Automatically Mining Negative Code Examples from Software Developer Q & A Forums

Ryan Serva, Zachary R. Senzer, Lori Pollock, and K. Vijay-Shanker
Computer and Information Sciences
University of Delaware, Newark, DE 19716
Email: {rserva, zsenzer, pollock, vijay}@udel.edu

***Abstract*—In addition to learning good practices and reusing code from mining code examples, programmers can be supported in their learning and code improvement processes through negative, or poorly written, code examples. While it is challenging to identify negative code examples automatically from within source code, we leverage a key insight that the natural language in questions that include code examples posted on forums can provide adequate clues. In this paper, we describe an automatic sentiment analysis-based technique for mining negative code examples from developer question and answer forums along with a technique to automatically mine negative sentiment indicators commonly used by developers, which are used to drive the sentiment-based technique.**

I. Introduction

Programmers learning how to use a programming language or an API often rely on code examples to support their learning activities [1], [2], [3]. They also often seek code examples to reuse for their current project. Thus, researchers have developed techniques for automatically mining code examples from various sources [4], [5], [6].

While it is well-known in educational psychology that good examples are an efficient means of learning new concepts [7], most people also learn from bad, or negative, examples. Similarly, in programming, code examples known to exhibit bad behavior can make good teaching tools. In the framework of teaching and learning styles for core computer science curricular components as presented by the Pedagogical Patterns Project, many of these patterns focus on the benefits of learning from negative examples [8]. For example, in the "Mistake" pattern, "students are asked to create an artifact such as a program or design that contains a specific error. Use of this pattern explicitly teaches students how to recognize and fix errors.

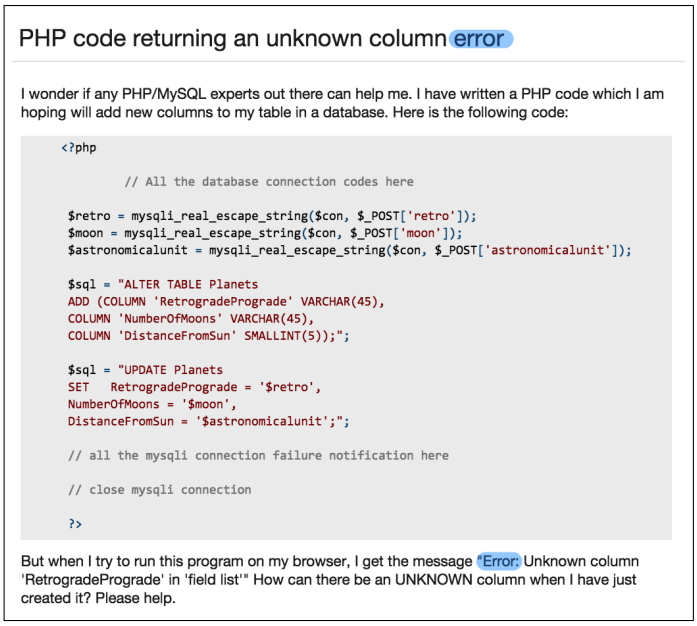
In addition, as a programmer is developing code within an IDE, it would be helpful to recognize poorly written code segments and notify the programmer. Similarly, one could build a tool that scans a project, identifies poor code segments and suggests improved replacements. However, these uses of negative code examples all rely on identifying negative code examples.

Poorly written code is plentiful; almost every project has some code segments that could be improved for efficiency, readability or other properties in addition to potentially buggy code. Current techniques for mining code examples for learning or reuse do not rate the quality of that code segment or distinguish between good and bad examples. These techniques can increase the chances that the code example is good for learning or reuse by mining from sources they believe to be overall good quality. There exist many analysis techniques for judging the quality of code, including computing various metrics such as cyclomatic complexity and code smell identification, but these techniques are not always applicable to small code segments used as examples.

In this paper, we present an automatic technique to mine bad, or negative, code examples, in the form of poorly written code segments. Throughout this paper, we consider *a negative code example to be a code segment that has at least one of the following problems: throws an error/exception, has a data race, terminates normally but does not achieve its intended functionality (semantic/logic error), uses incorrect or deprecated methods, is extremely inefficient, is insecure, or has a platform/portability problem.*

Instead of mining from code projects which would require then determining the quality of the mined example, we leverage the insight that question and answer (Q & A) forums such as Stack Overflow often include negative code examples. Specifically, it is common for a question to contain the code segment that the questioner is having problems with, or wants help with replacing with a better strategy. The question will be followed by answers that contain suggested better ways to perform the same task. One could assume that every code segment appearing in a question is a negative code example; however, our analysis of Stack Overflow revealed that this is not always the case, and this could lead to mining negative code examples that are indeed positive, or good, code examples. Thus, our approach applies sentiment analysis to text in the questions containing the code segments presented in the Q & A forum.

