The University of Saskatchewan
Department of Computer Science

Technical Report #2013-01

UNIVERSITY OF SASKATCHEWAN

Subjective Evaluation of Software Quality Using Crowdsource Knowledge: An Exploratory Study

Mohammad Masudur Rahman Chanchal K. Roy
University of Saskatchewan, Canada
{mor543, ckr353}@mail.usask.ca

Iman Keivanloo
Concordia University, Canada
i_keiv@encs.concordia.ca

***Abstract*—While subjective judgments have been heavily used in other areas of research including science, humanities, medical studies, and even in human computer interaction, there has been a marked lack on the use of such human factors in evaluating the quality of a software system. In this paper, we explore an idea of using subjective judgements in evaluating the code quality of a software system by using the available crowdsource knowledge of StackOverflow. We first introduce an algorithm for classifying the StackOverflow content as either promoted or discouraged programming solutions by exploiting available quantitative information such as view counts, votes, comment counts and so on. We then use the classified solutions as a vehicle for studying the code quality of open source projects based on the occurrences of promoted or discouraged solutions. Both using a cross-validation step with machine learning classifiers and a user study, we confirm that the proposed classifier is able capture the technical merits as well as social acceptance (a.k.a., subjective human judgements) of the programming solutions. In order to demonstrate the applicability of the proposed approach, we then conduct a case study with 332,628 StackOverflow posts for determining the subjective quality of 20 target open source software systems. In order to further confirm of whether such an approach makes sense we then also validate the findings with a traditional code quality evaluation tool and two project review sites. Our study shows that while such an ad-hoc nondeterministic approach by no means replaces the state of the art code quality measurements tools, it certainly has the potential to complement the existing well-rounded objective evaluation tools by incorporating a third dimension of subjective evaluation.**

***Index Terms*—Subjective evaluation; open source software;human factors;code quality; StackOverflow.**

I. Introduction

Programming is an ever-challenging problem solving adventure. Everyday, software developers deal with many programming problems and they often need to learn new technologies as a part of their work. To solve those programming problems or to learn new things, they look into different sources for helpful information [14]. It is a well-known educational psychology that a working example of the solution to a problem can be more effective for solving similar types of problems rather than writing them from scratch [14]. Also, for learning new things, working examples are more useful than any other text-based description. So, developers often reuse working code examples in their everyday problem solving and learning activities. This practice helps them to reduce the workload and makes the development process faster in one hand, but exposes the project to the risk of having bad quality software code on the other hand.

Code quality is a key factor to any software product [16]. It refers to testability, maintainability, portability, extensibility and localizability of the code [7]. Quality of software product is a visual characteristic and can be easily perceived and validated against the requirement specifications, but the quality of code is hidden and may become visible long after deployment of the product during maintenance phase [8]. The higher quality of the code, the more maintainability of the product throughout the life cycle.

Open source projects have gained much popularity nowadays and developers often reuse different modules or code fragments from them in the commercial projects. However, the quality or reliability of the open source projects need to be studied before reuse [12].

Thus, researchers are interested about analyzing the code quality issues of open source projects and making effective recommendation for reuse by the developers [12]. Existing research focuses on established code quality metrics [10], comment density [6], number of developers [15], code quality tools [12] etc. for overall quality evaluation of open source projects. As yet, software quality in general, is largely dependent on one's point of view [11]. From a developer's viewpoint the ease of modifying and further developing the software is clearly one of the most important quality dimensions. Traditionally one would call this viewpoint to quality software maintainability [13]. Unfortunately, there have been a marked lack on the use of such subjective viewpoints in evaluating software quality.

StackOverflow (SO) is a social programming Q & A site which is used by more than one million registered users (up to March, 2012) [14]. Here, users up-vote a post when they find it useful or down-vote a post if they find it not helpful for problem solving or it contains inefficient or buggy solutions. Nasehi et al. [14] study the characteristics of accepted solutions to programming questions in StackOverflow and argue that accepted solutions are very likely to contain efficient and concise code fragments accompanied by comprehensive textual description. Their study also reveals that the low-voted solutions from StackOverflow either do not contain code fragments or contain low quality code fragments. Treude et al. [17] study which type of answers are mostly accepted in StackOverflow and find that the answers containing code fragments (e.g., code review posts) are highly accepted (92%). So, it is reasonable to consider that the code fragment plays the major role behind the effectiveness of an answer post

