Recommending Main & Relevant Content from Web Pages about Programming Errors and Exceptions

Mohammad Masudur Rahman Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
{masud.rahman, chanchal.roy}@usask.ca

***Abstract*—During development and maintenance of a software product, software developers often search for relevant information in the web about an encountered error or exception, where they manually check the web pages returned by a search engine in order to extract a working solution. Both manual checking of the page content against the exception (and its context) and extracting an appropriate solution are non-trivial tasks. They are even more complicated with the bulk of noisy (e.g., advertisements, navigation menus) and irrelevant content in the web page. In this paper, we propose an IDE-based *context-aware* page content recommendation approach that not only returns all the noise-free (i.e., main) sections of a web page but also identifies the relevant sections of the page. The approach exploits the encountered exception and its context in the IDE in order to recommend the relevant sections of the web page resulted from an IDE-based web search. An extensive evaluation with 500 web pages related to 150 programming errors and exceptions and comparisons against four existing approaches show that the proposed approach is highly promising in terms of *recall* and F_1*-measure* with comparable *precision*.**

***Index Terms*—Content density, Content relevance, Content recommendation**

I. 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 relevant information in order to solve them. According to the study of Brandt et al. [6], developers spend about 19% of their programming time in web surfing for required information. While collecting information using search engines, they face different practical challenges such as choosing the most relevant results from hundreds of search results, and extracting useful or relevant content from a selected web page. While the first challenge can be partially handled with reliable ranking from the search engines, the latter task is performed manually by the developers. Both manual checking of the content of a web page for relevance against an encountered exception (and its context) and extracting an appropriate solution are non-trivial tasks. They are even further complicated with the bulk of irrelevant information (e.g., navigation menus, advertisements, copyright notice and so on) in the pages. Such information (e.g., navigational menus) is expected by the site owners, and it also benefits user browsing. However, for the developers, those content sections are completely irrelevant for the programming problem (e.g., an error or an exception) at hand, and thus they can be considered as *noise* [23]. As early as 2005, Gibson et al. [12] estimated that about 40%-50% of web data could be attributed as noise, and they suggested that the ratio is constantly increasing due to explosive growth of Internet and advances in information technology. Thus, the developers often spend a significant amount of time and effort in searching and then extracting the content of interest from a selected web page. However, recommendation of relevant content sections from the selected web page can help them get rid of information overload and can assist them in solving the programming problems with reduced cognitive effort.

A number of existing studies concentrate on web page content extraction applying different approaches such as template or similar structure detection [7, 16], machine learning [9, 10, 17], information retrieval, domain and context modeling [11], and page segmentation and filtration [8, 9, 13, 15, 19, 23]. While these state-of-the-art approaches have their inherent strengths, they also suffer from a number of limitations, and are only partially applicable to our research problem. First, most of the approaches are domain or template specific, and they extract content from the web pages of different domains such as news, consumer products, business and real estates, Wikipedia and social networking. However, no existing studies deal with programming related web pages. Second, programming sites and blogs often discuss a wide range of technical topics in the page, and not all of them are relevant to the programming problem (e.g., an error or an exception) at hand. The existing approaches fail to direct one to the right (or relevant) page sections, and thus, in practice, they neither help much in finding the solution nor can help in reducing the information overload.

In this paper, we propose an *IDE-based context-aware* novel web page content recommendation approach that extracts not only all the noise-free sections of a page but also identifies and recommends the page sections relevant to the encountered programming problem (e.g., an error or an exception) in the IDE. The approach discards noisy sections by exploiting the proposed metrics and Document Object Model (DOM) tree structure of the web page, and recommends relevant sections by exploiting the technical details of the exception in the IDE (i.e., context-aware). We integrate *Google search API* [2] in the IDE for web search, and the search results are exploited by the proposed approach for content recommendation (i.e., IDE-based). The approach also overcomes the limitations of the existing approaches. First, the density metrics–*text density*

and *link density* proposed by existing literature [9, 15, 23] do not differentiate between regular texts (e.g., news article) and programming related content (e.g., source code segments, stack traces) in a web page, which is essential for effective content recommendation from programming related web pages. Our approach considers a novel density metric related to such content– *code density* to complement the existing density metrics. Second, in programming Q & A sites (e.g., StackOverflow), users often post solutions to a programming problem as links, which help others in solving the problem. However, existing approaches annotate any link elements as *noise*, and use *link density* in order to filter out noisy segments, which are not properly applicable in this regard. Our approach proposes a modified *link density* measure to address this issue. Third, it proposes a novel idea in extracting and recommending content from a web page by leveraging *content relevance*. It considers the relevance of different sections of the page against a programming problem (e.g., an error or an exception) encountered in the IDE, and recommends the most relevant sections to the developers. The idea is to allow them to analyze the relevance of any page in the search result without manually checking the whole page as well as to direct them to the right (or relevant) sections. More interestingly, the approach estimates the relevance of each page exploiting the meta description of the page from the search engine, visualizes the relevance, and helps developers choose the page in the first place. Thus the approach helps the developers solve the encountered errors and exceptions with reduced effort.

We conduct experiments on our approach using a collection of 500 programming related web pages and 150 programming errors and exceptions. We manually develop two gold sets– *main-gold-set* and *relevant-gold-set* by carefully extracting all the noise-free and relevant sections of the pages respectively, and use them as *oracles* in order to evaluate our approach. In case of main (i.e., noise-free) content extraction, our approach extracts content from the web pages with a *precision* of 89.88%, a *recall* of 87.48% and a F_1*-measure* of 87.53%, which are highly promising according to existing relevant literature [9, 15, 23]. In case of relevant content recommendation, the approach recommends content with a *precision* of 80.50%, a *recall* of 78.39%, and a F_1*-measure* of 76.40%, which are also promising. We compared against four existing approaches, and found that our approach outperformed them in terms of *recall* and F_1*-measure* with comparable *precision* for the first case and in terms of all three performance metrics for the second case.

