Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions

Mohammad Masudur Rahman, Shamima Yeasmin, Chanchal K. Roy
Computer Science, University of Saskatchewan, Canada
{mor543, shy942, ckr353}@mail.usask.ca

***Abstract*—Study shows that software developers spend about 19% of their time looking for information in the web during software development and maintenance. Traditional web search forces them to leave the working environment (e.g., IDE) and look for information in the web browser. It also does not consider the context of the problems that the developers search solutions for. The frequent switching between web browser and the IDE is both time-consuming and distracting, and the keyword-based traditional web search often does not help much in problem solving. In this paper, we propose an Eclipse IDE-based web search solution that exploits the APIs provided by three popular web search engines– Google, Yahoo, Bing and a popular programming Q & A site, StackOverflow, and captures the content-relevance, context-relevance, popularity and search engine confidence of each candidate result against the encountered programming problems. Experiments with 75 programming errors and exceptions using the proposed approach show that inclusion of different types of contextual information associated with a given exception can enhance the recommendation accuracy of a given exception. Experiments both with two existing approaches and existing web search engines confirm that our approach can perform better than them in terms of recall, mean precision and other performance measures with little computational cost.**

***Index Terms*—IDE-based search; API mashup; Context-based search; Context code; Cosine similarity;**

I. INTRODUCTION

Studies show that up to 80% of total effort is spent on software maintenance [13]. During development and maintenance of a software product, developers face different programming challenges, and one of the major challenges is software bug fixation. Software bugs are often associated with programming errors or exceptions. Existing IDEs (e.g., Eclipse, Visual Studio) facilitate to diagnose the encountered errors and exceptions, and developers get valuable information for fixation from the stack traces produced by them. However, the information from the stack trace alone may not help enough in fixation, especially when the developers lack necessary skills or the encountered problems are relatively unfamiliar to them. Thus, for fixation, developers often dig into the world wide web and look for more helpful and up-to-date information. In a study by Brandt et al. [7], developers, in average, spent about 19% of their programming time in surfing the web for information. Goldman and Miller [10] conducted a study where they analyzed the events produced by the web browser and the IDE in temporal proximity, and concluded that 23% web pages visited were related to software development.

Determining the working solution to a programming problem from traditional web search involves trial and error approach in keyword selection, and developers often spend a lot of time to look for such solutions. The search forces them to leave the working environment (i.e., IDE) and look for the solutions in the web browsers. It also does not consider the context of the problems that they search solutions for. The frequent switching between IDE and the web browser is both distracting and time-consuming, and the keyword-based web search often does not help them much in problem solving. Moreover, checking relevance from hundreds of search results is a cognitive burden on the developers.

Existing studies focus on integrating commercial-off-the-shelf (COTS) tools into Eclipse IDE [14], recommending StackOverflow posts and displaying them within the IDE environment [9, 13], embedding traditional web browser inside the IDE [8] and so on. Cordeiro et al. [9] propose an IDE-based recommendation system for runtime-exception handling. They extract the question and answer posts from StackOverflow data dump and suggest posts relevant to the encountered exceptions by capturing the exception context from the stack traces generated by the IDE. Ponzanelli et al. [13] propose *Seahawk*, an Eclipse IDE plugin, that captures the context (e.g., source code under editing) of search in terms of several keywords, and recommends StackOverflow posts within the IDE. It also visualizes different components of a recommended post through an embedded and customized web browser for user-friendly use. However, both of the proposed approaches suffer from several limitations. First, they consider only one source (e.g., StackOverflow Q & A site) rather than the whole web for information and thus, their search scope is limited. Second, the developed corpus cannot be easily updated and is subjected to the availability of the data dump. For example, they use the StackOverflow data dump of September 2011, that means, it does not contain the posts generated after September 2011 and therefore, it cannot provide much help or suggestions related to the recently introduced software bugs or errors. Third, they only consider either stack trace or source code under editing as the context of a programming problem for search whereas both of them contain necessary information for fixation. For example, the approach by Cordeiro et al. [9] does not consider the target code that generates the exceptions and therefore, recommends solutions irrespective of the source

code context considering that the same exception might be generated from different context. Similarly, *Seahawk* [13] cannot answer the programming error or exception related questions as it does not consider the associated stack traces produced by the IDE.