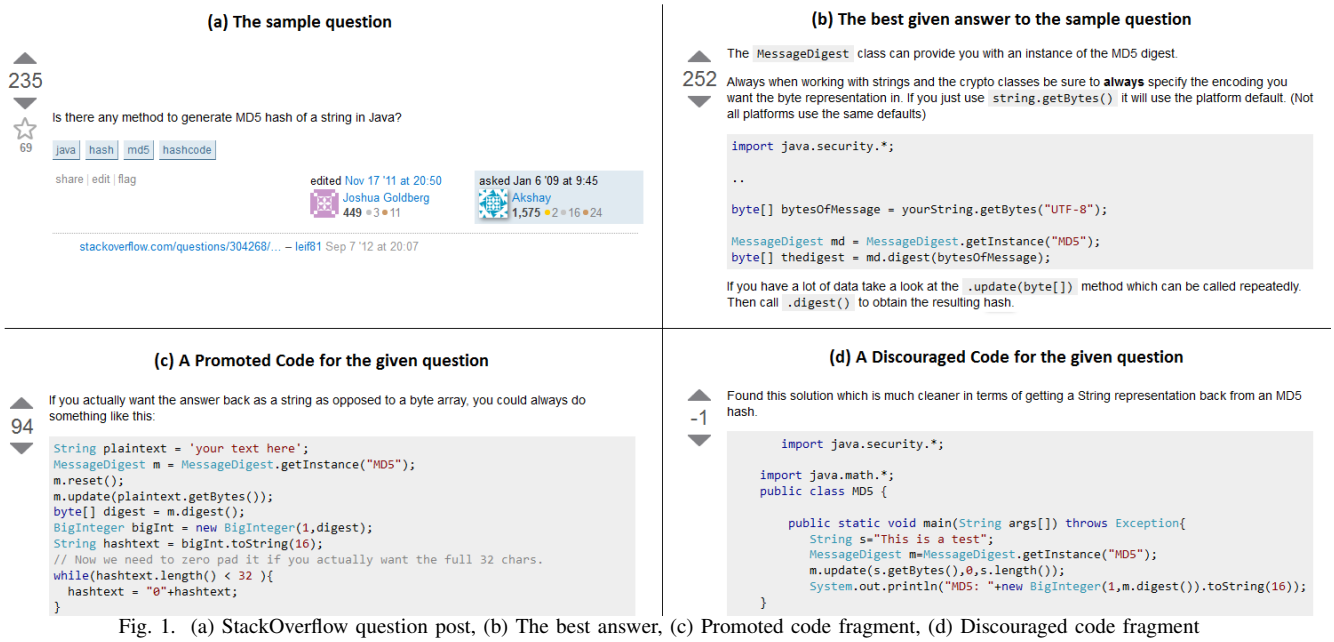

Fig. 1. (a) StackOverflow question post, (b) The best answer, (c) Promoted code fragment, (d) Discouraged code fragment

addressing a programming problem. In this study, we exploit this interesting characteristic of StackOverflow posts and then propose an algorithm to classify the code fragments as promoted or discouraged programming solutions by using available subjective judgements such as view counts, votes and so on of its large user base in the associated posts. We then use those classified code fragments as a vehicle in determining the quality of open source projects based on their occurrences in them. If a software system contains more promoted code fragments than discouraged fragments we consider that system as a healthy system, otherwise as a risky system.

We also evaluate the approach in two different ways: (1) Evaluation of the classification algorithm itself, and (2) A large case study with open source projects. In evaluating the classification algorithm, we also use two different approaches. First, we evaluate the classification algorithm by using the well known machine learning classifiers such as *J48, Naive Bayes and Bayes Net*, with an average classification accuracy of 93%. Second, we also conduct a user study with 13 graduate students and three professional software developers that partly simulates StackOverflow behaviours and confirms that the classification algorithm works well in terms of subjective judgement and that the proposed idea of using StackOverflow data makes sense. In the case study, we use the proposed algorithm for classifying 332,628 StackOverflow posts for determining the subjective quality of 20 open source software systems of different varieties and domains. In order to further validate whether the proposed approach indeed can determine whether a software system is healthy or risky, we use both an objective code quality evaluation tool, PMD [4] and two open source project review websites, namely, ohloh.net and sourceforge.net. We experienced significant correlation between our findings and the results returned by PMD. We also discovered interesting facts from the code review sites on the subject projects and noticed that those are consistent with our results. Both in the user study and in the case study we also performed extensive manual analysis in validating the findings and in deriving meaningful insights.

Our case study along with the user study show that the proposed approach can indeed aid in the subjective evaluation of a software project by using the available crowdsource knowledge of StackOverflow. Of course, such an ad-hoc nondeterministic approach by no means replaces the state of the art code quality measurements tools. However, it certainly has the potential to complement the existing well-rounded objective evaluation tools by incorporating a third dimension of subjective evaluation.

The rest of this paper is structured as follows - Section II shows a motivating example from SO. Section III reviews our approach and the proposed metrics. Section IV reports our corpus preparation and Section V evaluates the proposed classification algorithm. A case study with open source projects is presented in Section VI. Section VII discusses the previous work related to this research, whereas Section VIII presents the threats to validity, and finally, Section IX concludes the paper with our plan for future work.

II. The Motivating Example

StackOverflow posts contain invaluable information about programming problems, solutions etc. and that information can be manipulated for software development or maintenance activities. For example, Fig. 1(a) shows a StackOverflow question post asking for help *to get MD5 hash value from a string using Java*. The question scored 235, and 69 users marked it as a favourite question; that means, it is a technically important question and several users actually looked for the same information. So far there are 20 answers of varying qualities posted against the question [3]. Fig. 1(b) shows an

answer post which scored the highest among all answers and this score reflects its wide acceptability. So, it is reasonable to consider that the answer in Fig. 1(b) is the best answer for the target question. Fig. 1(d) shows an answer post having one of the lowest scores; that means, it is a potential representative of low quality answer for the question. In this research, we are interested to extract these topmost and lowest quality answers from StackOverflow to develop a code fragment corpus which we use for open source code quality prediction based on their occurrences in them.

III. The Proposed Approach

Fig. 2 shows a schematic diagram of our proposed subjective quality detection approach. We see that the approach takes both the StackOverflow posts and the target software system as input. It then apply different filtering and preprocessing of the posts having code examples before feeding them to the classification algorithm. Once the programming code examples of the question posts are classified as promoted or discouraged code fragments, the approach then detects the occurrences of these fragments in the target open source system. Depending on whether the target system contains more promoted or discouraged code fragments the approach returns whether the target system is a healthy or risky system. We, however, are more interested in finding the occurrences of discouraged code fragments in the target systems for determining their relative quality for the following reasons: (1) In StackOverflow, down-votes come from relatively reputed users (i.e., only user with *reputation>125* is permitted to down-vote) than up-votes, so, down-votes are associated with more careful judgments and technical expertise, (2) Even a single discouraged (e.g., buggy, malicious, unsafe etc.) code fragment can turn off the whole system, and (3) A discouraged code fragment has the normalized rejection rate (RR) of 0.20 or above, that means at least 20% of all voters discouraged the reuse of the code fragment. According to Pareto Principle, this is a significant portion of vote and it is assumed that 20% down-votes actually can make the 80% up-votes questionable. In the following we discuss each of the parts of the approach in details.

A. Identification of Promoted and Discouraged Code Fragments from SO Posts

Extraction of suitable code fragments from StackOverflow posts is our first contribution. In this section, we discuss about different steps to perform that task.

1) StackOverflow Code Fragment Database Development: The first step to identify encouraged and discouraged code fragments from StackOverflow posts is to extract them from the posts. Once extracted, we also perform automated and manual filtration to discard unexpected contents.

Extraction of Code Fragments from Posts: As StackOverflow facilitates the users to write the content in rich text format, the body content of the post is in HTML. We consider `<code>` tag and `<pre>` tag as the *containers* of source code snippets and look for their presence in the body content. To perform the tag checking, we use suitable regular expressions and extract the tag content.

