2 Feature selection analysis

In our study, SVM and NB use a set of 21 features to generate summaries of code examples. The values of these features for each sentence are used to compute the probability of the sentence being part of the summary. As described in previous sections, we compute different statistical measures to investigate the effectiveness of classifiers and quality of summaries generated by these classifiers. However, these measures do not inform us of the importance of features or lines in a code fragments. There, we perform a feature selection analysis to find which features are more informative than others in generating summary lines.

### 6.2.1 Fisher Score for feature selection analysis in SVM

For this analysis, we compute the Fisher Score value for each of the 21 features, using the approach proposed by Rastkar et al. [12], which is further inspired by the Jaakkola [38] work. Fisher Score is a simple filter technique, which computes the discriminability of features in supervised machine learning [12].

First, we measure the Fisher Score to decide the best subsets for the given features. Next, we apply SVM classifier to select the final best subset across different features. The Fisher Score for feature selection analysis can be computed using Eq. (6).

$$f(i) = \frac{(\bar{x}_i^{(+)} - \bar{x}_i)^2 + (\bar{x}_i^{(-)} - \bar{x}_i)^2}{\frac{1}{n_+-1}\sum_{k=1}^{n_+}(x_{k,i}^{(+)} - \bar{x}_i)^2 + \frac{1}{n_--1}\sum_{k=1}^{n_-}(x_{k,i}^{(-)} - \bar{x}_i)^2}, \quad (6)$$

where $\bar{x}_i$, $\bar{x}_i^{(+)}$, and $\bar{x}_i^{(-)}$ are the average of the $i$th feature of the whole, positive, and negative data sets, respectively; $x_{k,i}^{(+)}$ is the $i$th feature of the $k$th positive instance, and $x_{k,i}^{(-)}$ is the $i$th feature of the $k$th negative instance. The numerator indicates the discrimination between the positive and negative sets, and the denominator indicates the one within each of the two sets. Full details on Fisher-Score are provided in Refs. [39] and [38].

This score is independent of classifiers. However, it depends on the set of features and the training data [12]. The larger the Fisher Score is, the more likely this feature is discriminative [12, 39]. Therefore, the features with high Fisher Score are more informative in determining the lines that should be included in a summary.

Figure 6 shows the values of Fisher Score computed for the features. The results show that the "the length of current line of code in a fragment" and "part of method signature"

are among the most helpful and frequently used features in code fragments. Several features such as, "calling a constructor", "public type signature of a method or a class", "accessors (getter methods)" and "anonymous declaration" are next frequently used features. These results suggest that we might be able to train more efficient classifiers by combining callers and keyword features of the source code.

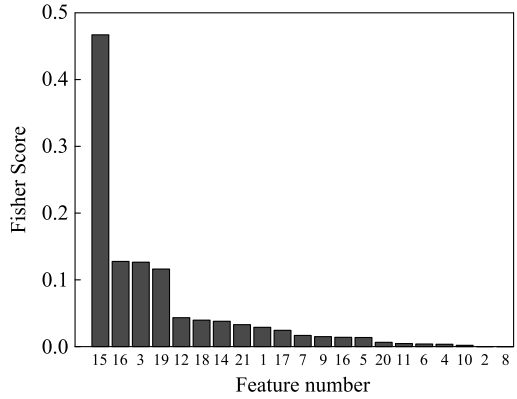



**Fig. 6** Fisher Score for features in the source code corpus

### 6.2.2 Feature selection analysis in SVM

After calculating the Fisher Score for each feature, we perform feature ranking on the basis of Precision, Recall, Accuracy, F-Measure[11], and AUC values, using SVM classifier. We perform this analysis to investigate the effect of features or subset of features on the summaries of code fragments. To rank the features, we calculate each measure in four folds and in each fold the number of features is increased by 5, i.e., $N$ = 5, 10, 15 and all, where $N$ denotes features. We select first 5 features i.e., $N = 5$ and apply to all statistical measures. This process is repeated with every fold — $N$ = 5, 10, 15, until $N$ reaches all, i.e., $N$ = all.

For $N = 5$, the Accuracy, Precision, Recall, F-Measure, and AUC values are 75%, 80%, 50%, 62% and 74%, respectively. When $N = 10$, there is a rise of approximately 6% in AUC values, while, there is a little rise in other (remaining) statistical measures. However, for $N = 15$ and all, either there is no rise for all evaluation measures or the growth is negligible. From above observations, we deduce that the first ten features are the most important and distinctive features as they have a profound influence on generating summary lines.

Figure 7 illustrates the AUC, Precision, Accuracy, Recall and F-Measure values when different parameters of $N$ are selected (i.e., from $N = 5$ to all).

In short, the "the length of current line of code in a fragment" and "part of method signature" features are most frequently used features. The results also suggest us that the first five features have keen effects on the values of statistical features than others, and thus, these features are the most important ones.

