Recommending Relevant Sections from a Webpage about Programming Errors and Exceptions

Mohammad Masudur Rahman
Department of Computer Science
University of Saskatchewan, Canada
masud.rahman@usask.ca

Chanchal K. Roy
Department of Computer Science
University of Saskatchewan, Canada
chanchal.roy@usask.ca

ABSTRACT

Programming errors or exceptions are inherent in software development and maintenance, and given today's Internet era, software developers often look at web for finding working solutions. They make use of a search engine for retrieving relevant pages, and then look for the appropriate solutions by manually going through the pages one by one. However, both the manual checking of a page's content against a given exception (and its context) and then working an appropriate solution out are non-trivial tasks. They are even more complex and time-consuming with the bulk of irrelevant (i.e., off-topic) and noisy (e.g., advertisements) content in the web page. In this paper, we propose an IDE-based and context-aware page content recommendation technique that locates and recommends relevant sections from a given web page by exploiting the technical details, in particular, the context of an encountered exception in the IDE. An evaluation with 250 web pages related to 80 programming exceptions, comparison with the only available closely related technique, and a case study involving comparison with VSM and LSA techniques show that the proposed technique is highly promising in terms of precision, recall and F_1-measure.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Techniques—*relevant content mining, traceability*

Keywords

Content relevance, Content recommendation, Traceability

1. INTRODUCTION

Studies show that about 80% of total effort is spent in software maintenance [20]. During the development and maintenance of a software product, software developers deal with different programming errors and exceptions, and they often search in the web for working solutions for solving them. According to Brandt et al. [5], developers spend about 19% of their development time in web surfing. During the collection of information using traditional web search, they first use a search engine with a few keywords for retrieving relevant pages. However, in order to locate the required information, they need to go through the pages one by one, which is challenging, and this paper focuses on this particular research problem. Both manual checking of a web page for relevant content against an error and working an appropriate solution out are non-trivial tasks. These tasks get even more complex and time-consuming with the bulk of irrelevant (i.e., off-topic) and noisy (e.g., advertisement) content in the page. As early as 2005, Gibson et al. [11] estimated that about 40%-50% of web data were simply noise. Thus, the developers often spend a significant amount of time and efforts in searching and then extracting the content of interest from the web pages. Fortunately, automated support in post-search content analysis can greatly benefit them in this regard. For example, identification and then recommendation of page sections relevant for the developers from a selected web page can help them get rid of information overload and locate the content of interest instantly, which in turn reduces their overall problem-solving efforts.

A number of existing studies focus on extracting the noise-free version of a web page by applying different techniques [6, 7, 8, 9, 10, 12, 13, 14, 15, 18, 22]. However, no studies target the extraction of relevant sections or sections of one's interest from the page. Thus, they fail to direct one to the right (or relevant) sections, and do not help much either in reducing information overload or in locating solution in the page. Furthermore, most of the techniques are domain specific (i.e., applies domain knowledge) or template specific (e.g., tabular structure) [6, 14], and they extract content from various domains such as news [8, 12, 13, 18, 22], Wikipedia [6], and real estates [10]. However, none of them deals with programming related web pages, which makes our work unique and novel in this context.

In this paper, we propose a novel technique that identifies and then recommends the relevant sections from a programming related web page by exploiting the technical details of an encountered exception in the IDE (i.e., *context-aware* technique). Once a developer searches about an exception using a few keywords, the search engine (e.g., in our case Google) returns a number of pages. Then the real challenge for her is to manually check those pages and collect meaningful information for the encountered exception, where our technique comes handy. For example, the code under devel-

.

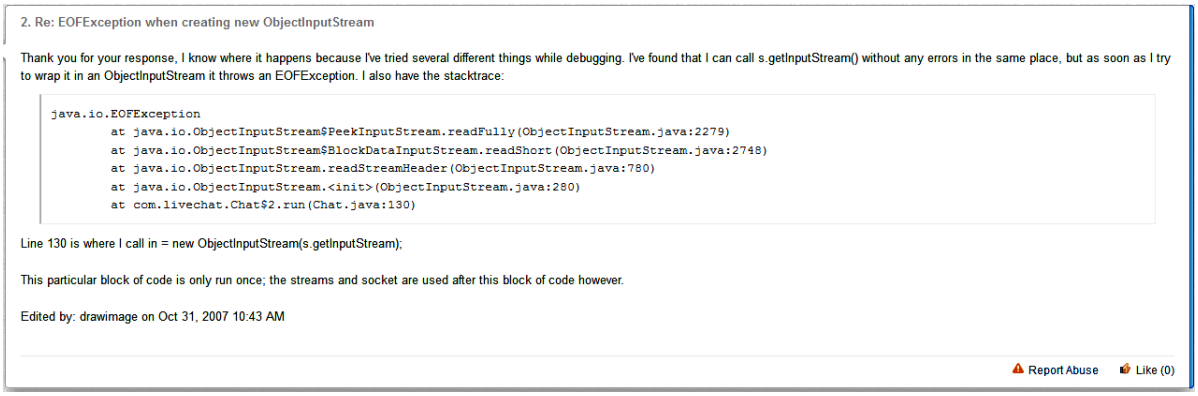

Figure 1: An Example Relevant Section

Listing 1: Context code of an Exception

```
//more code goes here ...
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
ArrayList<Record> currentList = new ArrayList<>();
int size = ois.readInt();
for (int i = 0; i < size; i++) {
  Record current = (Record) ois.readObject();
  currentList.add(current); }
```

Listing 2: Stack Trace of the Exception

```
java.io.EOFException
at java.io.ObjectInputStream$PeekInputStream.readFully(
     ObjectInputStream.java:2325)
at java.io.ObjectInputStream$BlockDataInputStream.
     readShort(ObjectInputStream.java:2794)
at java.io.ObjectInputStream.readStreamHeader(
     ObjectInputStream.java:801)
at java.io.ObjectInputStream.<init>(ObjectInputStream.
     java:299)
at core.MyEOFTest.main(MyEOFTest.java:40)
```

opment (hereby we call it *context code*) in Listing 1 triggers an *EOFException*, and the IDE reports the stack trace in Listing 2. Our technique analyzes both the content of a returned web page (e.g., Fig. 2) and the technical details of the exception (e.g., Listing 1 and Listing 2), analyzes legitimacy (i.e., content purity) and relevance of different sections from the page, and then identifies the most relevant section (Fig. 1, boxed area from Fig. 2) in the page for the developer. We integrate Google search API into Eclipse IDE to collect web pages for the developer provided search queries about an exception, and then use those pages for the recommendation of relevant sections from them one by one as the developer wishes. In this way, even though Google search may return a lot of web pages, our technique can reduce the burden for the developer by recommending the relevant sections or even indicating that some particular pages might not have any relevant sections at all for the encountered exception. We package our recommendation solution into an Eclipse plug-in prototype, called, *ContentSuggest* [1].