In this paper, we propose an IDE-based web search solution, *SurfClipse*, to the encountered errors and exceptions which addresses the concerns identified in case of existing approaches. We package the solution as an Eclipse plugin which collects search results from a remotely hosted web service [4] and displays them within the IDE. The proposed approach (1) exploits the search and ranking algorithms of three reliable web search engines (e.g., Google, Bing and Yahoo) and a programming Q & A site (e.g., StackOverflow) through the use of API endpoints, (2) provides both a content (e.g., error or exception message) relevance score and context (e.g., stack trace, associated source code) relevance score based ranking on the extracted results of step one, (3) facilitates the most recent solutions, accesses the complete and extensible solution set of StackOverflow and pulls solutions from a number of forums, discussion boards, blogs, programming Q & A sites and so on, and (4) demonstrates a potential use of web service technology for problem context-aware web search which can be easily leveraged by any IDE of any framework.

We conduct experiments using *SurfClipse*, and compare with two existing approaches [9, 13] and three traditional web search engines with 75 programming errors and exceptions related to Eclipse plugin framework and standard Java applications. The proposed approach recommends correct solutions for 68 (90.66%) exceptions, which essentially outperforms the existing techniques and keyword-based traditional web search approaches in terms of *recall* and other performance measures. We note from the experiments that neither stack trace nor source code alone can determine the complete context of all occurred exceptions, rather their combination represents a more precise context which is likely to produce more relevant results. This work is a significantly extended and refined version of our earlier work [15], where we just outlined the idea of an IDE-based meta search engine with limited experiments and validations.

The rest of the paper is organized as follows. Section II focuses on the theoretical concepts related our research, Section III-A discusses our proposed system model for IDE-based web search, and Section III-B presents our proposed content-based and context-based metrics and algorithms. Section IV discusses about the conducted experiments and experimental results, Section V identifies the possible threats to validity, Section VI discusses the existing studies related to our research and finally, Section VII concludes the paper with future plan.

II. Background

A. Cosine Similarity

Cosine similarity is a measure that indicates the orientation between two vector spaces with different number of dimensions. It is frequently used in information retrieval to measure the similarity between two text documents where each term is considered as a dimension and each document is considered as a vector of distinct terms. In our research, we often use cosine similarity measure to determine the relevance between the set of query terms and the terms extracted from a candidate result page. We consider each set as a *bag of words* (A collection of words with no fixed order), remove the *stop words* (Insignificant words in a sentence) and then perform *stemming* (A technique to extract the root of a word) on each term which provides their normalized forms. We prepare a combined set of terms, C, from two sets and then apply Equation (1) to calculate the *cosine similarity* score.

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

Here, S_{cos} is the *cosine similarity score*, A_i represents frequency of ith term from C in set A (e.g., search query), and B_i represents that frequency in set B (candidate result document). This measure values from zero (i.e., complete lexical dissimilarity) to one (i.e., complete lexical similarity), and helps to determine the content-relevance between a result page and the search query.

B. Degree of Interest

Cordeiro et al. [9] propose *Degree of Interest Score*, which is associated with the reference terms extracted from the stack trace of an occurred exception. They use the score to represent proximity of a reference to the target exception location. In our research, we also leverage this score to determine the relevance between two stack traces. Suppose, a stack trace has *N* method call references, then *Degree of Interest* score for each reference can be calculated using Equation (2).

$$S_{doi} = 1 - \frac{n_i - 1}{N} \quad (2)$$

Here, n_i represents the position of the reference in stack trace. The score values from zero to one, where zero represents the most distant reference from exception location, and one means the most likely reference that generates the exception.