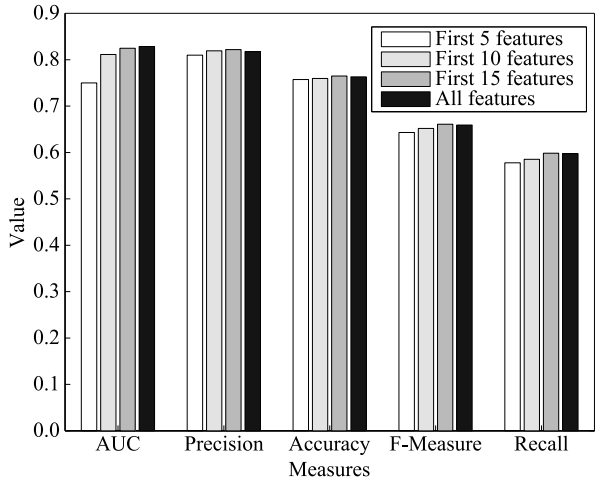



**Fig. 7** Feature ranking in the source code corpus for SVM classifier

To further evaluate the effectiveness of each feature, we calculate AUC, Precision, Accuracy, Recall, and F-Measure values for every feature without any threshold value. We find that, in this case, the feature 12 achieves the best results on all statistical measures while feature 13 is the second best. Figure 8 illustrates the AUC, Precision, Accuracy, Recall, and F-Measure values for every feature.

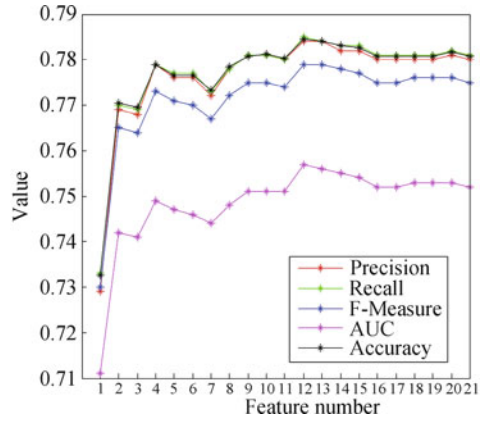



**Fig. 8** Feature ranking for each feature (no threshold value) in the source code corpus for SVM classifier

## 6.3 Threats to validity

There are four primary threats to our study, namely the size of code fragment corpus, human annotation, the participants in a data-drive small scale crowdsourcing activity and the manual feature extraction.

[11] This F-Measure is same as in Section 6.1.2

While the size of the corpus is sufficient for initial experimentation, we are limited in the size of the training set that can be used to train our classifiers. We are inaccessible to use a separate set of code fragments for training and testing. One reason is that the code fragments are not easily available at one place, rather that they are scattered over the Web at different URLs, making the collection process hard. Therefore, there is no choice except to rely on the code fragments corpus we create. We mitigate this problem by performing leave one out cross validation to maximize the validity of our results. In future, we plan to construct a corpus containing code fragments from different projects to produce more accurate results for wide variety of code fragments.

The second threat to the validity is the annotation of code fragments corpus by non-experts. Though we recruit four post-graduate students having over three years experience of software development for annotation, it might be possible that they are not experts in the programming language in which code examples are written, i.e., Java programming language. This might have distracted their attention to accomplish the task properly. Other case could be the possibility of annotators to please the experimenters. In future, we plan to reduce this risk by combining and evaluating summary results from both classifiers and humans.

The third threat could be the participants employment for crowdsourcing. It can happen that the participants contributed on our open call for feature extraction might not be the authority in specified field. As mentioned in Section 1, we discard some features which show that some of the participants are not competent enough or lacking enough knowledge to understand Java programming language to perform feature extraction properly. However, we assign fewer code fragments (50) to these participants in order to minimize the risk and encourage them to put all efforts in extracting features. In future, we consider checking the expertise of participants first, before assigning the task to reduce this risk.

As crowdsourcing requires obtaining needed services by soliciting contributors online, the process could be manual or automated. In our case, we employ individuals from the same institute for extracting features from the set of code fragments. As these individuals reside at one place, in general, they extract features manually. Though manual feature extraction is considered to be time consuming, error prone, and in some cases infeasible due to the complexity of analysis, manual effort in crowdsourcing minimizes these drawbacks. The reason is that participants have substantial experience and deep knowledge about software engineering, programming, and research. Therefore, feature selection through crowdsourcing has reduced the disadvantage of manual feature extraction considerably. In future, as an extension to this study, we are developing an automatic tool to extract features and corpus.