Our proposed technique also complements existing studies in certain aspects. First, existing density metrics [13, 22] fall short in extracting content from programming related web pages, and we propose a novel density metric for programming content– *code density* in order to complement them. Second, our technique introduces a novel idea of leveraging *content relevance* in the extraction and then recommendation of web page content. It should be noted that this work is fundamentally different from our previous work–SurfClipse [20] that returns a list of relevant pages for any exception. On the other hand, this work returns the most relevant sections from a given web page for the exception of interest.

We evaluate and validate our technique in three ways. An experiment using 250 programming related web pages, 80 programming exceptions, and their technical details shows that our technique recommends relevant content from a web page with a *precision* of 81.96%, a *recall* of 76.74%, and a F_1-*measure* of 76.30% on average, which are promising. We compared against the only available closely related technique– Sun et al. [22] and found that our technique outperformed

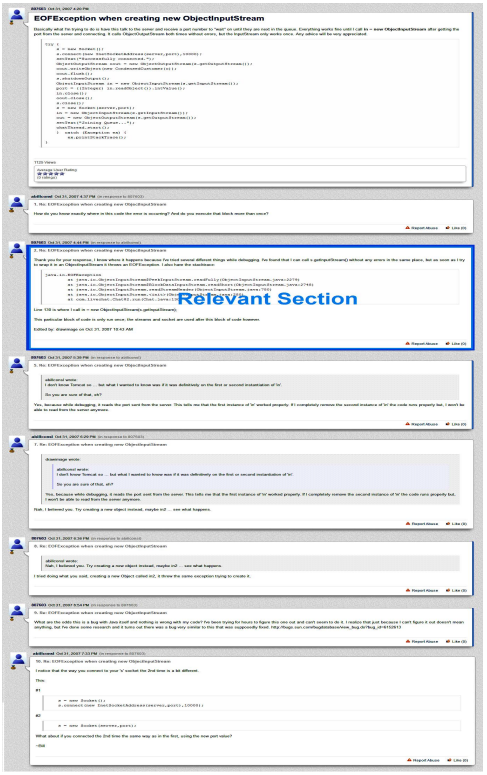

Figure 2: Relevant Section(s) in the Webpage

that technique in terms of all the performance metrics. A case study using 35 StackOverflow web pages and comparing with two state-of-the-art traceability link recovery techniques– VSM [3] and LSA [17] reports that our technique performs significantly well in identifying the page content marked as *relevant* by a large technical crowd. Thus, the paper makes the following technical contributions:

Listing 3: Example HTML Segment (taken from [2])

```
<div id="content">
<div id="question-header">
<h1 itemprop="name">
<a>How to instantiate inner class using reflection?</a>
</h1></div>
<div class="post-text" itemprop="description">
<p>I get this exception:</p>
<pre class="lang-java prettyprint prettyprinted">
<code>java.lang.InstantiationException ..</code>
</pre></div></div>
```

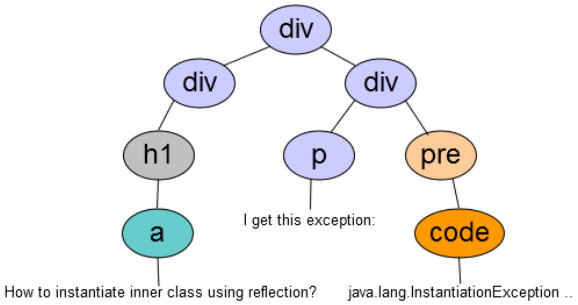

Figure 3: DOM Tree of Example in Listing 3

- We propose a novel metric–*code density* that complements existing density metrics, and extracts content from programming related web pages.
- We introduce *content relevance* in content extraction from a web page, which in turn provides a mean for supporting the developers in post-search content analysis through relevant section recommendation.
- We package the proposed solution into an Eclipse plug-in prototype [1], that captures the technical details of an encountered exception in the IDE, and then recommends the relevant sections from a given web page.

The rest of the paper is organized as follows– Section 2 focuses on the background concepts required for the research and Section 3 discusses the proposed technique including working modules, metrics and algorithms. Section 4 describes the conducted experiments, results and validation followed by a case study, Section 5 identifies the potential threats to validity, Section 6 focuses on the related work, and finally Section 7 concludes the paper.

2. BACKGROUND

Document Object Model (DOM): It is a cross-platform and language independent convention to represent the content of an HTML or XML document. In this model, a document is represented as a tree, where each of the tags is represented as an *inner node* and textual or graphical elements are represented as *leaf nodes*. For example, the HTML code segment in Listing 3 shows the title and body part of a programming question posted on StackOverflow Q & A site, and Fig. 3 shows the corresponding DOM tree. In our research, we use *Jsoup*[1], a popular Java library, for parsing and analyzing the DOM tree of any web page.

Cosine Similarity: It is a measure that is frequently used in information retrieval in order to determine the similarity between two text documents. In our research, we use cosine similarity measure for determining lexical similarity between the context (e.g., stack trace, context code) of a programming exception and the discussion text from a candidate section of a web page. We consider each of the problem context and discussion texts as a *bag of tokens*[2], discard the insignificant tokens (e.g., braces, semicolons, colons, dots and other punctuations), and decompose each token having a camel-case (e.g., StringBuffer) or dotted structure (e.g., java.io.IOException). We then prepare a combined set of tokens, C, from the two sets and calculate *cosine similarity*, S_{cos}, as follows.

$$S_{cos} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (1)$$

Here, A_i represents frequency of i^{th} token from C in set A (i.e., exception context), and B_i represents that frequency in set B (i.e., candidate discussion text). This measure values from zero (i.e., complete lexical dissimilarity) to one (i.e., complete lexical similarity), and helps to determine the lexical relevance between the context of an exception and the candidate section from a web page.

Logistic Regression: It is a probabilistic statistical classification model that predicts binary or dichotomous outcomes based on a set of predictor variables (i.e., features). It is widely used in medical and social science fields. In our research, we use the regression model in association with a machine learning technique for estimating the relative weights (i.e., predictive power) of different *density* and *relevance* metrics for page content extraction (Section 3.5). Logistic regression models the probabilities of different outcomes for a single trial as a function of predictor variables using a *logistic* function. The logistic function is a common sigmoid function, $F(t)$, as follows:

$$F(t) = \frac{e^t}{e^t + 1}, \quad t = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \quad (2)$$

where $F(t)$ is a logistic function of a variable t, which is again a function of the predictor variables x_1 and x_2. Here, β_1, β_2 are coefficients, and β_0 is the intercept in the regression equation. The function always returns a value between zero and one, and thus, provides a probabilistic measure for each type of the outcomes for the trial.