III. Proposed Approach for IDE-Based Context-Aware Web Search

A. Proposed Model for IDE-Based Search

Fig. 1 shows the schematic diagram of our proposed approach for IDE-based web search. This section discusses the architectural design of our proposed model which includes the working entities, working modes and so on.

1) Web Service Provider and Client Plugin: Our proposed model is based on client-server architecture and it has two major entities– an Eclipse plugin (client) and a web service provider (server). They communicate with each other through hyper text transfer protocol and facilitate the search results within the IDE environment. Once the developer selects an encountered exception from *Console View* or *Error Log* in the IDE, the client plugin collects associated context information (e.g., stack trace and the likely source code causing the exception) and generates a web search request to the service provider [4]. The service provider itself works like a meta

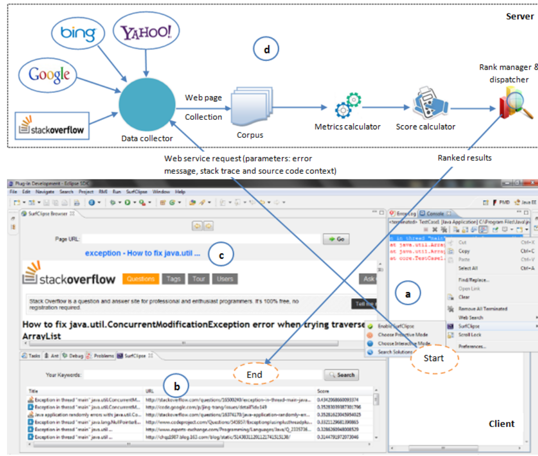

Fig. 1. Schematic Diagram of the Proposed Approach

search engine, that means, it collects results from multiple search engines against a single query and analyzes them to provide an enriched set of results. In our proposed model, *Data collector* module of the service provider collects results from three state-of-art search engines (Google, Bing and Yahoo) and a programming Q & A site, StackOverflow, and then accumulates the results to form a *Corpus*. The corpus is generated from about 100-150 results from different sources, and *Metrics calculator* module computes different proposed metrics (Section III-B) for each result entry in the corpus. The metrics capture the relevance of the result to the content and context of the query exception. Once metrics are computed, *Score calculator* calculates the final scores of each result entry, and *Rank manager and dispatcher* ranks the results and returns them to the client. The client plugin then captures the results and displays within the IDE in a convenient way. It also facilitates the browsing of the result through a customized web browser widget.

2) Plugin Working Modes: Eclipse plugin in the proposed model works in two modes– interactive and proactive. In interactive mode (e.g., Fig. 1-(a)), the developer can select the search query by choosing a suitable phrase from the exception stack trace or associated source code, and can make a web search request to the server (e.g., Fig. 1-(d)). Thus, the mode provides a flexible interface (e.g., Fig. 1-(c)) for web search from within the IDE (e.g., Fig. 1-(b)). In case of proactive mode, the web search request is initiated by the client plugin. In this mode, the plugin assigns a listener to the *Console View* which constantly checks for exception. Once an exception detected, the listener sends error message and other contextual information to the plugin, and then the plugin makes web search request to the service provider. Thus, in this mode, the developer gets rid of the burden of carefully choosing the search query and making the search request manually, and therefore can concentrate on her current tasks without interruption.

3) Corpus Development: Reusing existing data and services to provide an enriched output is an interesting idea, and we use it for the corpus development in our research. We exploit the available API services provided by three popular web search engines (Google, Bing and Yahoo) and a large programming Q & A site, StackOverflow, to collect the top 30-50 ranked results from each of them against the encountered error or exception, and then use them to develop a corpus dynamically. The idea is to leverage the existing search services and their recommendations to reduce the search scope and produce an effective solution set. Unlike a traditional search engine, which develops an index of all the result pages with some sort of relevance score against a query term, we store necessarily the complete HTML source of each result page as it is parsed and analyzed for relevant stack traces, source code snippets, exception messages and so on in the later phases for metrics calculation.