Sentiment analysis, also called opinion mining, is the analysis of natural language text to characterize the author's opinion of the document, sentence, or object being discussed [9]. An opinion is a positive, negative, or neutral sentiment, view, emotion, or attitude toward some entity or object. Our hypothesis is that the sentiment of the question author is an indicator of whether a code segment in the question is a negative code example. One could use a general purpose sentiment analyzer, such as Python NLTK text classification, or to improve the accuracy, train the sentiment analyzer on text from the intended target documents, Stack Overflow entries [10]. In our initial manual analysis of Stack Overflow entries, we observed that there are many words and phrases, such as 'infinite loop' and 'keep getting', that indicate negative code examples, but are not identified as negative sentiment indicators by general pur-



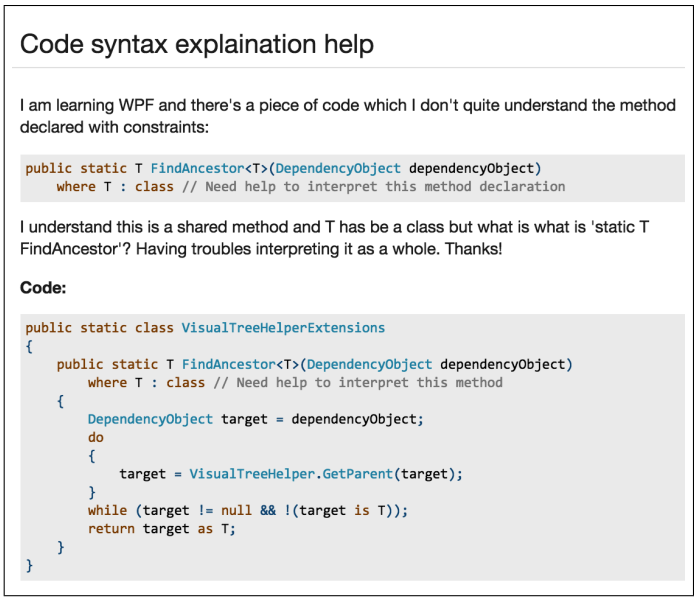
PHP code returning an unknown column error

I wonder if any PHP/MySQL experts out there can help me. I have written a PHP code which I am hoping will add new columns to my table in a database. Here is the following code:

```
<?php

        // All the database connection codes here

$retro = mysqli_real_escape_string($con, $_POST['retro']);
$moon = mysqli_real_escape_string($con, $_POST['moon']);
$astronomicalunit = mysqli_real_escape_string($con, $_POST['astronomicalunit']);

$sql = "ALTER TABLE Planets
ADD (COLUMN 'RetrogradePrograde' VARCHAR(45),
COLUMN 'NumberOfMoons' VARCHAR(45),
COLUMN 'DistanceFromSun' SMALLINT(5));";

$sql = "UPDATE Planets
SET   RetrogradePrograde = '$retro',
NumberOfMoons = '$moon',
DistanceFromSun = '$astronomicalunit';";

// all the mysqli connection failure notification here

// close mysqli connection

?>
```

But when I try to run this program on my browser, I get the message "Error: Unknown column 'RetrogradePrograde' in 'field list'" How can there be an UNKNOWN column when I have just created it? Please help.

Fig. 1: Negative code example within Stack Overflow question.



Code syntax explaination help

I am learning WPF and there's a piece of code which I don't quite understand the method declared with constraints:

```
public static T FindAncestor<T>(DependencyObject dependencyObject)
    where T : class // Need help to interpret this method declaration
```

I understand this is a shared method and T has be a class but what is what is 'static T FindAncestor'? Having troubles interpreting it as a whole. Thanks!

Code:

```
public static class VisualTreeHelperExtensions
{
    public static T FindAncestor<T>(DependencyObject dependencyObject)
        where T : class // Need help to interpret this method
    {
        DependencyObject target = dependencyObject;
        do
        {
            target = VisualTreeHelper.GetParent(target);
        }
        while (target != null && !(target is T));
        return target as T;
    }
}
```

Fig. 2: Code example (within Stack Overflow question) that is not a negative example.

pose sentiment analyzers. Thus, we developed a technique to automatically mine negative sentiment indicators for software-related Q & A forums to use for mining code examples, and we compare using the customized sentiment indicators versus a general purpose sentiment analysis in our evaluation.

The key contributions of this paper include:

- a sentiment analysis-based technique for mining negative code examples,
- a technique to mine negative sentiment indicators particularly used by software developers from software-related discussions, and
- an evaluation and qualtitative analysis of the automatic negative code example mining technique using precision and recall for a set of 240 human-judged Stack Overflow entries containing code segments.

II. Motivating Examples

The Stack Overflow question in Figure 1 exemplifies a negative code example; this code throws an error upon execution. Without looking carefully at the code, a reader would guess that there is a problem based on the preceding text that contains the phrase "help me". In addition, the text following the code contains the word "error" and the phrase "please help". These words and phrases are examples of negative sentiment indicators. In a PHP course, this code could be used in a lesson on adding columns. Being able to recognize how to avoid receiving unknown column errors will be very beneficial in the learning process. From this example, the consequences of using incorrect quotations and parenthetical order are able to be evaluated and avoided later on during the coding process. Also, from this example negative code segment, an IDE could detect that the wrong type of quotations are being used when adding the column, prompting the user that the current code will result in an unknown column error.

The code in Figure 2 shows another Stack Overflow question that contains a code segment; in this case, not a negative code example. Note that it can be inferred from the text that the questioner is asking for help in understanding this code. Upon observation, this code is not preceded or succeeded by any text that one would consider a negative sentiment indicator.

Thus, while there are many code examples that appear in Stack Overflow and other Q & A forums, naively mining all code examples from questions that contain a code example and classifying them as negative examples can lead to inappropriate mislabeling. Our hypothesis is that the clues left in the sentiment of the natural language text of the question can be strong indicators of whether a code segment is poorly written.