II. Motivating Example

Let us consider a web search scenario, where a developer looks for an appropriate solution to solve an encountered programming exception in the IDE. Once she searches with the error message of the exception (e.g., first line in Fig. 1-(c)), traditional web search (e.g., Google search) returns a ranked list of results with minimal or not much helpful hints about the content of the resultant page (Fig. 1-(a)). The developer then selects each of the top-ranked results, skims

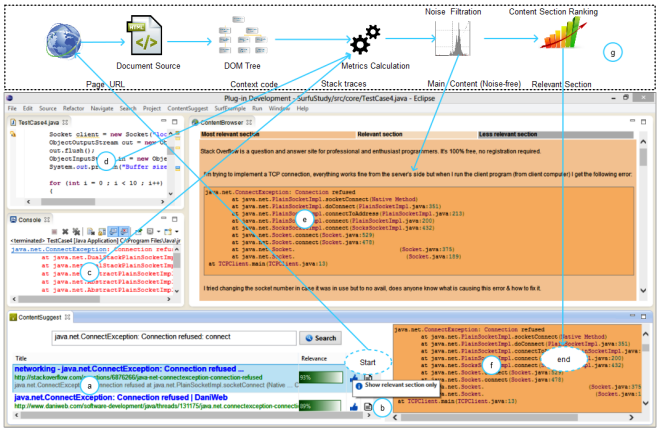

Fig. 1. Schematic Diagram of Proposed Approach

```
<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>
```

Listing 1. An Example HTML Segment

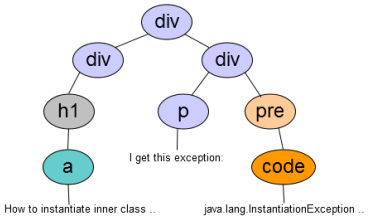

Fig. 2. DOM Tree of Example in Listing 1

through the page, and looks for relevant and interesting content sections (e.g., Fig. 1-(f)). However, she finds it difficult, time-consuming and not much productive. First, the traditional search engine is not aware of the detailed context of the exception at hand, and the ranking of the results may not reflect their relevance against the exception in the context. This forces the developer to browse through a number of resultant pages manually. Second, although manual analysis provides the required information; in practice, each result page contains a large amount of texts and multimedia content with a bulk of irrelevant and noisy elements, and the analysis costs a significant amount of development time and effort [6]. Our proposed approach (1) estimates the relevance of each page against the target exception (and its context), and visualizes the relevance estimate of the page (Fig. 1-(b)), (2) recommends the most relevant content sections of the page containing program elements (e.g., stack traces, code segment) of interests (Fig. 1-(f)), and (3) provides the noise-free content sections of the page with the visualization of their estimated relevance (Fig. 1-(e)) for detailed manual analysis. The idea is to partially automate the mining of desired information, and help developers work out the solutions with reduced effort.

III. Background

Document Object Model (DOM): It is a cross-platform and language independent convention to represent the content (i.e., objects) of an HTML or XML document. In the model, a document is considered 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 1 shows the title and body of a programming question posted on StackOverflow Q & A site, and Fig. 2 shows the corresponding DOM tree. In our research, we use *Jsoup* [4], a popular Java library, in order to parse and analyze the DOM tree of a web page.

Cosine similarity: It is a measure which is frequently used in information retrieval in order to determine the similarity between two text documents. In our research, we use cosine similarity measure in order to determine lexical similarity between the context (e.g., stack traces, context code) of a programming error or exception and the discussion text in a candidate section of a web page. We consider each of the problem context and discussion texts as a *bag of tokens* (A collection of tokens with no fixed order), 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 (e.g., java.io.IOException) structure. 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 (e.g., exception context), and B_i represents that frequency in set B (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 (i.e., content) relevance between the context of an exception and the candidate section of a web page.

Logistic regression: It is a probabilistic statistical classification model that predicts binary or categorical outcomes based on a set of *predictor variables* (e.g., 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 in order to estimate the relative weights (e.g., importance) of different *density* and *relevance* metrics for page content extraction (Section IV-D). Logistic regression models the probabilities of different outcomes of 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 outcomes for the trial.

IV. Proposed Approach

A. Working Modules

In Fig. 1, the schematic diagram of our proposed approach shows the working modules, and explains different steps required for page content extraction, recommendation and visualization. We package the whole solution as an installable Eclipse plug-in prototype [3], and it has three modules as follows:

Content Collector: The proposed approach exploits the technical details of an encountered programming error or exception in order to recommend relevant content sections from a selected web page. The *collector module* collects exception message and stack traces from the active *Console View* (Fig. 1-(c)), and context code (i.e., a segment of the source code that generates the exception) from the active text editor (Fig. 1-(d)) of the IDE. It also collects the HTML source of a resultant web page (Fig. 1-(a)). Once a developer selects a resultant page and requests for relevant or noise-free main content, the *collector module* downloads the HTML source of the corresponding page, and sends HTML source as well as the exception details to the *extractor module*.

Content Extractor: The *extractor module* (i.e., dashed rectangle, Fig. 1-(g)) parses each of the tags of a HTML document, and develops a DOM tree. It then analyzes each of the tree nodes, calculates its *content density* and *content relevance* (Section IV-B), and assigns a *content score*. The module then discards the noisy nodes based on the calculated scores, and returns the noise-free main content. It also identifies those DOM tree nodes most relevant to the encountered exception in the IDE, and recommends them to the developers. It should be noted that the *extractor module* returns all the content as HTML source so that the organization of texts and other elements are preserved during visualization.