B. Proposed Metrics

1) Search Engine Confidence Score (S_{sec}): According to *Alexa*[1], one of the widely recognized web traffic data providers, Google ranks first, Yahoo ranks fourth, Bing ranks nineteenth and StackOverflow ranks 67th among all websites in the web this year. While these ranks indicate their popularity (e.g., site traffic) and reliability (i.e., user trust) as information service providers, and existing studies show that different search engines perform differently, and even the same search engine performs differently based on the type of search query [11, 18], it is essentially reasonable to think that search results from different search engines of different ranks have different levels of acceptance. To determine the acceptance level of each search service provider, we conduct an experiment with 139 programming errors and exceptions[2]. We collect the top 10 search results against the exception query from each search tool and get their *Alexa ranks* [1]. Then, we consider the *Alexa ranks* of all result links provided by each search tool and calculate the average rank for a result link provided by them. The average rank for each search tool is then normalized and inversed which provides a value between zero and one, and we consider this value as a heuristic measure of confidence for the search tool. We use Equations (3) and (4) to get the search engine confidence for a result link.

$$R_{i,normal} = \frac{\bar{R}_i}{\sum_{i=1}^{n} \bar{R}_i} \tag{3}$$

$$S_{i,sec} = \frac{\frac{1}{R_{i,normal}}}{\sum_{i=1}^{n} \frac{1}{R_{i,normal}}} \tag{4}$$

Here, $\bar{R}_i$ represents the average *Alexa rank* for each search tool results, $R_{i,normal}$ is the normalized version of $\bar{R}_i$ and $S_{i,sec}$ refers to the final confidence score for each search tool based on *Alexa search traffic* statistics. We get a normalized confidence of 0.29 for Google, 0.35 for Bing, 0.36 for Yahoo and 1.00 for StackOverflow given that StackOverflow is a popular programming Q & A site, which has drawn the attention of a vast programming community (1.9 million[3]) and

[1]http://www.alexa.com/topsites, Visited on September, 2013
[2]http://homepage.usask.ca/∼mor543/query.txt
[3]http://en.wikipedia.org/wiki/Stackoverflow, Visited on September, 2013

contains about 12 million questions and answers. The idea is that the occurrence of a result link in multiple search engines against a single query provides the associated confidence scores from the search engines to the link. Thus, if a result occurs in all search provider results, it gets a confidence score of 2.00; however, finally, the scores of all results in the corpus are normalized for practical use.

2) Content Matching Score (S_{cms}): During errors or exceptions, the IDE or Java framework generally issues notifications from a fixed set of error or exception messages unless the developer handles the exception manually. Therefore, there is a great chance that a search result page titled with an error or exception message similar to the search query would discuss about the encountered problem by the developer, and would contain relevant and useful information for fixation. We propose a metric, *Title to Title Similarity* (S_{tts}), that measures the content similarity between the query message and the title of each result page. We use *cosine similarity measure* for this purpose, and it returns a value between zero (i.e., completely dissimilar) and one (i.e., exactly similar). As we noted, the result title may not always provide enough information about the content of the exception and therefore, page content needs to be consulted. We consider the stack traces, source code snippets and discussion text extracted from the page content as the legitimate sources of information about the discussed exceptions, and propose two cosine similarity-based metrics– *Title to Context Similarity* (S_{tcx}) and *Title to Description Similarity* (S_{tds}). *Title to Context Similarity* score determines the content relevance between query error message and the extracted exception context (e.g., stack traces, associated code snippets), and *Title to Description Similarity* score denotes the possibility of the occurrence of query error message in the discussion text.