III. Approach

With the increased use of online blogs, forums, news and other documents such as movie and product reviews and travel forums, has come the increased use of automatic techniques for identifying subjective versus objective text and distinguishing between positive, negative and neutral opinions. These techniques, called sentiment analysis or opinion mining, are based on natural language processing and the use of sentiment cues.

Our main goal is to explore ways to utilize negative sentiment indicators to use the opinion of the questioner to classify code segments in Q & A forum questions as negative or otherwise (i.e., non-negative). Our evaluation (see Section IV-B) combined with our manual observation of developer Q & A forum question text provides evidence that more precision should be achievable by developing a set of negative sentiment indicators customized for our goal of mining negative code examples from software developer forums. Instead of manually developing a set of customized negative sentiment indicators, we took a data-driven approach, in which we mine them automatically. The first subsection describes our methodology for automatically identifying negative sentiment word or

phrase indicators for the purpose of identifying negative code examples in developer forums. The second subsection explains how to utilize those indicators to classify code segments in forum questions as negative code examples.

A. Mining Negative Sentiment Indicators

Figure 3 depicts the phases of mining negative sentiment indicators related to code examples. The input is a set of software artifacts, and the output is a set of negative sentiment indicators for mining opinions on code segments. We follow a process shown to be successful in mining informative terms associated with genes by comparing the frequency of occurrence of a term in the gene's biomedical literature (query document set) to its frequency of occurrence in documents about genes in general (background document set) [11]. We first collected a set of software artifacts to serve as the *query document set*, which we believe should contain more natural language words and phrases indicative of negative sentiment toward a code segment. Similarly, we collected a set of software artifacts to serve as the *background document set*, which we believe should contain more general natural language related to software, but not particularly with negative sentiment. The hypothesis behind the approach to mining negative sentiment indicators is that words and phrases that occur more frequently in the query set when compared to the background set will also occur with negative code examples in forum questions.

The query and background document sets are processed similarly through the next few phases. We extract and stem all n-grams, currently bigrams and unigrams. While we could focus on mining single words (i.e., unigrams), our manual observations of Stack Overflow text entries motivated us to also mine collocations in the form of bigrams that correspond to some conventional way of expressing a concept where the words together have added meaning beyond the words taken individually, e.g., infinite loop.

For each extracted n-gram, we compute its frequency of occurrence in the query set and the background set, separately. Stop words are removed to increase precision. At this point, we have a set of n-grams with frequency information for both the query set and background set. We then compute a term score to compare the frequencies of occurrence of each n-gram in the two sets. We rank the n-grams based on the score and then categorize the n-grams as negative sentiment indicators, or not, using a threshold. We describe each phase in detail in the remainder of this subsection.

Query and Background Document Sets. The first design decision is what set of software artifacts would be most likely to contain negative sentiment indicators for code examples, i.e., the query document set. We chose the query document set to consist of a set of 150,000 Stack Overflow questions that contain any code tag, indicating that the questioner included at least one code segment. The intuition is that any question containing code is most likely asking for advice about that code, and thus the questioner believes it can be improved. The query document set was drawn from 250,000 questions.

Initially, we used language-specific bug reports for C++, Java, and Python as the query set. We expected that the text contained in bug reports would discuss problematic code features and would be suggestive of negative code examples. Our initial experiments indicated that the bug reports as a query set were very specific to the programming language, project identifiers, and structures used in the associated projects, and the mined negative sentiment indicators included considerable noise not easily filtered out. Another potential query set is the set of Stack Overflow questions that contain the keywords "error", "fix", or "broken". We observed that using such a query set did not actually expand to a good set of negative sentiment indicators, as it limited the scope of the set of documents included in the query set.

The next design decision is what set of software artifacts could serve as documents that contain general words and phrases, without an abundance of negative sentiment indicators, i.e., the background document set. The background document set consists of 100,000 Stack Overflow questions that contain no code tag, thus no code examples are explicitly denoted by the questioner. These were drawn from the same 250,000 questions that the query set was drawn. We hypothesized that questions without code segments are mostly asking questions that are not specific to code segments and thus contain more general terms and few negative sentiment toward code. While we observed that not all questions with code segments contain negative code examples, our intuition is that with large enough query and background sets, we can mine the negative sentiment indicators with these sets.

Initially, our background set was taken from language-specific programming documentation for C++, Java, and Python. However, we found that the background set was not satisfactory, as many words from the background set also had a high frequency in the query set. Documentation had few general terms beyond programming-specific words, which then showed up in both query and background sets. Another potential background set is the set of questions that contain the expression "how to", under the intuition that questions containing "how to" are asking for explanations as opposed to debugging help. We found that the "how to" documents for the background set did not contain exclusively explanatory code and thus did not serve as a good background set with mostly general terms.

Extracting and Preprocessing Terms. Once the query and background sets are collected, we extract the bigrams and unigrams from each document set by extracting terms from the natural language segments of the whole question post. We ensure that only the question post natural language text is considered as part of the query and background sets, including no moderator text or code segments as they skew results.

The natural language text portion is input to a natural language splitter, tokenizer and stemmer to extract individual terms. Instead of considering words as the unit, we use lexemes, which are formed by grouping words with the same stem because they typically express the same concept. To identify the lexemes, we use a homegrown morphological processor that accounts for different inflections using the methodology described in Miller et al. [12].