Content Visualizer: The visualizer module consists of two content visualization panels– one for relevant content and one for main (noise-free) content. We use SWT browser widget in order to display the HTML content within the scope of the IDE. The relevant content panel (Fig. 1-(f)) displays only the most relevant sections of a web page recommended by the *extractor module*. The panel also highlights different program elements of interest (e.g., stack traces, code segment) in those sections. The idea is to help a developer instantly decide if the page is worth viewing or not. Thus, the developer saves development time and effort in choosing the appropriate solution for the exception at hand. Once she is convinced by the most relevant sections, she can check the whole content of the page in the main content panel (Fig. 1-(e)). The main content panel displays all the noise-free sections of the web page, and visualizes the estimated relevance of each of the sections. It exploits the relevance metrics calculated by the *extractor module*. At present, besides code related elements, the panel visualizes three types of sections–*most relevant, relevant* and *less relevant* with three different colours (Fig. 1-(e)). The idea is to direct the developer to the interesting sections in the web page so that she can devise her solution

for the encountered exception without consulting the whole content of the page. It should be noted that the result panel (Fig. 1-(a, b)) also visualizes the estimated relevance of each resultant web page against the target exception by analyzing the meta description of the page from the search engine. This visualization helps the developer to choose the prospective result pages in the first place during searching for a solution.

B. Proposed Metrics

1) Content Density (CTD): In order to extract the main (i.e., noise-free) content of a web page, existing studies [9, 15, 23] propose two density metrics–*text* or *word density*, and *link density*. They use the heuristic that any link element is a part of advertisements or navigation menus, and therefore, it should be considered as *noise*. In their study, they focus on different news websites, which are mostly of simple structures containing regular texts (e.g., news articles), and the density metrics are designed accordingly. However, programming related web sites are more complex in structure, and they contain different important elements (e.g., code segments, stack traces) beyond regular texts. Besides, the heuristic about the linked elements may also not be applicable. In this section, we modify existing density metrics, add a new metric, and 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. In the DOM tree (Fig. 2), *text density* (TD_i) of a node is calculated considering 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. Existing literature [15, 23] 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 context of an exception of interest, where the context is represented as a set of tokens collected from stack traces and context code of the exception (Section V-C). We consider a linked text element as *noise* only if the relevance of the element is below a threshold (η=0.75); otherwise, we consider it as a legitimate textual element. Thus, in the DOM tree (Fig. 2), the *link density* (LD_i) of a node i is calculated considering 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. [23] 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 websites generally contain different program elements such as *stack traces*, *code segments*, and *class* or *method* names, and they are of significant interest to the developers. The developers often analyze or reuse them in order to solve 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 indication of legitimacy of a programming related web page. In the DOM tree, the code density (CD_i) of a node i is calculated 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.

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

We observe that code related elements are generally posted in the web page using <*code*>, <*pre*> and <*blockquote*> tags, and we consider their texts as code related texts in density calculation.

While *text density* metric represents a generalized form of density for all kinds of text, *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 composite density metric called *content density* (CTD_i), which is partially motivated by the idea of Sun et al. [23].

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

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., title of a web page) from getting an extremely lower value. Moreover, we introduce the programming text proportion of a tag, $\frac{CD_i}{TD_i}$, which supports the overall *content density* for the HTML tag that contains both programming related texts and comprehensive regular texts.

2) Content Relevance (CTR): Existing studies [9, 15, 23] exploit *text density* and *link density* based heuristics in order to determine the legitimacy of a text block in the web page, and they extract the noise-free (i.e., main) content sections from the page. However, they do not necessarily help the developers with information overload. Most of the web pages contain a lot of texts in different sections, and not all sections are relevant to the programming problem (e.g., an error or an exception) at hand. In our research, besides the density based metrics, we consider the relevance of a text block against the target programming problem in order to determine the

legitimacy of the text block. In this section, we discuss two of our proposed relevance metrics, which are partially motivated by our previous work [21] on the development of meta search engine for programming errors and exceptions.

Text Relevance (TR): *Text relevance* metric estimates the relevance of any textual content within an HTML tag against the exception of interest. We represent the exception and its context as a set of suitable keywords collected from corresponding stack traces and context code (Section V-C). We then calculate the *cosine similarity* (Section III) between those keywords and the textual content of the HTML tag. Since *cosine similarity* is an estimate of lexical similarity between two text segments, we consider the calculated measure as an estimate of lexical relevance of the HTML tag against the target exception. The measure values from zero to one, where one refers to complete lexical relevance and vice versa.

Code Relevance (CR): *Code relevance* metric is an estimate of relevance of any code segment or stack trace block embedded within an HTML tag against the exception of interest. In the DOM tree, in order to calculate the *code relevance* of a node (CR_i), we find out the <*code*>,<*pre*> and <*blockquote*> tags under the node, and analyze their content. These tags generally contain stack traces and code segments, and we use a set of regular expressions to identify the stack traces in the tag content. We then apply two different techniques in order to determine the relevance estimates of stack traces and code segments against the exception.

Stack traces of a programming exception contain the error message and a list of method call references pointing to the possible source locations that trigger the exception. Rahman et al. [21] propose a lexical similarity based stack trace matching approach, where they collect package name, class name and method name from each of the method call references, and develop a token list that represents a stack trace block.They then apply *cosine similarity* (Section III) between the token lists of two stack traces in order to determine their matching. We follow the similar approach to determine the relevance of a stack trace block in the web page against that of the target exception.