Automated & Manual Filtration: We perform manual checking on the extracted tag content and find that those tags generally contain the source code, but they may also contain non-code elements like errors or exception messages, configuration data, URL, file path, programming keywords etc. We consider a list of heuristics to filter out the extracted content containing non-code elements.

For automated filtration, the first feature we consider is the importance of a line. We find the following lines are insignificant and cannot be valid Java statements and therefore, discard them from extracted tag content- (1) Lines start with $<, ?, \#, \$, ./$ characters, (2) Commented lines, (3) Line containing less than 20 characters, (4) Blank lines, (5) Declaration or access modifier statements e.g., line starts with import, package Java keywords, and (6) URL/file location. Once unimportant lines are filtered, we discard the unimportant code fragments applying SO post score thresholds. The idea is to select the high quality promoted and discouraged code fragments. Once the automated filtration is done, we manually investigate the filtered code and choose an optimal list of fragments for the subsequent phase.

2) Proposed Classification Metrics: The basic de facto metric for StackOverflow Post (SOP) classification is the score of a post. In addition, we propose five other feature metrics derived from the raw data about votes, vote types, page views and comments. These features capture more practical insights about the user feedback on the post content.

Acceptance Rate (AR): It refers to the ratio of total number of up-votes (UV) to the total number of votes (TV) a post gets. So, it can be considered as the probability of acceptance for a post. Different posts having same score, may not have the same the acceptance rate.

$$AR = \frac{UV}{TV}, TV > 0 \tag{1}$$

Rejection Rate (RR): It refers to the ratio of total number of down-votes (DV) to the total number of votes (TV) a post gets. This is an important metric for classification or ranking and we can consider it as the probability of rejection for a post as the answer. If two posts are having the same score but different rejection rates, then the post with lower rejection rate is considered more reliable than the other.

$$RR = \frac{DV}{TV}, TV > 0 \tag{2}$$

Code Major Answer (CMA): The metric indicates whether the code fragments within the post plays the major role behind StackOverflow post's score or not. Apparently, to determine the rationale of code fragment in a post is a daunting task, but we propose the following statistical estimation to capture the idea.

$$CMA = \begin{cases} 1 & \text{if } SC_{avg} >= ST_{avg} \\ 0 & \text{if } SC_{avg} > ST_{avg}. \end{cases} \tag{3}$$

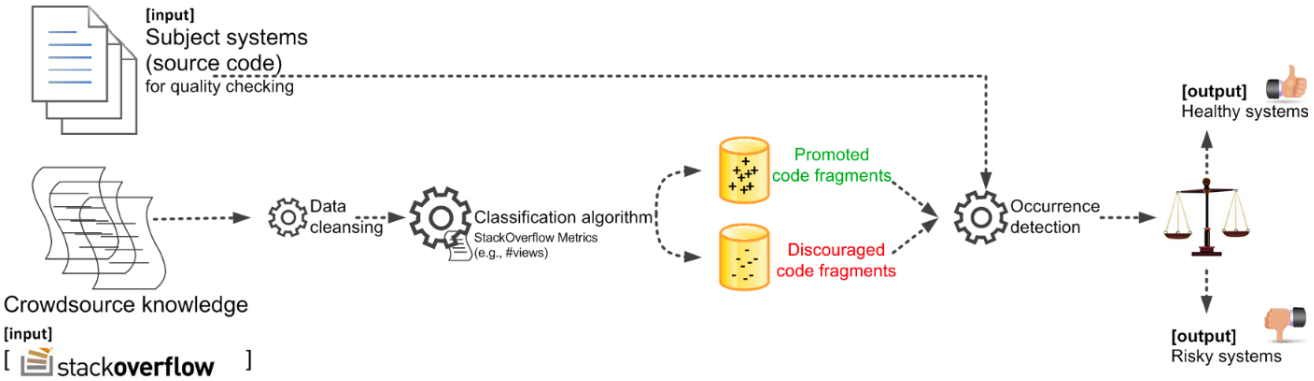

Fig. 2. A Schematic Diagram of the Proposed Subjective Quality Detection Approach

Here, SC_{avg} represents the average score of all the posts containing code fragments and answering a question, and ST_{avg} represents the average score of the set of posts not containing code fragments and answering the same question. So, if $SC_{avg} > ST_{avg}$, we consider that the corresponding set of answers depend on code fragments heavily. This metric plays an important role in SO code fragment classification.

View Count (VC): It refers to the number of times the post is viewed. StackOverflow only tracks the page view of the logged in registered users and the guest views are not considered. It is also an important metric for classification with the intuition- *the more a post is viewed, the more likely it is a good post*.

Comment Count (CC): It refers to the number of comments added against a StackOverflow post. We find a proportional relationship between score and the average comment count of StackOverflow posts; that means, a good post encourages more discussion and alternative solutions in terms of comments rather than a low quality post does.

3) Proposed Classification Algorithm: The score of StackOverflow post is a straightforward metric for classification into good or bad (low quality), but it provides less information for the classification or evaluation of the post. Thus, we use the above feature metrics (Section III-A2) for classification and break down the de facto two classes into six finer classes - *Highly Promoted* (C_1)-e.g., Fig. 1(b), *Promoted* (C_2)-e.g., Fig. 1(c), *Moderate* (C_3), *Discouraged* (C_4), *Highly Discouraged* (C_5)-e.g., Fig. 1(d), and *Unknown* (C_6).

We also use a few heuristics such as *Rejection rate threshold* (RR_0), *Up-vote threshold* (UV_0), and *Comment count threshold* (CC_0) to classify the SO posts into different classes. In case of selecting Rejection rate threshold, we use a popular heuristic called *Pareto Principle* [20] assuming that, if there are 20% down-votes out of all votes, then they can affect the rest 80% up-votes. Therefore, a post having 20% or more down-votes is considered as a Highly Discouraged (C_5) post. Moreover, we select up-vote count threshold, $UV_0 = 8$ since our study shows that average up-vote count for all posts having post *score>0* is eight. We also notice a proportional relationship between score and average comment count of posts which infers that a good post encourages more discussions or alternative solutions in terms of post comments. The average comment count for all posts is two and we select comment count threshold, $CC_0 = 2$. Algorithm 1 shows our proposed algorithm for StackOverflow Post (SOP) classification based on proposed metrics.