We extract both bigrams and unigrams. As mentioned earlier, while unigrams catch the single-word terms such as "error" and "wrong", we found that many phrases that indicate negative sentiment are actually two-word phrases.

After analyzing the word rankings from these data sets,

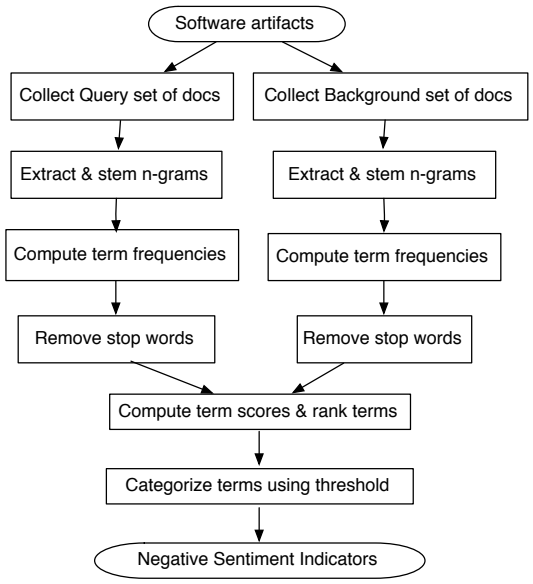


Fig. 3: Process of mining negative sentiment indicators.

we observed that some terms were occurring often in both the query and background sets, including general words such as "the", "is", and language and implementation-specific words, such as "java", "php", "sql", "database", "jframe". We created a stop word list and removed them in the extracted n-gram sets from the query and background document sets.

Scoring and Categorizing. The next step is to compute the frequency of occurrence of each term in the query set and background set, separately. For each term, we compute a score that contrasts the frequency of occurrence of the term in the query set with the frequency of the term in the background set. Again, following the success of the eGift project's scoring mechanism for contrasting document sets [11], our score $s(t)$ is computed as follows:

$$s(t) = (ntf_{query}(t) - ntf_{bkgd}(t)) * ln(\frac{1}{ntf_{bkgd}(t)})$$

where $ntf_{query}(t)$ and $ntf_{bkgd}(t)$ are the normalized term frequencies of occurrence of the term t in the query and background sets, respectively. To compare the frequency of occurrence of a term in the two sets, we prefer subtracting the normalized frequency to finding their ratio. We found that taking the ratio overemphasized the terms that occurred infrequently in the background (and might not even be related to the issue). Since the number and size of documents in the query set can vary from the number and size of documents in the background set, the frequencies are normalized. Following a well accepted technique for term frequency normalization, we normalize the term frequency weights by the most frequent term in the document set [13]. The .5 is a smoothing term used to dampen the second term to scale down the tf by the largest tf value in the document set. The goal is to mitigate the anomaly of higher term frequencies in longer documents because longer documents tend to repeat the same words. Thus,

$$ntf_{query}(t) = .5 + .5 * (\frac{tf_{query}(t)}{tf_{query}(mostfrequentterm)})$$

The scores are used to rank the terms in order of decreasing scores. We use a threshold to categorize all terms with a score above the threshold as negative sentiment indicators. Our evaluation study includes an analysis of threshold values to determine a good threshold based on precision and recall results. In our evaluation, we found that using a threshold for unigrams did not result in good performance, and thus we manually selected a subset of unigrams, produced from the training data, from the set of terms that scored above the threshold.

B. Sentiment-based Mining of Negative Examples

With an established set of negative sentiment indicators, we can mine negative code examples from Stack Overflow questions. The Stack Overflow questions are scanned for negative sentiment indicators in the natural language segments. Two possible parameters to explore are (1) the frequency that negative sentiment indicators must occur and (2) the location where they must occur to categorize a code segment as a negative code example. We evaluated various minimum frequencies that negative indicators need to occur in a Stack Overflow question to classify its contained code example as negative, and considered the code example to be negative if the negative indicators occur anywhere in the natural language text of the question.

We evaluated many configurations categorized into two main approaches to classify an entry as containing a negative code example: (1) the single-occurrence scheme: any negative indicator (i.e., bigram or unigram in Table I) occurs at least once in the question (with varying thresholds on the maximum ranking of the bigrams to be considered), and (2) the scoring-based scheme: different variations of scoring each entry such that occurrences of higher ranked bigrams in an entry contribute more to the score than lower ranked bigrams, where the bigram rankings are shown in Table I. The threshold to be considered a negative code example is a score of at least 100, which was empirically established through threshold analysis. For approach (1), Table II first two columns show the configurations based on bigram rank threshold and with or without unigrams considered. For approach (2), Table III first five columns show the configurations of ranges of bigram rankings for the weights of 100, 75, 40, and 25, respectively, for the variations evaluated, with and without unigrams. An entry that contains a unigram in the table has its score increased by 100, and thus is automatically within the threshold, again determined empirically.

IV. Evaluation

We designed our evaluation to answer two main research questions with both quantitative and qualitative analysis:

RQ1: How do different approaches to mining sentiment indicators and automatic mining of negative code examples impact the effectiveness of the automatic negative example miner? What is the best configuration?

RQ2: How effective is our automatic mining technique in identifying negative code examples?