In order to determine the relevance of a code segment within the HTML tag against an exception of interest, we collect the corresponding context code, and apply a state-of-the-art code clone detection technique by Roy and Cordy [22]. The technique determines the longest common subsequence of source tokens (S_{lcs}) between two code segments. We use it in order to determine the code similarity of the code segment in the page against the context code as follows, where S_{total} refers to the set 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 (i.e., stack traces and code segment) under an HTML tag are estimated, we average the estimates, and consider it as the *code relevance* (CR_i) for the tag i as follows, where N is number of program elements found under the tag in the DOM tree.

$$CR_i = \frac{1}{N}\sum_{j=1}^{N} CR_j \tag{8}$$

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:

$$CTR_i = \alpha \times TR_i + \beta \times CR_i \tag{9}$$

Here α and β are the relative weights (i.e., importance) of the corresponding relevance metrics, which are estimated using a machine learning based approach (Section IV-D) involving logistic regression.

C. Content Score Calculation

In our research, we consider two different aspects– *content density* and *content relevance* of a text block in the web page in order to determine its legitimacy in the noise-free main content as well as relevant content. While the density metric focuses on the amount of textual content, relevance metric checks the relevance of the content against the programming problem at hand. The idea is to provide the developers with not only noise-free but also relevant content so that they can find the required information with reduced effort. We combine both of those aspects, and propose a composite score metric called *content score* (CS_i) for each of the HTML tags in the web page as follows:

$$CTS_i = \gamma \times CTD_i + \delta \times CTR_i \tag{10}$$

Here γ and δ are the relative weights (i.e., importance) of the corresponding density and relevance metrics, which are estimated using a machine learning based approach (Section IV-D).

D. Metric Weight Estimation

In order to determine the 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, and collect the corresponding metrics of 705 text blocks by our approach from the pages. It should be noted that the individual relevance metrics are treated equally in terms of relative importance in the calculation of *content relevance* metric at this stage. We also identify whether each of the blocks is included in the main (i.e., noise-free) content or not, which provides a *binary class label* against the set of features (i.e., metrics) of the text block. Ideally, the classification is supposed to be done through manual analysis [18]; However, it is a non-trivial task, and we exploit our algorithm for classification. We then apply *Weka* [5], a machine learning classifier, with *logistic regression model* on the dataset of 705 block samples containing feature values and class labels. While considering the individual relevance metrics, the machine learning classification agrees with 83.33% of the sample labels, which is promising. However, we are interested to estimate the relative weight (i.e., importance) of the metrics of the text block.

TABLE I
DATASET STATISTICS

Dataset	SO[1]Pages (DSO)	Non-SO Pages (¬DSO)	All Pages (D)	Exceptions
Main set	208 (41.60%)	292 (58.40%)	500 (100%)	150
Relevant set	101 (40.40%)	149 (59.60%)	250 (100%)	80

[1] SO: StackOverflow

In association with classification results, *Weka* also reports the coefficient of each predictor variable (i.e., metric) in the regression model, which are tuned by the classifier tool in order to classify a sample with maximum accuracy. We believe that these coefficients are an estimate of the importance of the features used in the classification, and we consider them as the weights of the individual relevance metrics [18]. For the sake of simplicity and in order to reduce bias, we normalize those coefficients, and consider a heuristic weight α=1.00 for *text relevance* and β=0.59 for *code relevance* metrics. In case of composite density and relevance metrics, we find that the proposed approach performs significantly well with equal relative weights assigned. Thus, we consider a heuristic weight of 1.00 for both of the composite metrics–*content density* and *content relevance*.

E. Document Content Extraction

Once the *content score* of each of the tags in the HTML document is calculated, the noisy elements are discarded based on a *heuristic threshold*. Since the <*body*> tag contains the whole content and encloses all other tags in the document, its *content score* sums up the average density and average relevance estimates of the whole page. As Sun et al. [23] suggest in case of *text density*, in our research, we consider the *content score* of the <*body*> tag as the *threshold score*, and exploit the DOM tree structure of the HTML document to discard the noisy elements. An HTML document is generally divided into a set of identifiable blocks which are represented as the child nodes of *body* node in the DOM tree. We check each of the child nodes for its *content score*, and discard the ones having score less than the threshold. We then explore each of the child nodes having *content score* greater than the threshold, and find out its *inner node* with the highest score. The highest score of the node indicates that the corresponding tag in the HTML document contains the most legitimate content in terms of different density and relevance estimates. In order to extract the main (i.e., noise-free) content, we keep the highest scored node along with its child nodes, and mark them as *content node*. We apply the same process recursively for each node in the DOM tree, and finally we get any nodes in the tree annotated as either *content* or *noise*. We then discard the noisy nodes, and extract the HTML tags corresponding to the remaining nodes in the tree as the main (i.e., noise-free) content of the document.

In case of the relevant content recommendation, we analyze each of the child nodes under body node in the DOM tree, and choose the *highest scored relevant node*. The idea is to ensure that the recommended sections (i.e., nodes) are not only relevant but also are rich in legitimate content. In this case, we use the normalized version of the density and relevance metrics for the *content score* calculation to ensure their equal influence, whereas we use the original metrics in case of score calculation involving main content extraction (Section IV-C).