To complement our metrics-based classification which considers *absolute values* of selected metrics, we also take *the relative rank* of SO posts into consideration which we call *range based approach* for classification. This approach considers the following principles for classification and complements Algorithm 1 – (1) Posts having lowest 10% scores and containing code snippets are classified as Discouraged (C_4) posts. Existing studies [14, 17] suggest that code snippets are very likely to contribute to the SO post scores. Thus, when some posts get the lowest scores despite having code snippets, then the posts as well as the code snippets necessarily emerge as suitable candidates for discouraged ones, and (2) The posts having top 30% scores are classified under C_2. Due to the dynamic nature of community based evaluation, the rank of posts with high scores are likely to change, therefore, we consider top 30% posts as promoted posts (C_2) rather than choosing only the topmost. This idea can provide more robust classification for StackOverflow posts.

4) StackOverflow Code Fragment Classification: Existing studies suggest that the evaluation (e.g., votes) of a StackOverflow post is greatly influenced by the presence and soundness of the code fragments in it [14, 17]. Therefore, the user provided evaluation of the post equally implies to the quality of the code fragments as well. We classify the extracted and filter StackOverflow code fragments using the proposed metrics (III-A2) and the Algorithm 1 (Section III-A3). Thus, we get a list of promoted and discouraged code fragments from StackOverflow which we use for code quality evaluation of open source projects.

B. SO Code Fragment Based OSS Code Quality Prediction

Once we get the promoted and discouraged code fragments from StackOverflow posts, we use different available tools and techniques to find their occurrence in open source projects.

In general, we note that code fragments are unlikely to be found verbatim in the project source code except a few common answers. We consider different possible use cases

Algorithm 1 : Metrics Based SO Post Classification

```
Input: RR, VC, CC, UV
Output: C

if RR = RR_min and VC > VC_0 then
    if UV > UV_0 then
        C ← C_1
    else
        C ← C_2
    end if
end if
if RR = RR_min and CC > CC_0 then
    if UV > UV_0 then
        C ← C_1
    else
        C ← C_2
    end if
end if
if RR > RR_min and RR <= RR_0 − 0.10 then
    if VC > VC_0 or CC > CC_0 then
        C ← C_2
    else
        C ← C_3
    end if
end if
if RR > RR_0 − 0.10 and RR <= RR_0 then
    if CC > CC_0 then
        C ← C_3
    else
        C ← C_4
    end if
end if
if RR > RR_0 then
    if CC > CC_0 then
        C ← C_4
    else
        C ← C_5
    end if
else
    C ← C_6
end if
```

of the code fragments and realize that expecting the whole code fragment verbatim in the project codes is impractical and partial matching is a possible way to deal with the problem, although there is a risk of false positive results. We anticipate that clone detection tools are most likely to satisfy this goal of partial matching [18] between StackOverflow code fragments and open source projects. Therefore, we use two code clone detection tools (e.g., Nicad [9] and SimCad [18]) to find the occurrence of the SO code fragments in open source projects.

IV. SO Code Fragment Corpus Preparation

In this section, we outline the details of our SO code fragment corpus preparation.

Extracting StackOverflow Data: The first step of our experiment is collecting the data dump from StackOverflow site. We collected the most recent data (e.g., up to March, 2012) under creative common license [1]. The dumped data are a large zip file containing data of six categories - badges, posts, users, votes, post history and comments where each category has its own xml file. We worked with posts.xml and votes.xml file. While parsing posts.xml file, we found that there exist 4.5 million post entries with different tags, and we only considered the entries annotated with Java. We extracted 332,628 post entries containing 86,970 question posts and 245,658 answer posts. We also parsed the votes.xml to collect necessary information about user votes on the posts.

TABLE I
CLASSIFIED STACKOVERFLOW POSTS STATISTICS

Post Class	# Posts	Post Class	# Posts
Highly Promoted(C_1)	3278	Discouraged(C_4)	8024
Promoted(C_2)	44993	Highly Discouraged(C_5)	12419
Moderate(C_3)	263914	Unknown(C_6)	0

All information extracted was saved in a database for the manipulation in the subsequent phases.

Performing StackOverflow Post (SOP) Classification: In this phase, we used Algorithm 1 to classify the selected posts from the previous step into different classes and Table I shows the classification data. Here, we can see that 476 posts are classified as highly promoted (C_1) whereas 740 posts as highly discouraged (C_5). Table II shows the classification details of 10 StackOverflow posts along with the metrics. For example, posts with Post ID 1605332 and 331407 have the highest view counts with 100% acceptance rate. According to Algorithm 1, they are classified as highly promoted (C_1) posts. To verify the classification, we manually checked those posts in StackOverflow site and found that they are highly scored posts and marked as favourite post by more than 50 users.

After metrics based classification, we performed *range based classification* (Section III-A3) on StackOverflow answer posts. We found 3773 posts having top 30% scores and considered them as promoted (C_2) posts. We also got a collection of 2541 discouraged (C_4) answer posts containing lowest 10% scores for different questions. Finally, all classified posts constitute our corpus for further analysis and experiments.

Performing SO Code Fragment Classification: Once StackOverflow posts are classified, we used them to identify promoted and discouraged code fragments (Section III-A4). It is important to note that during code fragment classification, the metric *code major answer* plays an important role.

We extracted about 8,000 code fragments from the classified posts and performed filtration to discard unexpected non-code elements (Section III-A1). After automated and manual filtration, we got a list 2,455 SO code fragments which were likely to be selected for classification. However, to remove false positives, we manually inspected each of the code fragment and finally, got an optimal list of 1,921 code fragments. Then, we applied the proposed metrics (Section III-A2) and the algorithm (Section III-A3) to classify the code fragments into promoted and discouraged classes. Table III shows the detail statistics of promoted and discouraged SO code fragments which we used for the case study with open source projects.

V. Evaluation of the Classification Algorithm

One of the contributions of this research is to identify the promoted and discouraged code fragments from SO posts, and that is performed by our proposed classification algorithm. In this section, we present two evaluation procedures to validate the accuracy, sanity and applicability of the classification algorithm (Section III-A3).