3. PROPOSED APPROACH

3.1 Working Modules

Our proposed technique exploits the technical details of a programming exception encountered in the IDE, and recommends the relevant sections from a given web page. In Fig. 4, the schematic diagram of the technique shows the working modules, and explains different steps required for relevant content identification, recommendation and visualization. We package the whole solution as an Eclipse plug-in prototype [1], and it has three modules as follows:

Content Collector: The *collector module* collects exception message and stack trace from the active *Console View* (Fig. 4-(c)) and *context code*[3] from the active text editor (Fig. 4-(d)) once an exception occurs. It also collects the HTML source of the selected web page (e.g., top one selected in Fig. 4-(a)). Once the developer selects a web page and requests for relevant page sections, the *collector module* downloads HTML source of the page, and sends the source

[1] http://jsoup.org

[2] A collection of tokens with no fixed order

[3] A segment of the source code that generates the exception

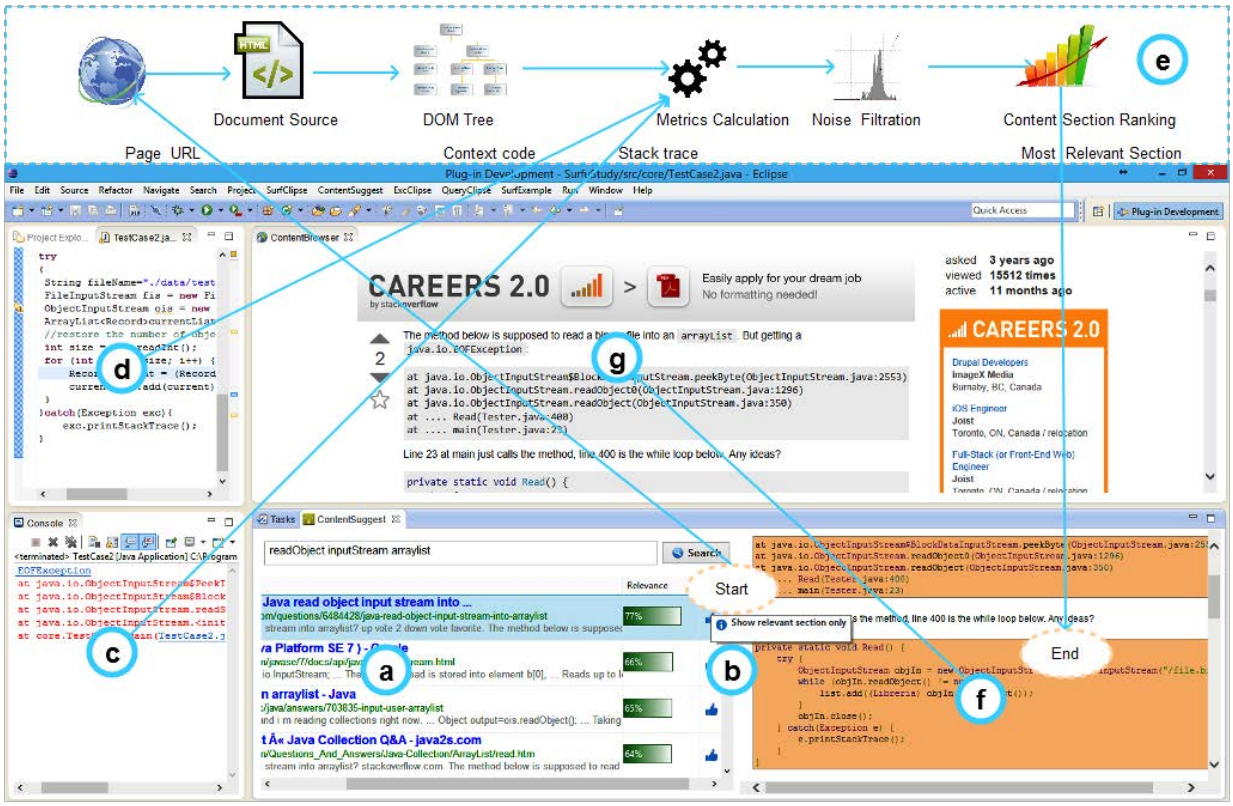

Figure 4: Schematic Diagram of the Proposed Approach

and the previously collected exception details from the IDE to the *extractor module*.

Content Extractor: The *extractor module* (i.e., dashed rectangle, Fig. 4-(e)) analyzes the source of the selected HTML page, parses each of the tags, and develops a DOM tree. It then analyzes each of the tree nodes, calculates their *content density* and *content relevance* (Section 3.2), and assigns content scores. The module then discards the noisy nodes based on their content scores, and identifies the DOM tree nodes most relevant to the encountered exception (Fig. 4-(c)) in the IDE for recommendation.

Content Visualizer: The *visualizer module* consists of two content visualization panels. First, the relevant content panel (Fig. 4-(f)) displays the most relevant section from a web page recommended by the *extractor module*, where it highlights different program elements of interest such as stack trace and code segment. The idea is to help a developer instantly decide if the page is worth browsing or not. Thus, the developer can save time and effort in choosing the appropriate solution for the exception at hand. Once she is convinced by the most relevant section, she then can check the whole page using the embedded browser (Fig. 4-(g)) for in-depth analysis. Second, the result panel (Fig. 4-(a, b)) visualizes the estimated relevance of each result page against the target exception by analyzing the meta description of the page from the search engine. This visualization helps the developer choose the prospective solution pages in the first place during search.

3.2 Proposed Metrics

In this section, we discuss our proposed density and relevance metrics that are used for extracting and recommending relevant section(s) from a given web page.

3.2.1 Content Density (CTD)

Existing studies [13, 22] propose two density metrics–*text density*, and *link density* for *noise-free* content extraction from a web page. However, these metrics are based on regular texts (e.g., news article), and they are neither properly applicable nor sufficient enough for content extraction from programming related web pages. These pages contain items other than regular texts such as stack traces, code segments, and configuration information. We thus modify existing metrics, introduce a new density metric, and then finally propose a composite density metric.

Text Density (TD): *Text Density* represents the amount of any textual content each of the HTML tags in the web page contains on average. The metric roughly estimates the content aspect of the page. Thus, in the DOM tree, *text density* (TD_i) of a node is calculated by capturing its number of child nodes (T_i) (i.e., inner nodes) and the amount of texts (C_i) it contains in the leaf nodes as follows:

$$TD_i = \frac{C_i}{T_i} \tag{3}$$

Link Density (LD): *Link Density* represents the amount of linked (i.e., noisy) texts each of the HTML tags contains on average. The metric roughly estimates the noise aspect of the page. Existing literature [13, 22] considers any linked text in the web page as *noise*. However, in our research, we make a careful choice about them. We analyze the relevance of each linked text element against the exception of interest, and consider the element as *noise* only if its relevance is below a carefully chosen heuristic threshold (η=0.75). We otherwise consider it as a legitimate textual element. Thus in the DOM tree, the *link density* (LD_i) of a node i is cal-

culated by capturing its number of child nodes (T_i) (i.e., inner nodes) and the amount of linked or noisy texts (LC_i) it contains in the leaf nodes as follows:

$$LD_i = \frac{LC_i}{T_i} \tag{4}$$

We consider each <*a*> tag, and check its relevance before considering it as *noise*. As Sun et al. [22] suggest, we also consider <*input*> and <*button*> as linked elements, and their content as linked texts.