According to Arif et al. [5], terms contained in different parts of the document deserve different levels of attention. For example, a phrase in the page title is more important than a phrase in discussion text to specify the subject matter of the document. We reflect this idea in content matching, and assign different weights to different content similarity scores. We use Equation (5) to determine the content relevance between the query exception and a result document.

$$S_{cms} = \alpha \times S_{tts} + \beta \times S_{tcx} + \gamma \times S_{tds} \quad (5)$$

Here, α, β and γ are the assigned weights to result page title, extracted context (stack trace and code snippet) and discussion text respectively, and they sum to one. Given that the similarity scores are generated from cosine-based measures, *Content Matching Score* always ranges from zero (i.e., completely irrelevant) to one (i.e., completely relevant).

3) Stack Trace Matching Score (S_{stm}): To solve the programming errors or exceptions, the associated context such as stack trace generated by the IDE plays an important role. Stack trace contains the error or exception type, system messages and method call references in different source files. In this research, we consider an incentive to the result links containing stack traces similar to that of the query error or exception. We consider both the lexical and structural perspectives of a stack trace, and propose two metrics– *Lexical Similarity Score* (S_{lex}) and *Structural Similarity Score* (S_{stc}), to determine the relevance between stack traces. Stack trace generally contains two types of information– a detailed technical message containing exception name(s) and likely cause(s) of exception, and a list of method call references. We parse the exception names, error message from the first part and extract package, class and method names from each reference in the second part to develop a token set for the stack trace. We use this token set to determine the lexical relevance between two stack traces, and we use *Cosine Similarity Score* for the purpose. It should be mentioned that we do not decompose the camel case tokens into granular levels (i.e., granularization introduces false positives) in order to perform meaningful similarity checking, which makes the relevance checking effective and useful.

Method call references and their sequence in the stack trace provide important clues to identify the target exception locations and thus, they can help to determine the relevance between two stack traces. We calculate the *Degree of Interest Score* for each reference in the query stack trace using Equation (2) and use them to determine *structural relevance* with the candidate stack traces extracted from the result pages. The idea is to determine the occurrence of query method call references in the candidate stack traces. However, complete matching between two references may not be likely and we exploit the idea of *confidence coefficient* proposed by Cordeiro et al. [9]. We get the *Structural Similarity Score* between query stack trace and a candidate stack trace using Equations (6) (proposed by Cordeiro et al. [9]) and Equation (7).

$$ms_i = S_{doi} \times c_i \quad (6)$$

$$S_{stc} = \frac{1}{N} \sum_{i=0}^{N} ms_i \quad (7)$$

Here, ms_i denotes the matching score between two references, c_i refers to confidence coefficient and N represents the number of method call references in the query stack trace. Once both lexical and structural relevance are found, we get the *Stack Trace Matching Score* using the following equation.

$$S_{st} = \delta \times S_{stc} + \sigma \times S_{lex} \quad (8)$$

Here, δ and σ denote two heuristic weights assigned to *structural similarity* and *lexical similarity* scores respectively, and they sum to one. *Stack Trace Matching Score* values from zero and one, where zero represents total irrelevance and one represents the complete relevance between two stack traces.

4) Source Code Context Matching Score (S_{ccx}): Sometimes, stack trace alone may not provide enough information for the diagnosis of the occurred exception, and the associated code generating the exception needs to be consulted. In programming Q & A sites, forums and discussion boards, users often post source code snippets related to the exception besides the stack traces for clarification. We are interested to check if the code snippets in the result link are similar

to the source code associated with the query exception. This coincidence is possible with the notion that the developers often reuse code snippets from different programming Q & A sites, forums or discussion boards in their programs directly or with minor modifications. Therefore, a result link containing code snippets similar to the code associated with the query exception is likely to discuss relevant issues that a developer needs to know in order to handle the exception. We consider the code surrounding the exception location in the source file as the source code context of the target exception, and use a state of the art code clone detection technique, NiCad [16] to determine its relevance with the code snippets extracted from the result pages. The idea is to identify the *longest common subsequence of tokens* between two token sets extracted from two different code snippets. We use Equation (9) to determine the relevance between the context code associated with query exception and the code snippets extracted from the result page.

$$S_{ccx} = \frac{|S_{lcs}|}{|S_{total}|} \quad (9)$$

Here, S_{lcs} denotes the longest common subsequence of tokens, and S_{total} denotes the set of tokens extracted from the code blocks considered as the context of the occurred exception. The *Source Code Context Matching Score* values from zero to one.