To answer RQ1, we focus on answering:

TABLE I: Mined sentiment indicators for negative code (frequency in data set: frequency in posts humans identified as negative)

Bigrams					Unigrams	
1. doing wrong (2:2)	2. follow error (0:0)	3. work fine (11:8)	4. error message (2:2)	5. work perfect (0:0)	1. error (25:23)	2. trying (49:37)
6. please ask (0:0)	7. fully address (0:0)	8. asked before (0:0)	9. user input (2:1)	10. first time (2:0)	3. wrong (8:8)	4. getting (10:7)
11. keep getting (0:0)	12. everything work (1:0)	13. miss something (0:0)	14. going wrong (1:1)	15. even though (2:2)	5. except (4:3)	6. expect (6:3)
16. better way (2:1)	17. something wrong (0:0)	18. shown below (0:0)	19. loop through (0:0)	20. right now (4:2)	7. null (2:2)	8. doing (14:10)
21. follow line (0:0)	22. correct way (1:1)	23. doing something (1:1)	24. much appreciate (0:0)	25. work correct (0:0)	9. work (61:48)	10. fine (15:12)
26. data frame (0:0)	27. someone help (1:0)	28. two table (1:1)	29. look something (0:0)	30. radio button (1:1)	11. something (11:8)	12. try (18:16)
31. syntax error (0:0)	32. user click (2:1)	33. same error (0:0)	34. base class (0:0)	35. function call (0:0)	13. compile (4:2)	14. remove (10:6)
36. great appreciate (2:1)	37. make sure (2:1)	38. follow except (0:0)	39. expect result (0:0)	40. submit button (0:0)	15. argue (2:2)	16. syntax (2:0)
41. return value (0:0)	42. last line (0:0)	43. end up (2:2)	44. work great (0:0)	45. regular express (0:0)	17. throw (3:2)	18. empty (8:5)
46. follow command (0:0)	47. second one (0:0)	48. public class (0:0)	49. work well (0:0)	50. view model (0:0)	19. unfortunate (2:2)	20. trouble (4:4)

(1) In the single-occurrence scheme where a negative sentiment indicator needs to only occur once in a question to classify the contained code example as negative, how does the threshold T for the ranking of the highest T scoring bigrams to be considered affect the precision and recall of the negative code example miner?

(2) How effective is a negative code example miner under different configurations of the scoring-based approach with weighting the bigram occurrences based on their ranking?

(3) How does including manually selected unigrams as indicators affect the results in both the single-occurrence and the scoring-based schemes?

In addition to quantitatively measuring precision and recall for the negative code example miner to answer RQ1, we address the following subquestions for answering RQ2:

(1) Given the best configuration examined, how does the precision and recall for negative code examples compare with using a general purpose sentiment analysis based approach?

(2) When our system is not effective, what is the breakdown between mined negative examples that humans judge to be non-negative and missed negative code examples?

(3) What are the characteristics of incorrectly classified code examples and missed negative code examples?

A. Experiment Design

Subjects, Variables, and Measures. The subjects in our study are 240 questions that contain code tags and are randomly selected from 150,000 Stack Overflow questions (not used in our development work) across multiple domains. If a randomly chosen question was manually deemed inadequate based on the use of code tags for keywords or other text in a sentence, it was replaced with an alternative, randomly chosen question.

The independent variable is the negative code example mining strategy, including the different approaches to mining sentiment indicators and categorizing code segments as negative examples. We implemented the different configurations of our sentiment-based approach: single-occurrence with different thresholds for ranked bigrams, and scoring-based with different weighting schemes, both approaches with and without the 20 unigrams selected from top-scored unigrams in our mined negative sentiment unigrams.

The dependent variable is the effectiveness of the techniques. We measure the effectiveness in terms of precision and recall. Precision is calculated by determining the percentage of mined negative code examples that are actually negative code examples as judged by human judges. Recall is computed by determining the percentage of all the actual negative code examples in the study (as identified by human judges) that are mined as negative code examples by the automatic miner.

Methodology. To obtain the ground truth for computing precision and recall, we recruited twelve PhD students in computer science as human judges, who have no knowledge of our techniques, are not authors on this paper, and who have prior computer science and programming experience. Each of the twelve judges was given the URLs for 40 of the randomly selected Stack Overflow questions. The human judges were given the whole entry so they had all the text surrounding the code example to make their judgement, as we were interested in the ground truth. For each question, the human judges were asked to provide their opinion of the reasons each question was posted. They were allowed to give as many reasons that they thought applied from the list: throws an error/exception, has a data race, terminates normally but does not achieve its intended functionality (semantic/logic error), uses incorrect or deprecated methods, is extremely inefficient, is insecure, has a platform/portability problem, wants code explained, wants something added to code, or other.

To account for subjectivity, each of the 240 questions was examined by two judges. Upon disagreement, another judge who had not previously evaluated the post judged the entry, and a majority opinion was used as the ground truth. We also collected information from each judge on their confidence level for each judgement. Confidence level was rated on a 4-point Likert scale.

Threats to Validity. Our technique pulled from 250,000 Stack Overflow questions for the query and background sets, with 60% for the query document set and 40% background document set. The results may not transfer to other Q & A forums; we chose Stack Overflow as it is the most used and we believe it is representative of most software developer forums. A larger study with larger query and background sets might yield different results; however, we found the results did not change much from our smaller development sets to our actual, much larger evaluation sets, so we do not expect very different results with larger query and background sets.