Code Density (CD): *Code Density* represents the amount of *code related texts* each of the HTML tags contains on average. Programming related web pages generally contain different program elements such as *stack traces* and *code segments*, and they are of great interest to the developers. The developers often analyze or reuse (i.e., code segments) them for solving their programming problems. We believe that the code related elements complement the discussion texts about programming, and thus *code density* can be considered as an important indicator of legitimacy of a programming related web page. In the DOM tree, the code density (CD_i) of a node i is calculated by considering its number of child nodes (T_i) (i.e., inner nodes) and the amount of code related texts (CC_i) it contains in the leaf nodes as follows:

$$CD_i = \frac{CC_i}{T_i} \tag{5}$$

Previous studies [19, 20] suggest that code related elements are generally posted in the page using `<code>, <pre>` and `<blockquote>` HTML tags. We thus consider the texts from those tags as code related texts in density calculation.

While *text density* metric represents a generalized form of density for all kinds of text, both *code density* and *link density* point to special types of text. *Code density* can be considered as a heuristic measure of programming elements in the text, whereas *link density* is a similar measure for *noise* in the content. In our research, we consider all three metrics of an HTML tag i, and propose a *log-based composite density metric* called *content density* (CTD_i). Our metric is adapted from the *Composite Text Density* metric of Sun et al. [22], and we choose the *log-based* metric in order to better distinguish a legitimate section from a noisy section of the page. Detailed rationale of log-based density can be found elsewhere [22].

$$CTD_i = (TD_i + \frac{CD_i}{TD_i}) \times log_{ln(\frac{TD_i \times LD_i}{\neg LD_i} + \frac{LD_b \times TD_i}{TD_b} + e)}(\frac{TD_i}{LD_i} + \frac{CD_i}{TD_i}) \tag{6}$$

Here, TD_i, CD_i, LD_i and $\neg LD_i$ represent *text density*, *code density*, *link density* and *non-link density* of the HTML tag i respectively. TD_b and LD_b represent the *text density* and *link density* of *body* tag respectively. In Equation (6), $\frac{TD_i}{LD_i}$ is a measure of the proportion of linked texts. When a tag has higher *link density*, $\frac{LD_i}{\neg LD_i} \times TD_i$ increases the log base, $\frac{TD_i}{LD_i}$ gets a lower value, and thus overall *content density* is penalized. However, $\frac{LD_b \times TD_i}{TD_b}$ maintains the balance between these two interacting parts, and prevents a lengthy and homogeneous text block from getting an extremely higher value or a single line text (e.g., page title) from getting an extremely lower value. Moreover, we introduce the programming text proportion of a tag, $\frac{CD_i}{TD_i}$, which improves the overall *content density* metric for the HTML tag that contains both programming texts and regular texts.

3.2.2 Content Relevance (CTR)

Existing studies [13, 22] apply different density metrics in order to discard noisy sections (e.g., advertisements) and extract legitimate sections from a web page. However, these metrics are not sufficient enough for relevant content extraction from the web page, i.e., our research problem. We thus leverage the technical details of an encountered exception in the IDE, and propose three relevance metrics for determining relevance of different sections from a web page.

Text Relevance (TR): *Text relevance* estimates relevance of the textual content from any HTML tag against a given exception and its context. The context of an exception is represented as a list of keywords collected from corresponding stack trace and context code (Section 3.6). For example, Listing 4 shows the context of our showcase exception–`EOFException` in Listing 1 and Listing 2. We calculate *cosine similarity* between such keyword list and the texts from each tag from the page. Cosine similarity measure represents the token overlap between two items. Since the context of an encountered exception contains important tokens such as class names and method names associated with the exception, lexical similarity between an HTML tag and the context suggests the tag's relevance for the exception. The similarity measure values from zero to one, where one refers to complete lexical relevance and vice versa.

Code Relevance (CR): *Code relevance* estimates relevance of a code segment or a stack trace block from an HTML tag against corresponding context code or stack trace of a given exception. In order to estimate *code relevance* of a node from the DOM tree, we analyze three types of child tags– `<code>`,`<pre>` and `<blockquote>` under that node. According to traditional heuristics [20], such tags generally contain the program elements (e.g., code segments). We apply two different techniques for stack traces and code segments for estimating their relevance with their counterparts.

Stack trace of a programming exception contains an error message followed by a list of method call references that point to the possible error locations in the code. We develop separate token list by collecting suitable tokens (e.g., class name, method name) from each of the stack trace blocks of an HTML tag and the stack trace in the IDE respectively. We then calculate *cosine similarity* between the two token lists, and consider the measure as an estimate of relevance for the tag to the exception in the IDE.

In order to estimate relevance of a code segment from the HTML tag against an exception of interest, we collect the *context code* of the exception, and apply a state-of-the-art code clone detection technique by Roy and Cordy [21]. The technique finds out the *longest common subsequence* of source tokens (S_{lcs}) between two code segments. We then use it for determining similarity of the code segment from the HTML tag with the context code as follows, where S_{total} refers to the sequence of all tokens collected from the context code of the target exception.

$$S_{ccx} = \frac{|S_{lcs}|}{|S_{total}|} \tag{7}$$

Once the relevance of all the program elements– stack traces and code segments under an HTML tag are estimated, we find the maximum estimate, and consider it as the *code relevance* for the tag. The metric helps in separating a highly relevant HTML tag containing relevant code elements from a less relevant tag in the web page.

```
Exception in thread "main" java.io.EOFException readInt
ObjectInputStream readStreamHeader BlockDataInputStream
readObject readShort add main readFully FileInputStream
     Record ArrayList PeekInputStream init
```

Listing 4: Context of the Exception in Listing 1 and Listing 2

While *text relevance* focuses on the relevance of any textual element within an HTML tag, *code relevance* estimates the relevance of program elements within it. We combine both relevance metrics in order to determine the *composite relevance metric* called *content relevance* (CTR) as follows:

$$CTR_i = \alpha \times TR_i + \beta \times CR_i \quad (8)$$

Here α and β are the relative weights (i.e., predictive power) of the corresponding relevance metrics. We consider a heuristic value of 1.00 for α and 0.59 for β, and they are estimated using a machine learning based technique (Section 3.5).

3.3 Content Score (CTS)

We consider two different aspects–*density* and *relevance* for each of the content sections in the page for extracting the relevant ones. While the density metrics focus on the legitimacy (i.e., purity) of the content in the page, relevance metrics check the relevance of the same content section against the programming problem (i.e., encountered exception) at hand. The idea is to recommend such section of a page to the developers that is both legitimate (i.e., noise-free) and relevant (i.e., discusses similar problem). We thus combine both aspects, normalize corresponding metrics from Section 3.2, and propose a composite score metric called *content score* (CTS_i) for each of the tags from the page as follows:

$$CTS_i = \gamma \times CTD_i + \delta \times CTR_i \quad (9)$$

Here γ and δ are relative weights (i.e., predictive power) of the corresponding density and relevance metrics–*content density (CTD)* and *content relevance (CTR)*. In our experiments, we note that our technique performs the best when the equal weight ($\gamma = \delta = 1.00$) is assigned to both metrics. The weight estimation process can be found in Section 3.5.