V. EXPERIMENT

A. Dataset Preparation

We use a dataset of 500 web pages (hereby *main set*) related to 150 programming errors and exceptions [1], a subset of *main set* containing 250 web pages (hereby *relevant set*), and the technical details (e.g., error messages, stack traces and context code) of the exceptions. Both the technical details and the web pages are collected from our previous work [21] on the recommendation of relevant web pages about programming errors and exceptions. We develop two gold sets– *main-gold-set* and *relevant-gold-set* [1] in order to evaluate our approach. The first gold set contains the main (i.e., noise free) content of each page in *main set*, whereas the second set contains the most relevant content sections of each page in *relevant set*. Both gold sets are developed through extensive manual analysis, and we spent about 40-50 working hours. One of the authors carefully extracts the content from the web page with the help of *design view* of *Dreamweaver* editor, which are then validated by the peer. We consider different elements such as advertisements, navigation menus, copyright notice, repeated discussion texts, and insignificant comments as noisy content, and we discard them in order to obtain the main content. In case of relevant content extraction, we analyze each of the content sections in the page, and attempt to determine their relevance against the exception of interest from the point of view of problem solving. We note that most of the pages in the dataset generally start with a page section that describes a programming problem involving an error or exception. The section contains a question title, and different program elements such as stack traces and code segments. We look for such sections in the pages and extract them as the relevant sections. It should be noted that the web pages were chosen based on their relevance in the first place, and we identify the most relevant section of a page by applying our best judgement and programming experience.

Both *main set* and *relevant set* contain about 40% of the pages from StackOverflow Q & A site, and we were interested to contrast them with other pages. We break each set into two more subsets–*StackOverflow pages (DSO)* and *Non-StackOverflow pages (¬DSO)*, and thus, we use three sets–*DSO, ¬DSO* and *D* for each of *main* and *relevant content* extraction, where *D* combines both *DSO* and *¬DSO* sets. Table I shows the statistics of the different sets of web pages.

B. Traditional Web Page Content Recommendation

We investigate the traditional content recommendation supports by the search engines such as Google, Bing and Yahoo. Each of the search providers returns the search results in the ranked list with the page title and a minimal description of the page usually collected during page indexing. They also highlight the terms in the title and the description those directly match with the search query terms. However, the limited overview and information highlighting are not often enough,

```
Exception in thread "main" java.io.EOFException
at java.io.ObjectInputStream$PeekInputStream.readFully(
    ObjectInputStream.java:2281)
at java.io.ObjectInputStream$BlockDataInputStream.readShort
    (ObjectInputStream.java:2750)
at java.io.ObjectInputStream.readStreamHeader(
    ObjectInputStream.java:780)
at java.io.ObjectInputStream.<init>(ObjectInputStream.java
    :280)
at HighScores.<init>(HighScores.java:45)
at HighScores.main(HighScores.java:151)
```

Listing 2. Stack Traces of an Exception

```
FileInputStream fis = new FileInputStream(file);
ObjectInputStream ois = new ObjectInputStream(fis);
ArrayList<Record> currentList = new ArrayList<>();
// restore the number of objects
int size = ois.readInt();
// restore the objects
for (int i = 0; i < size; i++) {
  Record current = (Record) ois.readObject();
  currentList.add(current); }
```

Listing 3. Context code of the Exception

and a developer needs to browse the resultant page in order to determine its relevance against an exception of interest in the context. During the browsing, one can exploit the *search feature* provided by the browser applications to complement the search engines; however, still it is not much helpful to find out the most relevant or interesting sections from a web page.

C. Exception Context Representation

In our research, we not only take the density metrics but also the relevance estimates of each of the page sections into consideration during main and relevant content extraction. Each page in the dataset is relevant to a particular exception, and we exploit the details (e.g., stack traces and context code) of that exception for relevance estimation of the page sections. We analyze the stack traces (e.g., Listing 2) and extract different tokens such as *package name*, *class name* and *method name* from each of the method call references in the traces. We also analyze the context code (e.g., Listing 3) of the exception and collect *class* and *method name* tokens. We use *Javaparser*[1] for compilable code and an *island parser* for non-compilable code in order to extract the tokens [21]. We then combine token sets both from the stack traces and context code, and appends the exception name along with the exception message (i.e., highlighted line of the traces in Listing 2) to the combined set in order to represent the *context of the exception* of interest. For example, Listing 4 shows the context representation by our approach for an *EOFException* with stack traces in Listing 2 and context code in Listing 3.

D. Performance Metrics

Our proposed approach is greatly aligned with the research areas of information retrieval and recommendation systems, and we use a list of performance metrics from those areas in order to evaluate our approach as follows [14, 21, 23].

Mean Precision (MP): *Precision* determines the percentage of the retrieved content that is expected (e.g., main content, relevant content) from a web page. In our research, we compare the retrieved content by our approach against the manually prepared gold sets. As Sun et al. [23] suggest, we use the *longest common subsequence* of tokens between retrieved content and gold content. Thus *precision* can be determined as follows, where *a* refers to the token sequence of retrieved main or relevant content and *b* refers to that of the corresponding gold content.

$$P = \frac{|LCS(a,b)|}{|a|}, \quad MP = \frac{\sum_{i=1}^{N} P_i}{N} \tag{11}$$

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

Mean Recall (MR): *Recall* measure determines the percentage of the expected content (e.g., main content, relevant content) of a web page that is retrieved by a system. We calculate the *recall* of a system as follows:

$$R = \frac{|LCS(a,b)|}{|b|}, \quad MR = \frac{\sum_{i=1}^{N} R_i}{N} \tag{12}$$

Mean Recall (MR) averages the *recall* measures for all the 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 system, F_1-measure is a combined and more meaningful metric for evaluation[2]. We calculate F_1-measure from the harmonic mean of *precision* and *recall* as follows:

$$F_1 = \frac{2 \times P \times R}{P + R}, \quad MF = \frac{\sum_{i=1}^{N} F_{1i}}{N} \tag{13}$$

Mean F_1(MF) averages all such measures.