5) StackOverflow Vote Score (S_{so}): StackOverflow, a popular programming Q & A site with 1.9 million users, maintains a score for each question, answer and comment posted by the users, and the score can be considered as a social and technical recognition of their merit [12]. In StackOverflow, a user can up-vote any question or answer post if she likes something about them, and can also down-vote if the post content seems erroneous, confusing or not helpful. Thus, the difference between up-votes and down-votes from a vast community of technical users, the score of post, is considered as an important metric for evaluation of the quality of the solution posted. In our research, we consider these scores of answer posts against an asked question in the result link from StackOverflow, and calculate *StackOverflow Vote Count* using Equation (10). Then we normalize the vote count and get *StackOverflow Vote Scoe* using Equation (11).

$$SO_k = \sum_{\forall p \epsilon P} V_p \quad (10)$$

$$S_{so} = \frac{SO_k - \lambda}{max(SO_k) - \lambda} \quad (11)$$

Here, SO_k refers to the StackOverflow vote count for a result page, V_p denotes the vote count for a post in the page, p refers to any post, and P denotes the set of question and answer posts found in a result page. λ denotes the minimum vote count, $max(SO_k)$ represents the maximum vote count and S_{so} is the normalized *StackOverflow Vote Score* for the result link. The score values from zero (i.e., least significant) to one (i.e., most significant), and it indicates the relative quality or popularity of the StackOverflow links in the eyes of a large crowd of technical users.

6) Search Traffic Rank Score (S_{str}): The amount of search traffic to a site can be considered as an important indicator of its popularity. In this research, we consider the relative popularity of the result links extracted from different search engines. We use the statistical data from two popular site traffic control organizations– *Alexa* and *compete* through their provided APIs and get the average popularity rank for each result link. Then, based on these ranks, we provide a normalized *Search Traffic Rank Score* to each result link between zero and one considering minimum and maximum ranks found.

C. Result Scores Calculation

The proposed metrics (Section III-B) focus on four aspects of each result– content relevance, context relevance, popularity and search engine confidence, and we consider those aspects for the calculation of final scores. We use *Content Matching Score* of each result page as its *Content Relevance,* (R_{cnt}) with the encountered exception. In this research, we consider stack trace and the code block triggering the exception as the context of the occurred exception, and use *Stack Trace Matching Score* and *Source Code Context Matching Score* to determine the *Context Relevance, (R_{cxt})* of each result page. Both stack trace and context code carry different levels of significance and we assign two heuristic weights to the matching scores to get the context relevance score.

$$R_{cxt} = w_{st} \times S_{stm} + w_{cc} \times S_{ccx} \quad (12)$$

Here, w_{st} and w_{cc} are the assigned weights to S_{stm} and S_{ccx} respectively, and they sum to one which gives R_{cxt} normalized, value from zero to one.

StackOverflow Vote Score and *Search Traffic Rank Score* are considered as the measures of popularity of each result link from different viewpoints, and they deserve different levels of attention. In the calculation of *Popularity Score*(S_{pop}) of each result link, we assign two different heuristic weights to these metrics.

$$S_{pop} = w_{so} \times S_{so} + w_{sr} \times S_{str} \quad (13)$$

Here, w_{so} and w_{sr} represent the assigned weights to S_{so} and S_{str} respectively, and they sum to one; this gives S_{pop} a normalized value from zero (i.e., the least popular) to one (i.e., the most popular).