## 7 Conclusions and future work

Developers rely on better summaries to understand the tasks at hand in a greater depth. Software developers often depend on Web-searches looking for the source code required to understand the problem and resolve it. Code fragments are one of the most effective and frequently desired documentations, and searched on web, to assist developers in accomplishing their tasks. Generally, these code fragments are not presented well on the Web. Developer consumes a lot of time in finding and understanding required fragments on Web — generating automatic summary lines for such fragments pacifies the problem.

In this paper, we investigate the automatic generation of code fragment summary lines using data-driven small-scale crowdsourcing and supervised machine learning classifiers (SVM and NB). We find that the existing code fragment classifiers generate summaries better than random classifier with the precision of 71%. We also find that our classifiers outperform the existing classifiers with higher accuracy (76%) and precision (82%). We introduce crowdsourcing on a limited scale in the field of summarizing source code repositories and employ it as a phenomenon for extracting code fragment features. With this level of precision, our summaries achieve a high level of agreement as human annotators to each other.

This work creates new directions to improve the effectiveness of the existing systems in summarizing software artifacts, in particular code fragments, to enable developers make better use of technologies and techniques. In our future research, we plan to generate summaries through crowdsourcing on a large scale. We further plan to evaluate these summaries using different classifiers trained on code fragments.

**Acknowledgements** We would like to extend our gratitude to the individuals who dedicated their time and effort to participate in crowdsourcing activity and annotation of our code fragment corpus. This work was supported in part by National Program on Key Basic Research Project (2013CB035906), in part by the New Century Excellent Talents in University (NCET-13-0073), and in part by the National Natural Science Foundation of China (Grant Nos. 61175062, 61370144).

## References

1. Haiduc S, Aponte J, Moreno L, Marcus A. On the use of automated text

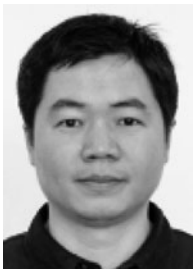
summarization techniques for summarizing source code. In: Proceedings of the 17th Working Conference on Reverse Engineering. 2010, 35–44

2. Cutrell E, Guan Z W. What are you looking for?: an eye-tracking study of information usage in Web search. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 2007, 407–416
3. Ying A T T, Robillard M P. Code fragment summarization. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013, 655–658
4. Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. 2010, 223–226
5. Eddy B P, Robinson J A, Kraft N A, Carver J C. Evaluating source code summarization techniques: replication and expansion. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. 2013, 13–22
6. Moreno L, Aponte J. On the analysis of human and automatic summaries of source code. CLEI Electronic Journal, 2012, 15(2): 2
7. Rastkar S, Murphy G C, Bradley A W J. Generating natural language summaries for crosscutting source code concerns. In: Proceedings of the 27th IEEE International Conference on Software Maintenance. 2011, 103–112
8. Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K. Automatic generation of natural language summaries for Java classes. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. 2013, 23–32
9. Moreno L, Marcus A, Pollock L, Vijay-Shanker K. JSummarizer: an automatic generator of natural language summaries for Java classes. In: Proceedings of the 21st IEEE International Conference on Program Comprehension. 2013, 230–232
10. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for Java methods. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. 2010, 43–52
11. Jiang H, Xuan J F, Ren Z L, Wu Y X, Wu X D. Misleading classification. Science China Information Sciences, 2014, 57(5): 1–17
12. Rastkar S, Murphy G C, Murray G. Summarizing software artifacts: a case study of bug reports. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. 2010, 505–514
13. Rastkar S, Murphy G C, Murray G. Automatic summarization of bug reports. IEEE Transactions on Software Engineering, 2014, 40(4): 366–380
14. Mani S, Catherine R, Sinha V S, Dubey A. Ausum: approach for unsupervised bug report summarization. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. 2012, 1–11
15. Radev D R, Jing H Y, Styś M, Tam D. Centroid-based summarization of multiple documents. Information Processing and Management, 2004, 40(6): 919–938
16. Carbonell J, Goldstein J. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. 1998, 335–336
17. Zhu X J, Goldberg A B, Gael J V, Andrzejewski D. Improving diversity in ranking using absorbing random walks. In: Proceedings of Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics. 2007, 97–104
18. Mei Q Z, Guo J, Radev D. Divrank: the interplay of prestige and diversity in information networks. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2010, 1009–1018
19. Lotufo R, Malik Z, Czarnecki K. Modelling the 'Hurried' bug report reading process to summarize bug reports. In: Proceedings of the 28th IEEE International Conference on Software Maintenance. 2012, 430–439
20. Xuan J F, Jiang H, Hu Y, Ren Z L, Zou W Q, Luo Z X, Wu X D. Towards effective bug triage with software data reduction techniques. IEEE Transactions on Knowledge and Data Engineering, 2015, 27(1): 264–280
21. Xuan J F, Jiang H, Ren Z L, Luo Z X. Solving the large scale next release problem with a backbone-based multilevel algorithm. IEEE Transactions on Software Engineering, 2012, 38(5): 1195–1212
22. Lloret E, Plaza L, Aker A. Analyzing the capabilities of crowdsourcing services for text summarization. Language Resources and Evaluation, 2013, 47(2): 337–369
23. Hong S G, Shin S, Yi M Y. Contextual keyword extraction by building sentences with crowdsourcing. Multimedia Tools Applications, 2014, 68(2): 401–412
24. Mizuyama H, Yamashita K, Hitomi K, Anse M. A prototype crowdsourcing approach for document summarization service. Sustainable Production and Service Supply Chains. 2013, 415: 435–442
25. Carletta J. Assessing agreement on classification tasks: the kappa statistic. Computational Linguistics, 1996, 22(2): 249–254
26. Cohen J. A coefficient of agreement for nominal scales. Educational and Psychological Measurement, 1960, 20(1): 37
27. Zhao Y X, Zhu Q H. Evaluation on crowdsourcing research: current status and future direction. Information Systems Frontiers, 2014, 16(3): 417–434
28. Howe J. The rise of crowdsourcing. Wired Magazine, 2006, 14(6): 1–4
29. Greengard S. Following the crowd. Communications of the ACM, 2011, 54(2): 20–22
30. Riedl C, Blohm I, Leimeister J M, Krcmar H. Rating scales for collective intelligence in innovation communities: why quick and easy decision making does not get it right. In: Proceedings of the International Conference on Information Systems. 2010, 52
31. Whitla P. Crowdsourcing and its application in marketing activities. Contemporary Management Research, 2009, 5(1): 15–28
32. Hsueh P Y, Melville P, Sindhwani V. Data quality from crowdsourcing: a study of annotation selection criteria. In: Proceedings of the NAACL HLT 2009 workshop on active learning for natural language processing. 2009, 27–35
33. Allahbakhsh M, Benatallah B, Ignjatovic A, Motahari-Nezhad H R, Bertino E, Dustdar S. Quality control in crowdsourcing systems: issues and directions. IEEE Internet Computing, 2013, 17(2): 76–81
34. Lofi C, Selke J, Balke W T. Information extraction meets crowdsourcing: a promising couple. Datenbank-Spektrum, 2012, 12(2): 109–120
35. Chang C C, Lin C J. LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2011, 2(3):