3.4 Extraction of Relevant Page Section(s)

An HTML page is generally divided into a set of identifiable sections (i.e., tags) that can be represented as the child nodes under `body` node in the corresponding DOM tree. Our contribution lies in identifying the most relevant section(s) from that page. Once content score (Section 3.3) for each of the tags (i.e., nodes) in the page (i.e., DOM tree) is calculated, we filter the tree nodes using a heuristic threshold. We consider the content score of `body` node as the threshold score, as suggested by Sun et al. [22] for density-based extraction. We preserve the child nodes under `body` node in the tree having scores greater than the threshold while discarding the others. We then explore each of the preserved child nodes, and find out the inner node with the highest content score. The highest score of the node indicates that the corresponding tag in the HTML page contains the most salient content in terms of legitimacy and relevance for the programming problem at hand. In order to discard noisy or less important elements, we keep that highest scored node along with its child nodes, and mark them as *content* whereas the remaining siblings are marked as *noise*. We apply the same process recursively for each node in the DOM tree, and finally we get each node in the tree annotated as either *content* or *noise*. Then we discard the noisy nodes, and extract the HTML tags corresponding to the remaining nodes in the tree as the *noise-free* sections of the page [22].

Since the first step extracts several sections that might not be equally relevant, we need further filtration for collecting the highly relevant section(s) from the page. We thus focus on *content relevance* (Section 3.2.2) of the preserved nodes, and choose the node with the highest content relevance for recommendation. This highest relevance score indicates that the corresponding HTML tag is relevant both in terms of programming content and discussion texts. We thus ensure that the recommended sections from the page are not only relevant to the problem at hand (i.e., exception) but also are legitimate enough in content to survive the noise filtration.

For example, our proposed technique returns these metric values– TD=32.74, LD=2.88, CD=24.29, CTD=0.02, CR=0.84, TR=0.83, CTR=0.99, and CTS=1.0144, for the page section in Fig. 1. This section outperforms other sections in Fig. 2 both in terms of legitimacy (i.e., purity) and relevance with the target exception in Listing 2. Thus, our technique marks the section as the highly relevant one, and extracts it for recommendation.

3.5 Metric Weight Estimation

In order to determine relative weights of two relevance metrics–*text relevance* and *code relevance* and two composite metrics–*content density* and *content relevance*, we choose 50 random web pages from the dataset. Details on dataset preparation can be found in Section 4.1. We then collect the corresponding metrics for 33,360 text blocks (i.e., tags) from those pages using our technique. We identify whether each of those blocks is included in the *gold content* or not (Section 4.1), which provides a *binary class label* (i.e., "0" or "1") for the text block against its set of metrics (i.e., features). We then apply machine learning on the collected block samples using logistic regression that provides a regression model [16] The model is developed on Weka[4], and it is validated using 10-fold cross-validation. The regression model contains a coefficient for each of the features which are tuned by Weka for classifying each sample with maximum accuracy. We believe that these coefficients are an estimate of the predictive power for the features used in the model, and we consider them as the weights of the individual relevance metrics [16]. For the sake of simplicity and for reducing bias, we normalize those coefficients, and consider a heuristic weight α=1.00 for *text relevance* and β=0.59 for *code relevance* metrics. While β has an initial value of 0.86 from the regression model, we got the global maximum at $\beta = 0.59$ for our dataset through iterative experiments [20].

In case of composite density and composite relevance metrics, we find that the proposed technique performs significantly well with equal relative weights assigned. Thus, we consider a heuristic weight of 1.00 for both of the composite metrics, i.e., $\gamma = \delta = 1.00$.

3.6 Exception Context Representation

In our research, we not only take the density metrics but also the relevance estimate of each of the sections from the web page into consideration. Each page in the dataset (Section 4.1) is relevant to a particular exception, and we exploit the details such as stack trace and context code of that ex-

[4]http://www.cs.waikato.ac.nz/ml/weka

ception in the dataset for relevance estimation of different sections from the page. We analyze the stack trace (e.g., Listing 2) and extract different tokens such as *package name*, *class name* and *method name* from each of the method call references. We also analyze the context code (e.g., Listing 1) of the exception and collect *class* and *method name* tokens. We use *Javaparser*[5] for compilable code and an *island parser* for non-compilable code for extracting the tokens [20]. Then we combine tokens from both the stack trace and the context code, and append the exception name along with the exception message (i.e., highlighted line of the stack trace in Listing 2) to the combined set. We call this token set as the *context representation* for the exception of interest. For example, Listing 4 shows the context representation by our technique for an *EOFException* with stack trace in Listing 2 and context code in Listing 1. We use such context representation to estimate the relevance of any page sections to the exception (Section 3.2.2).

4. EVALUATION & VALIDATION

4.1 Experimental Dataset

Data Collection: We use a dataset of 250 web pages and 80 programming exceptions associated with standard Java platform and Eclipse plug-in framework for experiments. We include the technical details of each exception such as stack trace and context code in the dataset. For details on how exceptions were collected, please consult our previous work [20]. It should be noted that each of the pages is carefully selected and taken from the dataset of the previous study [20]. We also collect the HTML source of each of the pages, and include in the dataset for the experiments.

Gold Set Development: We manually analyze each of the 250 pages, and extract *gold content* from them for the study. We consider the most relevant page section for a given exception as the gold content from the page. We also adopt a simplified definition for relevant sections in the page. Programming sites focusing on errors and exceptions often include code snippets and stack traces as a part of discussion. We look for such page sections that contain relevant stack traces or relevant code segments, and extract them as the gold content through an extensive manual analysis of 20 to 25 working hours. One can wonder about the simplified definition of relevance given that the concept of relevance is mostly subjective. However, our goal in this work is to present the presumably relevant sections to signal the relevance of a selected page, and help the developers find the solution with less information analyzed. Thus, the adoption of simplified relevance for page sections is justified.

Cross-Validation: Since the concept of relevance is subjective, in order to reduce the bias, we perform cross validation on the gold set with the help of peers. Two graduate research students randomly checked a subset of ten pages each, and submitted the most relevant sections from the pages against the exceptions of interest. We found that most of their choices match with our gold set selection which provides us confidence on the data. We made this gold set available online [1] for others to use.

Data from Stack Overflow: About 40% of the pages came from StackOverflow (SO) site, and we thus divide the pages into two subsets called *SO-Pages* and *Non-SO Pages*. Again all data are hosted online [1]. While both sets are used for evaluation and validation, we additionally use *SO-Pages* for a case study (Section 4.5) where our technique locates the *highest voted* and the *accepted* answer posts from a given StackOverflow page.