As with any study based on human judges for the ground truth, there might be some cases where the humans may not have correctly answered the questions. To limit this threat, we ensured that the human judges had considerable programming experience and familiarity with Q & A forums, and we ensured that each Stack Overflow question was judged by at least two judges, and when they disagreed, we collected a third opinion and took the majority opinion. We also gathered confidence levels for each judgement, and found quite high confidence overall in all questions.

It is possible that scaling to more than 240 questions in

TABLE II: Results for single-occurrence scheme with different thresholds on ranked bigrams

Bigram Rank Threshold	Unigrams added?	P	R	F-meas
0	yes	74.3	72.8	73.5
25	no	72.4	13.9	23.3
25	yes	73.2	74.2	73.7
50	no	71.8	18.5	29.4
50	yes	71.8	74.8	73.3
75	no	66.1	24.5	35.7
75	yes	71.5	77.5	74.4
100	no	65.6	26.5	37.8
100	yes	70.7	78.1	74.2

TABLE III: Results for scoring-based scheme with different weighting configurations

Weights and Bigram Ranges				Results			
100	75	40	25	Unigrams added?	P	R	F-meas
0-4	5-9	10-24	25+	no	75.0	7.9	14.3
0-4	5-9	10-24	25+	yes	74.8	73.8	74.3
0-9	10-24	25-49	50+	no	68.4	8.6	15.3
0-9	10-24	25-49	50+	yes	73.5	73.5	73.5
General Purpose Sentiment Evaluator				–	48.3	60.8	53.9

our evaluation set might lead to different results. However, we needed to make the human judgement work reasonable to recruit judges. We will expand the evaluation study in the near future with more participants. For similar reasons of tedious human judgements, we have not yet trained the general purpose sentiment analyzer on StackOverflow text, which may improve its effectiveness slightly.

B. Results

For our data set, the humans judged 151 of the code examples as negative based on our definition given in the introduction. They considered the remaining code examples included in the questions as being posted for alternative reasons, thus, not representative of a negative code example.

Overall, the human judges were confident in their responses. When the judges agreed with our system classifying a code example as negative, their average confidence was 2.66 on a Likert 0-3 scale, and similarly for all other situations of agreement/disagreement with the negative code example miner, with little variation.

Table I shows the set of mined negative sentiment indicators ranked by our scoring mechanism with a threshold of 50, along with the 20 manually selected unigrams from the set of mined unigrams ranked above the threshold of 50. Note that "follow error" was most likely "following error" before stemming. The numbers in the parentheses report the frequency that the n-gram occurred in our data set of 240 Stack Overflow questions and the frequency that the n-gram occurred in a question marked as containing a negative code example by our human judges, respectively. Note that the first frequency also represents how many questions that our negative sentiment miner reported as a negative code example using that n-gram. Surprisingly, bigrams such as "work fine" appeared 11 times; it appeared always as a negation. Similarly, "trying" appeared 49 times. These results show the strength of an empirically-driven approach.

RQ1: How do different approaches to mining sentiment indicators and automatic mining of negative code examples impact the effectiveness of the automatic negative example miner? Tables II and III together answer the three research subquestions, by reporting the precision, recall, and F-measure for various configurations. Table II reports results on the configurations of the single-occurrence scheme, while Table III reports results for the configurations of the scoring-based scheme.

In the single-occurrence scheme where a negative sentiment indicator needs to only occur once in a question to classify the contained code example as negative, how does the threshold T for the ranking of the highest T scoring bigrams to be considered affect the precision and recall of the negative code example miner? We report precision, recall, and F-measure for thresholds to include the top 25, 50, 75, and 100 ranked bigrams from our customized negative sentiment indicators. Overall, the precision is fairly similar at the different thresholds, especially when unigrams are also used. If only high precision is desired to ensure that the negative code examples that are mined are indeed negative code examples, then lower thresholds should be used. If more recall is desired without sacrificing much precision, then a threshold of 50 top bigrams with unigrams is a good choice.

How effective is a negative code example miner under different configurations of the scoring-based approach with weighting the bigram occurrences in questions based on their ranking? With our training set, we found the two ranges in Table III to be the best. While the precision did not vary much among the weighting schemes when unigrams were included, there was a noticeable difference in precision among the weighting schemes when unigrams are not included. Our best weighting scheme, which has the smallest ranges and added unigrams (which we call the 0-5-10-25 scoring-based scheme), had little added precision or recall over the best single-occurrence scheme. While we consider this to be the best scheme given that it achieves the highest precision among all schemes without much loss in recall, we cannot definitively say that weighting is beneficial.

How does including manually selected unigrams as indicators affect the results in both the single-occurrence and the scoring-based schemes? Including the unigrams in the negative sentiment indicator set achieves much better recall for all configurations of both the single-occurrence scheme and the scoring-based scheme than without the unigrams. As seen in Table II, the 20 unigrams alone (depicted by the first row of results) achieved a much better F-measure than any pure bigram scheme. This is the insight that led us to our scoring-based scheme, where we weighted bigrams based on ranking and left unigrams unweighted. As indicated by Table I, the bigrams did not appear frequently enough in our data set to make much impact, and the unigrams dominated the results, especially the recall.

RQ2: How effective is our automatic mining technique in identifying negative code examples? We compared our best scheme, the 0-5-10-25 scoring-based scheme, against our negative code example miner that uses the general purpose sentiment analyzer from Python NLTK [10]. To create this miner, we input the text of each of the 240 Stack Overflow questions to the NLTK text classifier, which outputs a cate-

gorization of positive, negative, or neutral. In this work, we combined positive and neutral into non-negative.