TABLE II
CLASSIFICATION OF STACKOVERFLOW POSTS

PID	UV	DV	AR	RR	CC	VC	Class
1605332	12	0	1.00	0.00	3	5144	C_1
331407	11	0	1.00	0.00	3	3707	C_1
77213	14	0	1.00	0.00	0	2286	C_2
299555	16	2	0.89	0.11	6	0	C_3
25596	26	1	0.96	0.04	0	0	C_3
65185	45	2	0.95	0.05	0	0	C_3
1486124	2	4	0.33	0.67	10	0	C_4
343491	4	5	0.44	0.56	3	218	C_4
3772173	0	3	0.00	1.00	4	119	C_4
3315554	0	3	0.00	1.00	10	202	C_5
3616265	0	2	0.00	1.00	1	436	C_5

TABLE III
CLASSIFIED CODE FRAGMENTS (EXTRACTED FROM SO POSTS)

Fragment Class	# Fragments
Highly promoted(C_1)	70
Promoted(C_2)	841
Moderate(C_3)	138
Discouraged(C_4)	138
Highly discouraged(C_5)	596
Unknown(C_6)	106

TABLE IV
MACHINE LEARNING CLASSIFICATION OF SO POSTS

#Sample	Classifier	#Classified	#Misclassified	Accuracy
6500	Proposed	6500	0	100%
	Naive Bayes	5728	772	88.12%
	Bayes Net	6025	475	92.69%
	J48	6099	401	93.83%
13000	Proposed	13000	0	100%
	Naive Bayes	11337	1663	87.21%
	Bayes Net	12268	732	94.37%
	J48	12231	769	94.08%
18030	Proposed	18030	0	100%
	Naive Bayes	15747	2283	87.33%
	Bayes Net	17121	909	94.96%
	J48	16994	1036	94.25%

TABLE V
CODE FRAGMENT EVALUATION BY USER STUDY

		PCCA[1]		
		Promoted	Discouraged	Moderate
SCUS[2]	Promoted	3	0	0
	Discouraged	1	7	1
	Moderate	1	3	4

[1] Proposed class by our classification algorithm for code examples
[2] Suggested class from user study for code examples

A. Comparison with Machine Learning Algorithms

We started our classification with a small sized gold dataset containing labeled posts from StackOverflow using three machine learning classifiers such as *J48*, *Naive Bayes* and *Bayes net*. We used *Weka* and performed associative rule mining among the evaluation metrics and anticipated that an ad-hoc approach that could exploit certain rules was likely to perform better and therefore, we continued with the ad-hoc classification approach benefited from those machine learning algorithms. However, to validate our approach for consistency, we compared against those three state of the art machine learning classifiers with complete data and found that they can reproduce our classification results up to 93% accuracy on average. We selected three independent samples of StackOverflow posts (Table IV) and considered the proposed feature metrics of each posts for the classification. We also used a 10-fold cross validation to generalize the classification accuracy.

Table IV shows the results of our experiment. Here, we can see that Naive Bayes performs almost the same (e.g., accuracy 87%) for all 3 sample sets, whereas the classification accuracy increases consistently for Bayes net and J48 with the addition of new samples. However, it is interesting to note that these two algorithms reproduce nearly same results like our proposed algorithm, which validates that our approach is consistent with state of the art machine learning classifiers in one hand and it performs better than any single classifier under study on the other hand.

B. Validation by User Study

In the previous section, we validated the proposed classification approach using machine learning algorithms. However, in order to further confirm the applicability of the approach for the identification of promoted and discouraged code fragments, we conducted a user study. We chose 16 prospective participants (13 Graduate research students and three professional software developers) for the study. We selected five interesting programming questions and four answers (containing code fragments) for each question from StackOverflow. It should be noted that any type of StackOverflow evaluation such as score, comment, favourite count, view count etc. were hidden from the participants. The idea is to validate whether the feedback provided by our selected users matches against the decisions made by our classification algorithm or StackOverflow evaluations. At this stage, we considered three broad classes out of six finer classes shown in Table III for effective decision making about the code fragments.

Table V shows the findings of our conducted user study. In this study, each participant reviewed 20 answers under five questions and provided ratings, comments and suggestions about the answers and most importantly about the code examples contained by them. From Table V, we can see that the user study suggests 3 code fragments as *promoted*, 9 code fragments as *discouraged* and 8 code fragments as *moderate*. Interestingly, our classification algorithm works fairly well and can identify 3 *promoted*, 7 *discouraged* and 4 *moderate* code fragments respectively from the corresponding results returned by the user study. Thus, the proposed algorithm can classify 14 fragments out of 20 fragments correctly with an agreement of 70% which is a satisfactory result against the ad-hoc or subjective evaluation performed by the users. It should be mentioned that for the case study and further analysis, we used only the promoted and discouraged code fragments, and the proposed approach can identify 10 code fragments out of 11 promoted and discouraged ones which gives an agreement of 91% against the user study decisions. Thus, the finding validates the applicability of our approach to identify promoted

```
The code snippet below shows how to display n digits. The
     trick is to set variable pp to 1 followed by n zeros.
     In the example below, variable pp value has 5 zeros,
     so 5 digits will be displayed.

double pp = 10000;
double myVal = 22.268699999999967;
String needVal = "22.2687";
double i = (5.0/pp);
String format = "%10.4f";
String getVal = String.format(format,(Math.round((myVal +i)
     *pp)/pp)-i).trim();
```

Listing 1. The discouraged code example

TABLE VI
GRAPHICS AND IMAGE MANIPULATION PROJECTS

Project	# Source Files	Project	# Source Files
FidoCadj	75	Im4java	87
JavaNotelab	216	Javapeg	200
Jhotdraw7	689	JID2	53
JIU	212	JKiwi	48
Tess4j	14	TreeView	230

TABLE VII
MISCELLANEOUS OPEN SOURCE PROJECTS

Project	# Source Files	Project	# Source Files
Ant-Contrib	186	Carol	367
DNSJava	182	Jabref	305
JFreeChart	1060	DSpace	1304
JLine	30	JSch	134
Jtds	119	Jxplorer	204

or discouraged code fragments.

Regarding the 30% disagreement of our classification approach against the user study, we manually looked into StackOverflow and found that the users were not always successful to identify the correct class. For example, Listing 1 is an answer to the question *How to round a number to n decimal places in Java* which was suggested as *moderate* by the user study and our proposed approach marked it as *discouraged*. Analyzing StackOverflow, we also found that it is a *highly discouraged* code fragment containing inefficient solution that lacks generalization and consistency; but the user study failed to identify that. When we analyzed the comments from the user study, we noticed that most of them rated it as moderate because of its simplicity in the code.