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

Listing 4. Context of the Exception

TABLE II
EXPERIMENTAL RESULTS OF PROPOSED CONTENT EXTRACTION APPROACH

Content Type	Metric	SO Pages (DSO)	Non-SO Pages (¬DSO)	All (D)
Main content	MP	91.27%	88.90%	89.88%
	MR	89.27%	86.20%	87.48%
	MF	90.00%	85.76%	**87.53%**
Relevant content	MP	89.91%	74.12%	80.50%
	MR	74.90%	80.76%	78.39%
	MF	80.07%	73.91%	**76.40%**

E. Experimental Results

We conduct experiments with two datasets– *main set* and *relevant set*, and extract main (noise-free) content and relevant content respectively from their pages. We then check those extracted content against the carefully prepared *main-gold-set* and *relevant-gold-set* respectively, and evaluate the performance of our approach. Table II and Table III summarize the findings of our evaluation.

From Table II, we note that our proposed approach extracts main content with a *mean precision* of 89.88% and a *mean recall* of 87.48%, which are highly promising. In case of the main content extraction from a web page, both *precision* and

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

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

TABLE III
EXPERIMENTAL RESULTS FOR DIFFERENT SCORE COMPONENTS

Score Combination	Metric	Main Content			Relevant Content		
		SO Pages (DSO)	Non-SO Pages (¬DSO)	All (D)	SO Pages (DSO)	Non-SO Pages (¬DSO)	All (D)
{Content Density (CTD)}	MP	90.89%	88.86%	89.71%	50.91%	49.50%	50.07%
	MR	89.38%	86.20%	87.53%	91.74%	75.71%	82.18%
	MF	89.85%	85.75%	**87.45**%	62.32%	53.76%	**57.22**%
{Content Relevance (CTR)}	MP	89.80%	75.40%	81.39%	86.63%	69.17%	76.23%
	MR	25.66%	37.83%	32.76%	52.17%	57.66%	55.44%
	MF	33.82%	45.31%	**40.53**%	61.07%	55.88%	**57.98**%
{Density (CTD), Relevance (CTR)}	MP	91.27%	88.90%	89.88%	89.91%	74.12%	80.50%
	MR	89.27%	86.20%	87.48%	74.90%	80.76%	78.39%
	MF	90.00%	85.76%	**87.53**%	80.07%	73.91%	**76.40**%

recall are important, and our approach also performs well in terms of the combined metric, F_1*-measure* (e.g., 87.53%). The *main set* contains about 41.60% of the pages from StackOverflow. During gold set preparation, we notice that the pages from StackOverflow follow a consistent structure with relatively less noise, and relevant content sections are more precise and persistent than those in the pages from other web sites, which are helpful for content extraction. We were interested to check the performance of our approach against three different subsets of *main set* (Table I). We note that the approach performs almost equally well for all the subsets of different types (i.e., websites) and different sizes (i.e., number of pages) (Table I), which demonstrates the robustness and generality of the approach. In case of relevant content, our approach recommends content with a *mean precision* of 80.50%, a *mean recall* of 78.39%, and a *mean* F_1*-measure* of 76.40%. In this case, we use *relevant set* of 250 web pages, and 40.40% of the pages are from StackOverflow. In order to evaluate the recommendation performance against different subsets, we conduct the experiments, and find that the approach provides the most precise recommendation of 89.91% with StackOverflow pages. The aforementioned scenario of StackOverflow might partially help our approach to perform better; however, the approach also recommends with a *mean precision* of 74.12% with non-StackOverflow web pages, which is promising and significantly better than that of the existing approaches (Table IV).

Table III investigates the effectiveness of the two aspects–*content density* and *content relevance* that we propose for content extraction from a web page. We consider each of those aspects in isolation as well as in combination, and evaluate our approach against different sets of web pages for different types of extraction–*main content* and *relevant content*. In case of *content density*, the proposed approach performs significantly well in terms of all three performance metrics for main content extraction with all three subsets–*StackOverflow pages (DSO), Non-StackOverflow pages (¬DSO)* and *D* of *main set*. However, the metric is found not much effective in case of relevant content extraction with any of the subsets of *relevant set*, and the approach provides imprecise results. For example, it extracts the relevant content from a web page of any of the three subsets (*DSO, ¬DSO* and *D*) with a maximum *mean precision* of 50.91% and a *mean* F_1*-measure* of 62.32%. In case of *content relevance* metric, the proposed approach extracts relevant content from a web page with relatively better precision (e.g., 86.63%); however, the recall rates are poor in both main and relevant content extraction. On the other hand, when we combine both the density and relevance metrics, we experience significant improvement in all three performance metrics for both types (e.g., main and relevant) of extraction with each of the sets of web pages. For example, main and relevant content sections of a page are extracted by our approach with a mean F_1*-measure* of 87.53% and 76.40% respectively.

It should be noted that in case of main content extraction, much improvements in performance are not achieved with the combination of metrics than with density metric only. The finding disproves our primary intuition about the effectiveness of relevance metric in main content extraction. However, the finding about the relevant content extraction clearly shows the effectiveness of the combination of density and relevance metrics which is one of our primary objectives of this work.

F. Comparison with Existing approaches

In order to validate the performance of our approach, we compare against four existing approaches [13, 14, 19, 23] for main content extraction, and one existing approach [23] for relevant content extraction from the literature. It should be noted that those four approaches only extract main content from web pages, and the rest one is adapted for relevant content extraction, whereas our approach extracts both main and relevant content from the pages. Table IV summarizes the findings from our comparative study.