4.2 Performance Metrics

Our proposed technique is greatly aligned with the research areas of information retrieval and recommendation systems, and we use a list of performance metrics from those areas for evaluating our technique as follows [20, 22]:

Mean Precision (MP): *Precision* determines the percentage of the retrieved content that is expected (i.e., in the gold content) from a web page. In our evaluation, we compare the retrieved content by our technique with the manually extracted gold content. As Sun et al. [22] suggest, we use longest common subsequence (LCS) of words between retrieved content and gold content for precision calculation. Thus, *precision* can be determined as follows, where a refers to the word sequence of retrieved content and b refers to that of the corresponding gold content.

$$P = \frac{|LCS(a,b)|}{|a|}, \quad MP = \frac{\sum_{i=1}^{N} P_i}{N} \tag{10}$$

Mean Precision (MP) averages the precision measures for all web pages (N) in the dataset.

Mean Recall (MR): *Recall* determines the percentage of the expected content (i.e., gold content) that is retrieved from a web page by a technique. We calculate the *recall* of a technique as follows:

$$R = \frac{|LCS(a,b)|}{|b|}, \quad MR = \frac{\sum_{i=1}^{N} R_i}{N} \tag{11}$$

Mean Recall (MR) averages the *recall* measures for all pages (N) in the dataset.

Mean F_1-measure (MF): While each of *precision* and *recall* focuses on a particular aspect of the performance of a technique, F_1-measure is a combined and more meaningful metric for evaluation[6]. We calculate F_1-measure from the harmonic mean of *precision* and *recall* [22] as follows:

$$F_1 = \frac{2 \times P \times R}{P + R}, \quad MF = \frac{\sum_{i=1}^{N} F_{1i}}{N} \tag{12}$$

Mean F_1 (MF) averages all such measures.

4.3 Experimental Results

We conduct experiments on the proposed technique using our dataset, and evaluate the technique using three performance metrics–*precision*, *recall* and *F_1-measure*. Our technique extracts relevant content from the web pages with a *mean precision* of 81.96%, a *mean recall* of 76.74%, and a *mean F_1-measure* of 76.30%. Table 1 investigates the effectiveness of the two aspects–*density* and *relevance* associated with the page content in extracting relevant sections. We consider each of these aspects in isolation as well as in combination, and evaluate our technique with different sets of web pages. In case of *content density*, the proposed technique performs well in terms of *recall* and performs significantly poor in terms of *precision* with all three sets–*StackOverflow pages, Non-StackOverflow pages* and *All pages*. For example, the technique can return only 50.07% relevant content (i.e., *precision*) while it uses *density* metrics alone. In the case of *content relevance* metric, our technique extracts relevant content from a web page with relatively better *preci-*

[5] http://code.google.com/p/javaparser/

[6] http://stats.stackexchange.com/questions/49226/

Table 1: Experimental Results for Different Metrics

Score Combination	Metric	SO Pages	Non-SO Pages	All Pages
{Content Density (CTD)}	MP	50.91%	49.50%	50.07%
	MR	91.74%	75.71%	**82.18%**
	MF	62.32%	53.76%	57.22%
{Content Relevance (CTR)}	MP	86.63%	69.17%	**76.23%**
	MR	52.17%	57.66%	55.44%
	MF	61.07%	55.88%	57.98%
{Content Density (CTD) & Content Relevance (CTR)}	MP	92.64%	74.60%	**81.96%**
	MR	74.17%	78.51%	**76.74%**
	MF	80.95%	73.09%	**76.30%**

MP=Mean Precision, **MR**=Mean Recall, **MF**=Mean F_1-measure

sion (e.g., 76.23%), but the *recall* rates are still poor (e.g., 55.44%). On the other hand, when we combine both the density and relevance metrics, we experience significant improvements in all three performance metrics with each of the sets of web pages. For example, our technique successfully extracts 76.74% of the gold content with 81.96% precision when both metrics are considered in combination. This clearly shows the benefit of our introduced paradigm– *content relevance* in the extraction of relevant content from a web page, which is one of our primary objectives of this work. The finding also justifies our use of various density and relevance metrics since it demonstrates their isolated and combined effectiveness in relevant content extraction.

Among the two subsets, our technique performs comparatively better for *StackOverflow pages* with the metrics both in isolation and in combination. During gold set development, we note that StackOverflow pages are structurally organized in the presentation of questions and answers, and they contain relatively less noise. This might have helped our technique perform better. However, we do further investigation with SO-Pages in Section 4.5.

4.4 Comparison with Existing Approaches

Since there is no existing study that addresses the same research problem as ours, i.e., relevant section(s) extraction from programming related web pages, we choose a closely related existing technique– Sun et al. [22]. It applies a list of density metrics for extracting *noise-free* content from a given web page. We replicate their technique with minor adjustments in our working environment, and collect the most legitimate (i.e., representative) section of the page extracted by the technique. The idea is to investigate how closely the legitimate section by Sun et al. [22] matches with the manually extracted relevant section (i.e., gold content), and also to validate the performance of our technique. We compare with their technique for the same dataset, and find out that our technique performs comparatively better in terms of all performance metrics. Table 2 and Fig. 5 report our findings from the comparative study.

Fig. 5 shows the comparative analysis between the two techniques using box plots. We note that our technique performs significantly better in terms of especially *precision* and *f-measure* than Sun et al. [22]. Our technique provides a *median* measure from 85% to 100% for all three metrics, whereas their technique provides such measure from 40% to 50% with an exception in *recall* that ranges around 80%. Table 2 further breaks down the results into different subsets– *SO-Pages* and *Non-SO Pages*, and we experience the similar findings. While all these findings demonstrate the potential of our technique in locating relevant sections in the page, one could still argue about the relevance of the extracted section, in particular, because of the subjectivity involved. This concern is addressed using a case study in Section 4.5.

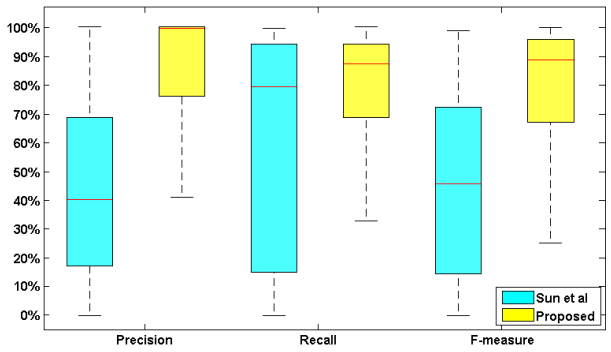

Figure 5: Comparison with Existing Technique

Table 2: Comparison with an Existing Technique

Content Extractor	Metric	SO Pages	Non-SO Pages	All Pages (D)
Sun et al. [22] (Density)	MP	52.63%	38.89%	**44.44%**
	MR	86.49%	41.84%	**59.88%**
	MF	62.57%	34.49%	**45.84%**
Proposed Approach {Density & Relevance}	MP	92.64%	74.60%	**81.96%**
	MR	74.17%	78.51%	**76.74%**
	MF	80.95%	73.09%	**76.30%**

4.5 Case Study with StackOverflow Pages

Although our gold set for the experiments is carefully prepared and validated by the peers, it may still contain subjective bias. Thus, the evaluation and the validation results might be biased. In order to mitigate this threat, we exploit an alternative approach, and conduct a case study using StackOverflow questions and answers. This section discusses the details of the conducted case study.