VI. A CASE STUDY WITH OPEN SOURCE PROJECTS

In this section, we reported a case study with 1921 highly promoted and discouraged code fragments (Table III) from StackOverflow posts using 20 open source projects. Here, the idea is to detect the occurrences of those code fragments in the open source projects and provide useful insights such as relative subjective quality of the projects.

A. The Case Study Configuration

1) Open Source Projects: We selected 20 open source Java projects from *sourceforge.net* of varying sizes and types to conduct the study. Table VI shows 10 related open source projects for graphics and image manipulation whereas Table VII contains 10 different projects from different domains such as utility, networking, database management and digital repository management.

2) Occurrence Detection: As noted for detecting the occurrences of the SO code fragments in the open source projects we used SimCad and NiCad clone detection tools. Once the code fragments were detected from the open source projects for the given SO fragments, we investigated the results from each tool to filter out the possible false positives and then we merged the results.

B. Case Study Results and Interpretation - Project Code Quality Insights

Table X and Table XI show the statistics of the occurrence of StackOverflow fragments of Table III in different open source projects and their classification. Our target is to provide the relative subjective code quality of open source projects using that classification information and the occurred StackOverflow code fragments in the projects.

1) Result Interpretation: As noted our target is to help the developers by providing important insights into the code quality (e.g., relative quality) for a list of related open source projects so that they can take wise decision before reuse for customized or professional development. For example, let us consider two open source projects- A and B. Project A contains a few code fragments which are highly discouraged (e.g., low-voted, buggy, malicious, or fragments that do not follow standard practices) by the large crowd of StackOverflow whereas project B does not contain such occurrences. Our idea of this research is to mark project A as *risky for development and maintenance, therefore discouraged* and project B as *healthy for the developer reuse*.

From Table X, we can see that all projects except *FidoCadj*(P_1) and *Tess4j*(P_9) contain at least four discouraged code fragments each. As we know that discouraged code fragments contain inefficient, buggy, out of date programming solutions and therefore should be avoided, their occurrence in the projects demonstrates a potential threat to the over all project code quality. We note that *JHotdraw7, JIU and TreeView* contain the maximum number of discouraged code fragments and therefore, they are risky for developer reuse. On the other hand, our proposed approach for relative code quality evaluation marks *FidoCadj* and *Tess4j* as healthy for developer reuse. The basic idea is to exploit the programming community view and their subjective evaluation in the case open source project recommendation.

We further attempted to find out how generally this concept can be applied to other domains. Table XI shows the code fragments found in 10 cross-domain projects. Here, we can see that no project is free of the discouraged code; every project contains a combination of promoted and discouraged code fragments. However, *Ant-Contrib*(P_{11}), *DNSJava*(P_{13}) and *Jxplorer*(P_{20}) contain the lowest number of discouraged code fragments. So, they can be considered possibly less risky whereas *DSpace* (P_{16}) and *JFreechart*(P_{15}) hold higher risk than any other projects in the list and therefore, they are *not recommended* for developer reuse.

TABLE VIII
PROJECT SOURCE CODE EVALUATION BY PMD

Project	# TEW[1]	# ERH[2]	# ER[3]	# PMDD[4]	PAD[5]
FidoCadj	3347	28	60	Promoted	Promoted
JHotdraw7	16547	301	201	Discouraged	Discouraged
JIU	5111	144	189	Discouraged	Discouraged
Java Notelab	4983	334	162	Discouraged	Discouraged
TreeView	7688	314	185	Discouraged	Discouraged
Ant-Contrib	3347	70	110	Promoted	Promoted
JLine	758	32	26	Promoted	Promoted
Jxplorer	9340	322	428	Discouraged	Promoted
JFreeChart	29347	240	421	Discouraged	Discouraged
DSpace	31433	1144	660	Discouraged	Discouraged

[1] Total no. of error and warning messages
[2] No. of errors (high priority)
[3] No. of errors
[4] Decision by PMD based on relative quality
[5] Decision by Our Proposed approach based on relative quality

2) Summary: Fig. 3 visualizes the comparison between the occurrence of promoted and discouraged code fragments in 10 open source projects from graphics domain. We can observe that *JHotDraw7* (P_5) contains the maximum number of discouraged code fragments, whereas *FidoCadj* (P_1) and *Tess4j* (P_9) contain no discouraged fragments. Basically, each of the rest projects is having discouraged code fragments and notably more than the promoted code fragments occurred. However, we did not try to discover a correlation between promoted and discouraged occurrence counts, but according to our proposed *community view of project* idea, most of the projects enlisted in Fig. 3 are *not recommended* for the developer reuse.

Fig. 4 shows the percentage of source files affected by discouraged code fragments for those 10 Java graphics projects. Here, the idea is to demonstrate the dispersion of discouraged code fragments in the project. we can see that discouraged code fragments are more distributed in *JIU* (P_7) and *JKiwi* (P_8) than rest of the projects and therefore, they are costly for maintenance. In summary, our study shows that StackOverflow can be successfully applied for OSS code quality evaluation to (1) spot faulty or discouraged code fragments, and (2) assess the relative code quality among different projects.

C. Case Study Results Validation by PMD

We used a standard source code quality evaluation tool, *PMD* [4] for conducting an objective evaluation of the subject systems with the aim of validating our subjective evaluation for the same systems. PMD is a source code analyzer that analyzes the project code and detects possible bugs, dead code, suboptimal code, overcomplicated exceptions, duplicated code etc. We chose 10 projects from Table VI and VII and analyzed with the PMD tool. PMD produces five types of evaluation data about a project under analysis based on the well accepted rule set - (1) error (high priority), (2) error, (3) warning (high priority), (4) warning, and (5) information. For this evaluation task, we considered no. of error (high priority), no. of error and total no. of error and warning messages for a project. Table VIII shows the results of our conducted evaluation.