27

36. Fawcett T. Roc graphs: notes and practical considerations for researchers. Machine Learning, 2004, 31: 1–38
37. Hassan S, Rafi M, Shaikh M S. Comparing SVM and naive bayes classifiers for text categorization with wikitology as knowledge enrichment. In: Proceedings of 2011 IEEE 14th International Multitopic Conference. 2011, 31–34
38. Jaakkola T, Diekhans M, Haussler D. Using the fisher kernel method to detect remote protein homologies. In: Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology. 1999, 149–158
39. Chen Y W, Lin C J. Combining SVMs with various feature selection strategies. Studies in Fuzziness and Soft Computing, 2006, 207: 315–324

Najam Nazar received his BS degree in Computer Science from University of the Punjab, Lahore, Pakistan in 2005 and MS degree in Software Engineering from Chalmers University of Technology, Sweden in 2010. He is currently working towards his PhD degree in Software Engineering at Dalian University of Technology, China. His current research interest includes mining software repositories, data mining, natural language processing, machine learning, software product lines, and agile methodologies.



He Jiang received the PhD degree in computer science from the University of Science and Technology of China, China. He is currently a Professor in Dalian University of Technology, China. His current research interests include computational intelligence and its applications in software engineering and data mining. He is also a member of the ACM and the CCF.

Guojun Gao received his Bachelor's Degree in Software Engineering from School of Software, Dalian University of Technology, China in 2014. Currently, he is pursuing MS degree in Software Engineering from the same university. His research interests include the defects prediction, detection in software engineering.

Tao Zhang received the BE, ME degrees in Automation and Software Engineering from Northeastern University, China, in 2005 and 2008, respectively. He received the PhD degree in Computer Science from University of Seoul, South Korea in 2013. He was a research professor at the University of Seoul, South Korea from 2013 to 2014. Currently, he is a postdoctoral fellow at the Hong Kong Polytechnic University, China. His research interest includes mining software maintenance, security and privacy for mobile apps, and recommendation systems.

Xiaochen Li received the BS degree in software engineering from the Dalian University of Technology, China in 2015. He is currently a PhD candidate in Dalian University of Technology. His current research interest is mining software repositories in software engineering.

Zhilei Ren received the BS degree in Software Engineering and the PhD degree in computational mathematics from the Dalian University of Technology, China in 2007 and 2013, respectively. He is currently a lecturer in Dalian University of Technology. His current research interests include evolutionary computation and its applications in software engineering. He is a member of the ACM and the CCF.