The general purpose sentiment analyzer-based approach yields 48.3% precision, 60.8% recall, and 53.9 F-measure on our data set. Our best configuration, the 0-5-10-25 scoring-based scheme, achieves 74.8% precision, 73.8% recall, and an F-measure of 74.3. We believe this is very promising, as it increases precision by 26.5%, a 54.9% increase in precision over the general purpose sentiment analysis-based system.

C. Qualitative Analysis

Our qualitative analysis focuses on answering the two subquestions of RQ2: *When our system is not effective, what is the breakdown between mined negative examples that humans judge to be non-negative and missed negative code examples? What are the characteristics of incorrectly classified code examples and missed negative code examples?* We examined the Stack Overflow questions where our best configuration, the 0-5-10-25 scoring-based scheme, either missed negative code examples or incorrectly identified an example as negative code. There exist 39 questions that were identified incorrectly as containing negative code examples and 41 questions that the human judges said contained negative code examples, but the system missed them. Table IV shows examples from each of these categories after showing some examples of negative code examples where the human judges and the system agreed.

Analysis of the code segments incorrectly identified as negative code examples showed that most of them were cases where the author wanted to add something to their code, and was using code examples as context for the question. In the third example in Table IV, the poster is soliciting help regarding “changing the ’jagged’ effect of the flames”. The poster is modifying non-negative code taken from sample code online. The poster’s dissatisfaction is not with the sample code, but with attempting to amend the code to suit his/her intentions. Further, as shown in the fourth post, occasionally the mined sentiment indicators are incorrectly applied. Many times, if code was identified incorrectly as negative, the sentences containing the negative sentiment did not pertain to the code. Rather, the code served a contextual role.

Our analysis of the questions where the negative code miner missed negative code examples revealed some limitations of using a system based on n-grams. The fifth example in Table IV contains a segment of inefficient code. The poster mentions the word “lag” multiple times. Lag is a popular indicator of negative code; however, it was not contained in the mined bigram/unigram list, resulting in the negative code example going undetected. Lastly, the majority of negative code examples that were missed were logic errors involving code that did not work as the author specified. These questions did not contain negative sentiment, rather, they contained syntactical shifts. Posters used words such as “but” and “although” to indicate that there was a source of dissatisfaction with the code. As seen in the sixth example, using a desktop browser, the code returned the url perfectly. However, the poster continues, “but when I am testing same website code on my mobile browser...”, indicating the result is not correct as it was previously.

V. Related Work

Prior work has shown that code examples are an important learning tool [1], [2]. Code examples have also been mined for other purposes. Kim et al. mined ”high-quality code examples” from the internet to implement a new approach for a code search engine [4]. Holmes et al. mined code examples that were indicative of a certain API’s implementation [5]. They created the Strathcona Recommendation Tool, which could be used by developers to find mined, API-relevant fragments of code. Zhong et al. developed a tool called MAPO, which automatically mines API usage patterns [6]. Keivanloo et al. strived to make it possible to automatically mine working code examples from open source Java projects [14]. They found that most code search engines tend to fail to take into account whether code works or not, thereby not providing developers with intended results. Ponzanelli et al. developed an Eclipse plug-in, Prompter, which scans through a developer’s code and retrieves relevant Stack Overflow question results [15].

Sentiment analysis has been applied to discussion forums such as travel forums and other non-software developer contexts where opinions are common [9], [16], [17]. In software engineering, sentiment analysis has been used to examine developer emotions in various developer communications. Novella et al. analyzed the relationship between the phrasing of a question and the overall response to that question [18]. In particular, they explored how the ”emotional style” of a question affects the will of others on the website to contribute an answer.

Tourney et al. used sentiment analysis to identify the feelings of distress and happiness that might exist within a software development team [19]. Mailing lists from the Apache Software Foundation were used to identify the positive and negative sentiment. They filtered out all source code and only took into account natural language, then applied the SentiStrength tool to determine the sentiment expressed in the emails. Murgia et al. explored the same issue of developer satisfaction, but applied it to Apache Software Foundation issue reports [19]. By using human annotators, they found that issue reports express emotion on factors such as design decisions. Bazelli et al. examined the personality traits of Stack Overflow users based on linguistic inquiry and word count [20]. They disregarded all code tags in their analysis. The sentiment indicators used by these researchers cannot be applied to our research problem, as we are looking for sentiment about code segments embedded in questions, which requires identifying sentiment indicators related specifically to the nearby code segments, not overall opinion.

VI. Conclusion

In this paper, we present the first known technique for automatically mining negative code examples. Such negative code examples can be used for learning, quality assessment, and interactively helping programmers within the IDE to improve their code. The approach demonstrates yet another useful kind of information that can be mined from Q & A forums, by performing sentiment analysis on questions that contain code segments. Our evaluation shows that we can achieve precision of 75% with recall of 74% with a single configuration of weighted sentiment indicators, which is promising and much better than using general purpose sentiment analysis.