Dataset Preparation: StackOverflow API[7] provides access to a rich dataset of questions and answers, and we exploit that API service in developing our dataset for the case study. The goal is to investigate whether our proposed technique can actually locate the answer posts within the page that are either accepted as solutions or highly voted by the large user base of StackOverflow. In StackOverflow, each of the posted answers is reviewed by thousands of technical users, and we leverage that crowd knowledge (i.e., the evaluation by hundreds, if not thousands of users) in developing the gold set for this case study. We chose 35 StackOverflow questions related to 29 programming exceptions from our dataset based on this condition– each of the questions should have at least three answers with one answer accepted as *solution*. We develop two gold sets– *most-voted-gold-set* (i.e., contains top-scored answers) and *accepted-gold-set* (i.e., contains accepted answers) for the study. While the question section is discarded from the StackOverflow page for reducing noise, we preserve the DOM structure of the page for enabling our technique to operate conveniently and to leverage the structure for relevant section extraction.

Running of Study: One can essentially think of our research problem as a standard traceability problem since our technique basically links the encountered exception (and its details) to the relevant sections in the web page. However, there exist several important particulars which need to be considered. First, our technique attempts to identify the most relevant section(s) from within a single HTML page rather than an entire page (i.e., document) from within a large corpus. Second, automatically separating relevant sections from an HTML page is a non-trivial task, and our technique exploits the DOM structure of the page for ex-

[7] http://data.stackexchange.com/stackoverflow

Table 3: Comparison with Existing IR Techniques

Content Extractor	Metric	Accepted Posts	Most Voted Posts
Latent Sematic Analysis [17]	MP	19.98%	**23.02%**
	MR	21.78%	**23.17%**
	MF	18.43%	21.07%
Vector Space Model [3]	MP	22.50%	**33.89%**
	MR	23.08%	**31.90%**
	MF	19.77%	30.44%
Proposed Approach {Density & Relevance}	MP	23.10%	**31.36%**
	MR	45.15%	**54.42%**
	MF	26.99%	**35.90%**

MP=Mean Precision, **MR**=Mean Recall, **MF**=Mean F_1-measure

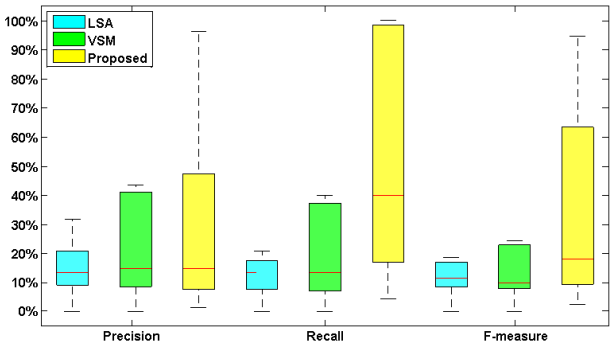

Figure 6: Comparison with existing IR techniques

tracting such sections. We apply our technique on each of 35 StackOverflow pages, extract the most relevant sections, and then compare them with the most voted and accepted answer posts from the gold set. Two state-of-the-art information retrieval techniques– Latent Semantic Analysis (LSA) and Vector Space Model (VSM) are successfully applied in traceability link recovery by several existing studies [3, 4, 17], and we also contrast our technique with them. Since those techniques require a corpus for information retrieval, we represent each of the answer posts from the page as an individual document in the corpus for that page. We then use the *context* (Section 3.6) of the exception related to that page as the search query for retrieving the most relevant document (i.e., answer post). For LSA, we use *TML*[8], a text mining library, and for VSM, we use *Apache Lucene*[9], a popular VSM-based search engine.

Results and Discussions: From Table 3, we note that our technique performs comparably in terms of *precision* and significantly better in terms of *recall* than the other two techniques. For example, our technique returns 54.42% of the gold content with a *precision* of 31.36%, which is promising. One can argue that the results are relatively poor compared to our previously reported results (Section 4.3), which is true. However, we would argue that these results are still promising according to the relevant literature [3, 17], and our technique actually performs better than the two state-of-the-art information retrieval techniques– LSA and VSM. They can retrieve at most 23.17% and 31.90% of the gold content (i.e., *most-voted-gold-set*) respectively. On the other hand, our technique is found more effective in automatically locating the answer post within a given page which is reported as the most helpful (i.e., most up-voted post) by thousands of technical users from StackOverflow. As shown in Table 3, our technique also locates the answer post accepted as solution by the users from a given SO page more effectively than the other two competing techniques.

[8]http://tml-java.sourceforge.net
[9]http://lucene.apache.org/core

Fig. 6 summarizes the comparative analysis among the three techniques using box plots. We note that our technique is comparable to *LSA* and *VSM* in terms of *precision* and significantly better in terms of *recall*. The *median recall* of our technique ranges from 40% to 50% whereas its counterparts in the remaining techniques range from 10% to 20%. Since *F-measure* combines both *precision* and *recall*, we report that our technique actually performs better as a whole due to its improved *recall* rates. It also should be noted that our technique automatically locates the gold answer sections by exploiting the DOM structure of the whole page. On the other hand, those techniques operate on already extracted answer sections from the Stack Overflow page, which provides their comparable precision. Thus, despite of smaller sample size and relatively lower performance than that of first experiment (Section 4.3), the finding clearly demonstrates the potential of our technique for relevant and useful content recommendation.

5. THREATS TO VALIDITY

In our research, we note a few issues worthy of discussion. First, the lack of a fully-fledged user study for evaluating usability of the technique is a potential threat. However, our objective was to focus on the technical aspects of the approach. Furthermore, in order to at least partially evaluate the usability, we conduct a limited user study with five participants, where three of them have professional software development experience. We ask them six questions about our relevance visualization, highlighting of the artifacts of interest, and IDE-based information search. Five out of five participants responded and suggested that the proposed technique is likely to be really helpful in extracting the desired information from a web page. However, a fully-fledged user study is required to explore the actual usability of our technique that we consider as a scope of future study.

Second, the dataset (Section 4.1) prepared for evaluation and validation may contain subjective bias. In order to reduce the bias, we perform cross-validation with the help of peers, analyze their suggestions and then finalize the gold set. More importantly, we conduct a case study with StackOverflow pages, where gold sets are prepared by exploiting the feedback from thousands of technical users. The study also demonstrates the potential of our technique against two traceability link recovery techniques– LSA and VSM.