Table VIII contains five projects from graphics domain in the first five rows. Here, we can see that *FidoCadj* contains only 28 high priority errors whereas *JHotdraw7*, *Java Notelab* and *TreeView* contain 301, 334 and 314 high priority errors respectively such as *AvoidThrowingNullPointerException*, *AvoidUsingShortType*, *ConstructorCallOverridableMethod* etc which are detected by PMD. Now, if we consider the total number of errors and warning messages for each project, we also get the similar concept about the quality of the project. Thus, in this case, PMD based evaluation denotes that *FidoCadj* is relatively less risky for maintenance as it contains less number of errors and alarming programming constructs compared to others. Interestingly, similar type of conclusions were drawn by our case study (Section VI-B) regarding *FidoCadj* and others projects.

The last five rows of Table VIII show the PMD values of five cross-domain projects. Here, we can see *Ant-Contrib* and *JLine* contain less number of errors and warnings detected by PMD compared to other projects. *JFreeChart* and *DSpace* contain the maximum number of high priority errors which makes them *not recommended for developer reuse*. However, we found that the recommendation of our proposed approach did not match with PMD based evaluation for *Jxplorer* project. It contains 322 high priority errors which is relatively higher compared to the *promoted* projects and therefore, it is marked as *discouraged* for developer reuse. We thus attempted to find this answer during the evaluation with project review sites in Section VI-D below.

D. Case Study Results Validation by Project Review Sites

In this section, we chose two open source project review sites, namely, *ohloh.net*, *sourceforge.net* and collected evaluation information about the subject systems under study. The idea is to consider the project developers' activities and the feedback provided the users of those projects to perform an overall evaluation of the project. For *ohloh.net*, we found quite a few interesting metrics such as *Y-O-Y commit status* and *percentage of comment lines of total source code* which can be considered as the ad-hoc indicators of the source code quality of the project. Ohloh determines *Y-O-Y commit status* by comparing the total number of commits made by all developers during the most recent twelve months with the same figure for the previous twelve months. From *sourceforge.net*, we found weekly downloads count and number of recommendations made by the project users, which can be considered as the metrics for project quality evaluation. Table IX shows our case study validation results using project review sites.

From the table, we see that three projects -*FidoCadj*, *Ant Contrib* and *Jxplorer* are recommended and rest two projects are discouraged. Here, *Y-O-Y* metric reflects the current state of maintenance or development of the project, and weekly downloads is an indicator of a project's popularity. We note that projects having decreased *Y-O-Y* are also having less downloads, which denotes that without regular maintenance, the code quality or project quality degrades and thus popularity falls down (e.g., less weekly downloads). However, we note an exception in case of *Jxplorer* which has more downloads

TABLE IX
PROJECT EVALUATION BY REVIEW SITES

Project	Y-O-Y[1]	POC[2]	#WD[3]	# RC[4]	OPR[5]
FidoCadj	increasing	32%	678	51	Promoted
JHotdraw7	decreasing	48%	81	17	Discouraged
Java Notelab	decreasing	32%	97	16	Discouraged
Ant-Contrib	stable	32%	1199	50	Promoted
Jxplorer	decreasing	32%	2794	142	Promoted

[1] Y-O-Y commit status of the project
[2] Percentage of comments of the total source lines (32% is standard)
[3] Weekly downloads
[4] Total no. of recommendation by software users.
[5] Overall project recommendation

TABLE X
FRAGMENTS CLASSIFICATION IN JAVA GRAPHICS PROJECTS

Project	C_1	C_2	C_3	C_4	C_5	C_6
FidoCadj (P_1)	2	1	4	0	0	0
Im4java (P_2)	1	5	3	3	3	0
JavaNotelab (P_3)	1	2	0	3	3	0
Javapeg (P_4)	1	4	3	3	2	0
JHotDraw7 (P_5)	1	6	5	5	4	0
JID2 (P_6)	3	1	2	2	2	0
JIU (P_7)	1	3	3	4	3	0
JKiwi (P_8)	1	1	2	2	2	0
Tess4j (P_1)	1	0	0	0	0	0
TreeView (P_{10})	3	4	2	4	3	0

despite decreasing *Y-O-Y* commit status. To find a suitable explanation we looked into another metric- rating of the project and found that it was 66%. For *Jxplorer*, we found that 142 users recommended the projects whereas 73 users discouraged it. That means, the project obviously contains some issues which need to be fixed and therefore, the project is not a recommendable one.

VII. RELATED WORK

Code quality is an important and critical health indicator of software projects [8]. Software quality is a visual characteristic which can be verified against the customer's requirement specification; whereas, quality of code is a hidden attribute of software which can be perceived long after the product is delivered [8]. So, the consequences of bad quality software code are more serious and costlier in maintenance phase. Several studies try to analyze code quality issues from different perspectives. While some studies focus on identifying code quality metrics like cyclomatic complexity, line of code, function point, program dependency or control flow measure (e.g., fain-in and fan-out) [2, 7, 15, 16], others focus on developing different static code analysis tools [8, 19]. However, developer's opinion about software code quality is not recognized yet by any existing studies. In this research, we considered the opinions of a large programming community about a source code snippet and used them for code quality evaluation as a subjective measure of software quality.

Social learning site like StackOverflow plays an important role in solving programming problems and promoting good coding practices [14]. This site has social incentives such as vote, reputation, badge etc. to encourage good questions and answers. Our work is influenced by few existing researches on StackOverflow. Nasehi et al. [14] study the characteristics of a good code example and argue that highly scored answer posts are very likely to contain concise and efficient code examples. van Emden and Moonen [19] investigate which type of questions are frequently answered and discover that code-review questions are mostly answered and accepted. From these two studies, we deduce the idea that code fragment plays a major role behind the score of a post and the evaluation of StackOverflow post necessarily applies to its code fragments. Anderson et al. [5] study how the dynamics of community activities can shape the set of answer posts for a StackOverflow question over time. We also consider the dynamic nature of post evaluation by the crowd over time; therefore, we use score range based post classification besides the metrics based post classification.