TABLE IV: Examples of StackOverflow Entries Mined and Analyzed

Identified correctly as negative code examples
I am **right now** debugging my socket application, witch involves running and shutting down it consistently. My problem is when I run and shut it down then run it again I receive 10048 **error** code, witch indicate address already in use. I tried to set socket descriptor to SO_REUSEADDR but still receiving 10048 **error** code if I run and close my application consistently. /***CODE***/ *http://stackoverflow.com/questions/20009407/how-to-reuseaddress-option-with-socket-properly*
I'm **trying** to clear a part of the canvas using clearRect but it doesn't seem to **work**. /***CODE***/ In the above code i'm trying to clear the **work** "GRAPE" but it does not **work**. Where am I **going wrong** ? *http://stackoverflow.com/questions/20051279/issues-with-clearrect-in-canvas*
Identified incorrectly as negative code examples
I am using the "fire" effect from this neat little code snippet from: http://www.script-tutorials.com/html5-fire-effect/ but as I want to use it without the background image I am **trying** to **work** out how not to have the hard edge of the canvas around the flames. I have been going through the function but since it is the **first time** I work with canvas I am not understanding how to control the animation. If you go on the tutorial link above you will see the full code from the extract I posted below. I believe this to be the part where the flames are actually drawn. /***CODE***/ I do not know if anyone can help or at least point me in the right directions. Is there a way of a) change the "jagged" effect of the flames (like the yellow ones) so as they do not cut off on the edges of the canves or b) Make the entire animation softer around the canvas edges or fade? I have been searching for standard HTML5 canvas flame tutorials online but they are either really complex or really cheesy. If anyone knows of a good alternative, let me know. *http://stackoverflow.com/questions/20600444/html5-canvas-fire-fade-around-edges*
I need to use some dynamic information from "system.xml" on my 'config.xml'.How i do that??**Something** like this:(system.xml) /***CODE***/ And on my config i'll use that information setted up by customer(user) to do **something** else.Its possible? *http://stackoverflow.com/questions/21563645/how-to-use-dynamic-information-on-config-xml*
Identified incorrectly as non-negative code examples
Recently I've noticed in my game whenever iAd changes the currently displayed ad, there is about a 0.25 second lag in the game, which is just enough to be noticeable. After the ad finishes loading, there are no lag problems, but up until that point, if the user is in-game, that could hurt their experience.Has anyone found a solution to the iAd lag problem? /***CODE***/ I've also noticed that there is a lag right when I ad the banner view to the screen. *http://stackoverflow.com/questions/22373212/iad-lags-when-changing-ads*
I am using html form to integrate Paypal and my form is as /***CODE***/ When I am testing this Website code on desktop browsers this is posting paypal data back on return url perfectly. But when I am testing same website code on my Mobile browser payapl is not posting back any data on return url. It just get back on return url without any data.Please help me. *http://stackoverflow.com/questions/20406455/paypal-not-posting-data-on-return-url-on-mobile-browsers*

Our future work focuses on improving the precision of the automatic miner of negative code examples. We are investigating techniques to take into account the context and location of the negative sentiment indicators within the forum questions. We are also examining the potential role of machine learning.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF 1422184. We thank Oana Tudor for her guidance on the NLP tools and techniques.

REFERENCES

[1] M. Robillard, "What makes apis hard to learn? answers from developers," *Software, IEEE*, vol. 26, no. 6, pp. 27–34, 2009.

[2] F. Shull, F. Lanubile, and V. Basili, "Investigating reading techniques for object-oriented framework learning," *Software Engineering, IEEE Transactions on*, vol. 26, no. 11, pp. 1101–1118, 2000.

[3] S. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming q & a in stackoverflow," in *Software Maintenance (ICSM), 28th IEEE International Conference on*, Sept 2012, pp. 25–34.

[4] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, "Towards an intelligent code search engine." in *AAAI*, 2010.

[5] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, 2005, pp. 237–240.

[6] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," vol. 5653, pp. 318–343, 2009.

[7] J. L. Plass, R. Moreno, and R. Brünken, *Cognitive load theory*. Cambridge University Press, 2010.

[8] J. Bergin, in *Pedagogical Patterns: Advice For Educators*. Joseph Bergin Software Tools, August 2012.

[9] B. Liu, *Sentiment Analysis and Opinion Mining*. Morgan and Claypool Publishers, 2012.

[10] N. Project. (2015, May) Natural language processing apis and python nltk demos. [Online]. Available: text-processing.com

[11] C. Tudor, C. Schmidt, and K. Vijay-Shanker, "egift: Mining gene information from the literature," *BMC Bioinformatics*, vol. 11, no. 1, p. 418, 2010.

[12] J. Miller, M. Torii, and K. Vijay-Shanker, "Building domain-specific taggers without annotated (domain) data," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*.

[13] C. D. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[14] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014.

[15] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014.

[16] "Content analysis of online discussion forums: A comparative analysis of protocols," *Educational Technology Research and Development*, vol. 52, no. 2, 2004.

[17] A. F. Wicaksono and S.-H. Myaeng, "Automatic extraction of advice-revealing sentences foradvice mining from online forums," in *Proceedings of the Seventh International Conference on Knowledge Capture*, ser. K-CAP 2013.

[18] N. Novielli, F. Calefato, and F. Lanubile, "Towards discovering the role of emotions in stack overflow," in *Proceedings of the 6th International Workshop on Social Software Engineering*, ser. SSE 2014.

[19] A. Murgia, P. Tourani, B. Adams, and M. Ortu, "Do developers feel emotions? an exploratory analysis of emotions in software artifacts," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014.

[20] B. Bazelli, A. Hindle, and E. Stroulia, "On the personality traits of stackoverflow users," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*.