Sun et al. [23] propose a main (i.e., noise-free) content extraction approach, where they exploit the DOM-tree structure of a web page and a text density-based scoring technique. From Table IV, we note that our approach performs significantly better than their approach in terms of all three performance metrics with each of the sets (e.g., *StackOverflow pages (DSO), Non-StackOverflow pages (¬DSO*) and *D*) of web pages. For example, with *¬DSO* set containing web pages other than from StackOverflow, the approach by Sun et al. can extract main content with a *mean precision* of 78.70%, a *mean recall* of 75.67% and a *mean* F_1*-measure* of 75.48%. On the other hand, our approach extracts such content from the same dataset with 88.90% *precision*, 86.20% *recall and* 85.76% F_1*-measure*. Their approach performs well only with StackOverflow pages, whereas our approach performs more consistently with different sets of web pages, and also provides relatively more promising results. One can think of the *Density Only* version of our approach equivalent to the approach of Sun et al. from theoretical perspective; however,

TABLE IV
COMPARISON WITH EXISTING APPROACHES

Content Type	Content Extractor	Metric	SO Pages (DSO)	Non-SO Pages (¬DSO)	All (D)	Content Extractor	Metric	SO Pages (DSO)	Non-SO Pages (¬DSO)	All (D)
Main content	Sun et al. [23]	MP	**80.61**%	**78.70**%	79.49%	DSC [19]	MP	98.27%	91.05%	**94.05**%
		MR	86.41%	**75.67**%	80.14%		MR	62.69%	67.56%	**65.53**%
		MF	83.14%	**75.48**%	**78.67**%		MF	74.54%	75.26%	**74.96**%
	ACCB [13]	MP	90.65%	**93.07**%	92.06%	TCCB [14]	MP	**96.47**%	88.89%	92.04%
		MR	77.32%	**79.98**%	**78.87**%		MR	**65.98**%	61.43%	63.32%
		MF	83.07%	84.64%	**83.99**%		MF	76.96%	68.70%	**72.14**%
	Proposed Approach (Density Only)	MP	90.89%	88.86%	89.71%	Proposed Approach (Density and Relevance)	MP	**91.27**%	**88.90**%	**89.88**%
		MR	89.38%	86.20%	87.53%		MR	**89.27**%	**86.20**%	**87.48**%
		MF	89.85%	85.75%	**87.45**%		MF	90.00%	**85.76**%	**87.53**%
Relevant content	Sun et al. [23]	MP	52.63%	38.89%	**44.44**%	Proposed Approach (Density and Relevance)	MP	89.91%	74.12%	**80.50**%
		MR	86.49%	41.84%	**59.88**%		MR	74.90%	80.76%	**78.39**%
		MF	62.57%	34.49%	**45.84**%		MF	80.07%	73.91%	**76.40**%

experimental results (Table IV) show that our density metric is more effective than theirs in extracting the main content from a web page.

Pinto et al. [19] propose another content extraction system that exploits *Document Slope Curves (DSC)* filtration technique and locates the regions of a web page where word tokens are more frequent than HTML tag tokens using a windowing technique. We also compare against this approach, and find that it provides results with relatively higher precisions, but the recall rates are quite poor compared to those of our approach. For example, the approach extracts main content from our dataset with a *mean precision* of 94.05%, but returns only 65.53% (i.e., *recall*) of the expected page content compared to 87.48% (i.e., *recall*) of our approach (Table IV).

Gottron [13] introduces a content extraction algorithm that identifies the homogeneously formatted texts from the HTML source token sequence of a web page. In our research, we compare against two different versions of the algorithm–*ACCB* [13] and *TCCB* [13, 14]. From Table IV, we note that both approaches provide relatively more precise results compared to our approach; however, their *recall rates* are relatively poor. For example, ACCB and TCCB returns 79.98% (i.e., *recall*) and 65.98% (i.e., *recall*) of the page content respectively at a maximum level from any of the datasets, and they extract main content with 93.07% and 96.47% *precisions* respectively. On other hand, our approach returns about 87.48% (i.e., *recall*) page content with 89.88% *precision*, which demonstrate the robustness of our approach. More interestingly, it extracts 89.27% (i.e., *recall*) of the page content with 91.27% *precision* for StackOverflow pages. It should be noted that ACCB is relatively closer to our approach in terms of F_1*-measure* (e.g., about 84%); however, it performs comparatively poor in terms of *recall* which is 78.87%.

We also compare our approach in case of relevant content extraction, where we adapt and implement the approach of Sun et al. [23] in our working environment for comparison. To our knowledge, no existing approaches focus on such content extraction or recommendation despite of its appealing motivation. We find that approach to be the most aligned for such purpose and it also applies metrics similar to ours for main content extraction. We choose the highest scored content section in terms of *text density* [23] as the most legitimate or relevant section of a web page. However, from Table IV, we note that the density-based metric of the existing approach fails to identify and extract the relevant content of a web page satisfactorily. For example, their approach recommends relevant content with a *mean precision* of 44.44% and a *mean recall* of 59.88%, which provide a *mean* F_1*-measure* of 45.84%. On the other hand, our proposed approach recommends such content with a *precision* of 80.50%, a *recall* of 78.39% and a F_1*-measure* of 76.40%. The other approaches [13, 14, 19] are not meant for relevant content extraction; however, the way they extract different content sections from a web page, they are less likely to identify and extract the relevant content sections.

VI. THREATS TO VALIDITY

In our research, we note a few issues worthy of discussion. First, in order to reduce noisy elements, we remove all the <*style*> and <*script*> tags from an HTML page during parsing, which are often responsible for dynamic loading of advertisement banners or irrelevant widgets. Thus, the look and feel of the pages gets changed, which is a potential threat to our work. In order to mitigate that threat, we add a customized style where we highlight different artifacts of interest (e.g., stack traces, code segments, class or method name) and visualize the relevance of each of the content sections in the page against the exception of interest.

Second, our approach provides a heuristic estimate of relevance for each of the web pages in the ranked search results from the search engine (Fig. 1-(b)). One might suggest that using similar heuristics the search results could be ranked against the context of the given error or exception. While this has potential, we note that the search engine (e.g., Google) provides a satisfactory ranking with the error message of the exception as the search query for most of the time. Furthermore, in this research, our objective is to complement the existing ranking with relevance information. The idea is to help the developers with informed decision making during search by removing noisy elements and finding the relevant page sections.