Recently, open source projects are getting enormous importance against the proprietary software, but, as they are developed with a flexible and ad-hoc management style, their quality and reliability need to be studied [10]. Several studies about the quality of open source code are reported which are also of great interest to us. Gyimothy et al. [10] propose a fault prediction model for open source code using object-oriented metrics. They analyze the source code of Mozilla against the metrics to determine the fault-proneness of the code. In our research, we also propose a list of metrics related to StackOverflow post evaluation by a large programming community to get the insights into open source project code quality. Lavazza et al. [12] focus on the trustworthiness of the open source code and they propose an approach to find a quantitative relationship between the perceived quality of source code and the set of quality metrics. Our approach infers the quality and reliability of open source code based on the occurrence of promoted and discouraged code fragments from StackOverflow posts which reflects the subjective judgments in measuring software quality.

VIII. LIMITATIONS AND THREATS TO VALIDITY

As code fragments from StackOverflow are used in a modified form in the original project, we used clone detection technology to find the occurrence of the code fragments in the project though partial matching. We considered 70% - 100% similarity of project codes to StackOverflow fragment as acceptable, but the accuracy of the fragment detection largely depends on the performance of the clone detection tools. However, we used two state of the art clone detectors that give very high precision and recall [9] [18]. Furthermore, we performed manual investigation in removing false positives.

The scores and evaluations of StackOverflow posts are dynamic in nature, therefore, the classification of posts is likely to change over time. While the proposed approach can always be rerun with new data as time goes, we also attempted to address the issue during classification of SO posts using the relative rank-based classification (Section III-C).

TABLE XI
FRAGMENTS CLASSIFICATION IN MISCELLANEOUS PROJECTS

Project	C_1	C_2	C_3	C_4	C_5	C_6
Ant-Contrib (P_{11})	1	1	2	0	3	0
Carol (P_{12})	1	4	3	2	2	0
DNSJava (P_{13})	1	3	3	3	0	0
Jabref (P_{14})	2	3	4	4	4	0
JFreechart (P_{15})	1	2	4	3	7	0
DSpace (P_{16})	1	6	4	5	8	0
JLine (P_{17})	1	2	2	0	2	0
JSch (P_{18})	2	1	3	3	3	0
Jdts (P_{19})	2	4	6	1	3	0
Jxplorer (P_{20})	1	3	3	1	2	0

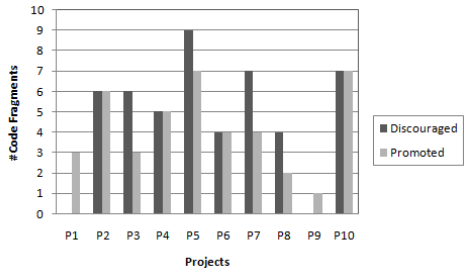

Fig. 3. Promoted and Discouraged Code Fragments in Graphics Projects

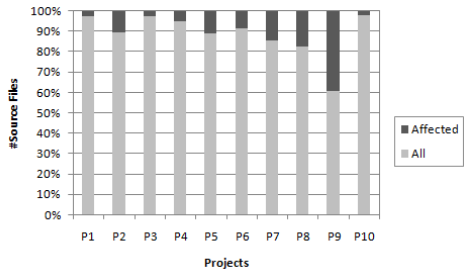

Fig. 4. Ratio of Files Containing Discouraged Code Fragments

IX. CONCLUSION AND FUTURE WORK

To summarize, in this research, we proposed an approach for SO-based OSS code quality evaluation which essentially reflects the subjective evaluation of the software projects by a large developer crowd. We evaluated the classification algorithm using well known machine learning classifiers, a user study involving 16 prospective participants and manual analysis. We then demonstrated the applicability of the approach with an example case study with 20 software systems using ∼2000 classified code fragments from SO. Our approach detected a number of discouraged and promoted code fragments in each system and evaluated the project code quality. In order to further confirm whether such subjective evaluation indeed has any values, we also validated the results of the case study using an objective code quality evaluation tool, PMD and two code review sites and experienced strong agreements between them and our results. Our study shows that the proposed approach, while not being a replacement of existing objective quality measurement tools, can indeed aid in the subjective evaluation of software systems and can be used as a third dimension to the existing tools. As an immediate future work we plan to build an Eclipse plug-in following the proposed approach.

REFERENCES

[1] Stackoverflow data dump. URL http://blog.stackoverflow.com/category/cc-wiki-dump/.

[2] Code smell metrics. URL http://c2.com/cgi/wiki?CodeSmellMetrics.

[3] Stackoverflow q & a post. URL http://stackoverflow.com/questions/415953/generate-md5-hash-in-java.

[4] Pmd. URL http://pmd.sourceforge.net/.

[5] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec. Discovering value from community activity on focused question answering sites: a case study of stack overflow. In *KDD*, pages 850–858, 2012.

[6] O. Arafat and D. Riehle. The comment density of open source software code. In *ICSE*, pages 195–198, 2009.

[7] M. Bolton. What are the useful metrics for code quality? URL http://www.linkedin.com/answers/technology/software-development/TCH_SFT/77088-3083818.

[8] Y. Bugayenko. Quality of code can be planned and automatically controlled. In *VALID*, pages 92 –97, 2009.

[9] J. R. Cordy and C. K. Roy. The nicad clone detector. In *ICPC*, pages 219–220, 2011.

[10] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *TSE*, pages 897 – 910, 2005.

[11] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, 1996.

[12] L. Lavazza, S. Morasca, D. Taibi, and D. Tosi. Predicting oss trustworthiness on the basis of elementary code assessment. In *ESEM*, pages 36:1–36:4, 2010.

[13] M. Mäntylä and C. Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Emp. Soft. Engg.*, 11(3):395–431, 2006.

[14] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example -a study of programming q and a in stackoverflow. In *ICSM*, pages 25 –35.

[15] B. Norick, J. Krohn, E. Howard, B. Welna, and C. Izurieta. Effects of the number of developers on code quality in open source software: a case study. In *ESEM*, pages 62:1–62:1, 2010.

[16] R. PloÌĹsch, H. Gruber, C. KoÌĹrner, and M. Saft. A method for continuous code quality management using static analysis. In *QUATIC*, pages 370 –375, 2010.

[17] C. Treude, O. Barzilay, and M.-A. Storey. How do programmers ask and answer questions on the web? (nier track). In *ICSE*, pages 804–807, 2011.

[18] M. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *WCRE*, pages 13 –22, 2011.

[19] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *WCRE*, pages 97 – 106, 2002.

[20] H. Zhang. On the distribution of software faults. *TSE*, pages 301 –302, 2008.