Third, metric weights are estimated using a limited training dataset (Section 3.5) that might cause weight overfitting. However, we also tune and test the weights significantly against different set of pages to mitigate the threat.

6. RELATED WORK

A number of existing studies are conducted on web page content extraction, and they apply different techniques such as template or similar structure detection [6, 14], machine learning [8, 9, 15], information retrieval, domain modeling [10], and page segmentation and filtration [7, 8, 12, 13, 18, 22]. The last group of techniques using page segmentation and noise filtration are closely related to our research in terms of working methodologies although they are driven by different goals. In order to extract *noise-free* content from a web page, they apply several density metrics and link element based heuristics. On the other hand, we complement those density metrics, introduce novel relevance metrics, and

then combine both metrics for relevant section extraction from a web page. In particular, we recommend relevant sections from the page for an encountered exception in the IDE. Sun et al. [22] exploit link elements (e.g., `<a>`, `<input>` tags) for the filtration of noisy sections in a web page. This is probably ideal for news-based websites. However, the idea may not be properly applicable for programming related web sites as our experimental results suggest (Section 4.4). The technique by Pinto et al. [18] is actually designed with a table-based architecture of the web page in mind, which may not be applicable for modern complex websites. The two versions of *Code Content Blurring* by Gottron [12] are only tested against the news-based websites containing simple structures and homogeneous texts.

The other studies use different methodologies that are not closely related to our work, and we do not compare against them in our experiments. Furche et al. [10] analyze real estate websites and extract property or price related information. They exploit a domain-specific model for content extraction which might not be applicable for programming related websites. Chun et al. [8] analyze news-based websites, extract different densitometric features, and apply a machine learning classifier (C4.5) for classifying the legitimate and noisy content sections. Their approach is subject to the amount and quality of training data as well as the performance of the classifier. Cafarella [6] focuses on Wikipedia pages, identifies the special structures (e.g., tabular), and mines different factual information (e.g., list of American presidents) from the pages. Thus, while other techniques focus on extracting the noise-free sections or mining the factual or commercial data from news, real estate or Wikipedia pages, our technique attempts to support software developers in collecting relevant information for problem at hand from the programming related web pages.

This work is to some extent similar to our previous work–SurfClipse [20] since both of them analyze exceptions and web pages. However, this work–ContentSuggest is also significantly different from our previous work that returns a list of relevant pages for any exception. On the other hand, this work returns the most relevant section(s) from a given web page for the exception of one's interest. From technical point of view, it proposes a novel metric–*content relevance* for relevant section extraction, which was not considered by any of the existing approaches. Our technique not only extracts the noise-free sections but also directs a developer to the right (or relevant) sections in the page by exploiting the details of an exception encountered in the IDE, which is a novel idea of developer support, and is not provided by any of the existing approaches.

There exist also several studies [3, 4, 17] in the literature that use information retrieval techniques for traceability link recovery, and they are also related to our work to some extent. While most of them focus on establishing links from software artifacts such as source code to software documentations or requirement documents, we attempt to link an encountered exception (and its details) to the most relevant or the most helpful section from a given web page. For detailed comparison with information retrieval techniques, we refer the readers to Section 4.5.

7. CONCLUSION

To summarize, we propose a novel recommendation technique that recommends the most relevant section(s) from a given web page for an encountered exception in the IDE. Experiments with 250 web pages and 80 programming exceptions show that our technique can extract relevant content with a precision of 81.96%, a recall of 76.74% and a F_1-measure of 76.30%, which are promising. Comparison with the only available closely related technique also shows that our technique performs significantly better in all performance metrics. Finally, a case study with StackOverflow pages, where we compare with two state-of-the-art traceability link recovery techniques, shows that our technique is highly promising in identifying the top-scored answer posts from a StackOveflow Q & A page by using the proposed metrics as well.

References

[1] ContentSuggest Web Portal. URL `http://www.usask.ca/~mor543/contentsuggest`.

[2] Stackoverflow Post. URL `http://stackoverflow.com/questions/17485297`.

[3] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *TSE*, 28(10):970–983, 2002.

[4] G. Bavota, A. De Lucia, R. Oliveto, A. Panichella, F. Ricci, and G. Tortora. The Role of Artefact Corpus in LSI-based Traceability Recovery. In *Proc. TEFSE*, pages 83–89, 2013.

[5] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.

[6] M.J. Cafarella. *Extracting and Managing Structured Web Data*. PhD thesis, 2009.

[7] D. Cai, S. Yu, J. Wen, and W. Ma. Extracting Content Structure for Web Pages Based on Visual Representation. In *Proc. APWeb*, pages 406–417, 2003.

[8] Y. Chun, L. Yazhou, and Q. Qiong. An Approach for News Web-Pages Content Extraction Using Densitometric Features. In *Advances in Electric and Electronics*, volume 155, pages 135–139. 2012.

[9] B.D. Davison. Recognizing Nepotistic Links on the Web. In *Proc. AAAI*, pages 23–28, 2000.

[10] T. Furche, G. Gottlob, G. Grasso, G. Orsi, C. Schallhart, and C. Wang. Little Knowledge Rules the Web: Domain-centric Result Page Extraction. In *Proc. RR*, pages 61–76, 2011.

[11] D. Gibson, K. Punera, and A. Tomkins. The Volume and Evolution of Web Page Templates. In *Proc. WWW*, pages 830–839, 2005.

[12] T. Gottron. Content Code Blurring: A New Approach to Content Extraction. In *Proc. DEXA*, pages 29–33, 2008.

[13] M. Kim, Y. Kim, W. Song, and A. Khil. Main Content Extraction from Web Documents Using Text Block Context. In *Proc. DEXA*, pages 81–93. 2013.

[14] C. Kohlschutter, P. Fankhauser, and W. Nejdl. Boilerplate Detection Using Shallow Text Features. In *Proc. WSDM*, pages 441–450, 2010.

[15] N. Kushmerick. Learning to Remove Internet Advertisements. In *Proc. AGENTS*, pages 175–181, 1999.

[16] C. Le Goues and W. Weimer. Measuring Code Quality to Improve Specification Mining. *TSE*, 38(1):175–190, 2012.

[17] A. Marcus and J.I. Maletic. Recovering Documentation-to-Source-Code Traceability Links Using Latent Semantic Indexing. In *Proc. ICSE*, pages 125–135, 2003.

[18] D. Pinto, M. Branstein, R. Coleman, W. B. Croft, M. King, W. Li, and X. Wei. QuASM: A System for Question Answering Using Semi-structured Data. In *Proc. JCDL*, pages 46–55, 2002.

[19] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.

[20] M.M Rahman, S. Yeasmin, and C. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE SEW*, pages 194–203, 2014.

[21] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proc. ICPC*, pages 172–181, 2008.

[22] F. Sun, D. Song, and L. Liao. DOM Based Content Extraction via Text Density. In *Proc. SIGIR*, pages 245–254, 2011.