Third, the lack of a fully-fledged user study for evaluating the usability of the approach is a potential threat. However, our objective was to focus on the technical aspects of the approach. Furthermore, in order at least partially evaluate the usability, we conduct a limited user study with five prospective participants, where three of them have professional software development experience. We ask them six questions about the customized style of the page, relevance visualization of each page in the search result and different page sections,

highlighting of the artifacts of interest, and the IDE-based information search. Five out of five participants agree that the proposed approach is likely to be more helpful than traditional browsers in problem solving for the developers. All of them feel comfortable with the modified layout and style, and they even speculate that the relevance visualization feature might be really helpful in extracting the desired information from the web page. The details of this mini user study could be found online [1]. However, a fully-fledged user study is required to explore the true potential of our approach which we consider as a scope of future study.

VII. Related Works

A number of existing studies are conducted on web page content extraction, and they apply different techniques such as template or similar structure detection [7, 16], machine learning [9, 10, 17], information retrieval, domain modeling [11], and page segmentation and filtration [8, 9, 13, 15, 19, 23]. The last group of techniques–*page segmentation and noise filtration* are closely related to our research in terms of working methodologies and goals. Sun et al. [23] exploit link elements for the filtration of noisy sections in a web page. The approach by Pinto et al. [19] 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 [13] are only tested against the news-based websites containing simple and regular structures. We compare against all four of the above approaches, and find that our approach performs relatively better than all of them in terms of *recall* and F_1*-measure*, and also provides comparable *precision*. For a detailed comparison, the readers are referred to Section V-F.

The other studies use different methodologies and are not closely related to our work, and we do not compare against them in our experiments. Furche et al. [11] analyze real state websites and extract property or price related information. They exploit a domain-specific model for content extraction which may not be applicable for programming related websites. Chun et al. [9] analyze news-based websites, extract different densitometric features, and apply a machine learning classifier (C4.5) in order to classify the legitimate and noisy content sections. Their approach is subject to the amount or quality of the training data and the performance of the classifier. Cafarella [7] focuses on Wikipedia pages, identifies the tabular structures, and mines different factual information (e.g., list of American presidents) from the pages. Thus, while other approaches focus on extracting the noise-free sections or mining the factual or commercial data from the news-based, real state or Wikipedia pages, our approach attempts to support software developers in finding context-relevant information from the programming related pages. From technical point of view, it proposes and leverages a novel metric–*content relevance* for content (e.g., relevant) extraction, which was not considered by any of the existing approaches.

VIII. Conclusion and Future Works

To summarize, we propose an *IDE-based context-aware* web page content recommendation approach that not only extracts all the noise-free sections but also recommends the web page sections relevant to an encountered exception in the IDE. It visualizes the relevance of a web page in the search result from the search engine and different sections in the page, and also highlights different programming elements of interest in the page. We conduct experiments with a collection of 500 web pages related to 150 programming errors and exceptions. Our approach extracts main content with a F_1*-measure* of 87.53%, and relevant content with a F_1*-measure* of 76.40%, which are promising. We compare against four existing approaches in case of main content and one approach in the case of relevant content extraction, and find that our approach outperforms them in terms of *recall* and F_1*-measure* for the first case, and in terms of all three metrics for the second case. The experimental results show that while having comparable *precision* our proposed metrics–*content density* and *content relevance* are more effective than existing density-based metrics in identifying and then recommending the relevant sections from a web page. In future, we plan to conduct a fully-fledged user study with prospective participants.

References

[1] Content Suggest Dataset. URL http://www.usask.ca/~mor543/contentsuggest/data.
[2] Google Custom Search Engine. URL https://www.google.com/cse/.
[3] ContentSuggest Eclipse Plugin. URL https://marketplace.eclipse.org/content/contentsuggest.
[4] JSoup: Java HTML Parser. URL http://jsoup.org.
[5] WEKA Tool. URL http://www.cs.waikato.ac.nz/ml/weka.
[6] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proc. SIGCHI*, pages 513–522, 2010.
[7] M.J. Cafarella. *Extracting and Managing Structured Web Data*. PhD thesis, 2009.
[8] 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.
[9] 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.
[10] B.D. Davison. Recognizing Nepotistic Links on the Web. In *Proc. AAAI*, pages 23–28, 2000.
[11] 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.
[12] D. Gibson, K. Punera, and A. Tomkins. The Volume and Evolution of Web Page Templates. In *Proc. WWW*, pages 830–839, 2005.
[13] T. Gottron. Content Code Blurring: A New Approach to Content Extraction. In *Proc. DEXA*, pages 29–33, 2008.
[14] Thomas Gottron. Combining Content Extraction Heuristics: The CombinE System. In *Proc. IIWAS*, pages 591–595, 2008.
[15] 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.
[16] C. Kohlschutter, P. Fankhauser, and W. Nejdl. Boilerplate Detection Using Shallow Text Features. In *Proc. WSDM*, pages 441–450, 2010.
[17] N. Kushmerick. Learning to Remove Internet Advertisements. In *Proc. AGENTS*, pages 175–181, 1999.
[18] C. Le Goues and W. Weimer. Measuring Code Quality to Improve Specification Mining. *TSE*, 38(1):175–190, 2012.
[19] 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.
[20] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.
[21] 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. SEW*, pages 194–203, 2014.
[22] 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.
[23] F. Sun, D. Song, and L. Liao. DOM Based Content Extraction via Text Density. In *Proc. SIGIR*, pages 245–254, 2011.