Confidence of each result obtained from the associated search engines can be considered as a support measure for the result link against a search query. We consider *Search Engine Confidence Score* as the confidence of each result. Thus, the four component scores associated with four aspects can be combined using Equation (14) to get the final score for each result.

$$S_{final} = \sum_{\exists w \epsilon W, \exists RS \epsilon (R_{cnt}, R_{cxt}, S_{pop}, S_{sec})} w \times RS \quad (14)$$

Here, RS denotes a measure of content relevance, context relevance, popularity or confidence of a result in the corpus, and w denotes the individual weight (i.e., importance) associated with each RS. We assign a heuristic weight of

0.35 to content-relevance, 0.85 to context-relevance, 0.20 to popularity and 0.10 to the impression of the result link with the search engines. We choose these heuristic weights based on our extensive and iterative controlled experiments with a subset of all exceptions, manual analysis on the experimental results, discussion among the authors, and also some helpful ideas from the existing study [9]. While these heuristic values might seem a bit arbitrary, we find the combination to be the best in our experiments to represent the relative importance of different aspects of the final score for a result. Once the final scores are found, the results are sorted in a descending order and the top twenty or thirty of them are returned to the requesting client.

IV. Experiment Design, Results and Validation

A. Dataset preparation

We collect the *workspace logs* of Eclipse IDE from six graduate research students of Software Research Lab, University of Saskatchewan, and extract the exceptions (e.g., error message, stack trace) occurred during the last six months. We collect a list of 214 stack traces from them. We find that most of the stack traces are duplicate of one another, and we choose 38 distinct stack traces involving 44 *Exception classes*, which are mostly related to Eclipse plugin development framework. To include exceptions related to standard Java application development and prepare a balanced dataset, we choose 37 exceptions from a list of common Java exceptions [3]. Then, we generate some of those exceptions using code examples, and we also perform exhaustive web search to collect the stack traces and source code context associated with those exceptions. We finally get a list of 75 exceptions associated with 75 stack traces and 37 contextual code blocks [3], which we use as the dataset for different experiments. It should be noted that we cannot collect helpful context codes for the exceptions extracted from the *workspace logs* of the IDE. We collect the most appropriate solutions for the exceptions with the help of different available search engines such as Google, Bing and Yahoo. Given that checking relevance of a solution is controlled by different subjective factors, we select the solution list carefully. First, one of the authors performs exhaustive web search for two days and collects a potential list of solutions for the exceptions which are shared with other authors. The other authors review the exception information (e.g., stack trace, code context) and the solution list independently, and provide their feedback about the selection of the solution list. Then, the suggestions of all authors are accumulated to finalize a solution set [3] for the exceptions in the dataset.

B. Investigation with Eclipse IDE

We analyze the features provided by Eclipse IDE in order to find out how often the developers can get necessary help in fixing the encountered exceptions. It is interesting to note that the IDE provides nice debugging tools for them to analyze the exceptions through check-pointing, but they do not help much in platform-level exceptions associated with different runtime libraries and configuration files. For example, if a Java application tries to consume more memory space than alloted, Java framework will issue this exception message– *java.lang.OutOfMemoryError: Java heap space* and the debugging tools have very little opportunity to suggest about the fixation. There is also an internal web browser widget in recent versions of Eclipse IDE; however, it is intended for browsing web pages and is defaulted to *Bing Search* which does not consider the context of the encountered exceptions, and thus cannot help the developers much effectively.

C. Search Query Formulation

Every web search request from Eclipse client plugin has three components– query for search engines, stack trace and code context. Our proposed solution works in two modes– interactive and proactive. In case of interactive mode, the developer forms the search query by carefully selecting keywords from the exception message and context information, whereas the plugin is responsible for preparing the search query itself in case of proactive mode. This section discusses the query formulation techniques for proactive mode.

In this research, we consider both stack trace and the code block likely responsible for the exception as the context for the exception. Therefore, we collect information from the context besides the exception message to generate the search query for the exception. Traditional search engines generally do not allow big queries (i.e., stack traces) [2] or perform poorly against them [11, 18], and in order to overcome that challenge and collect search results, we use a sophisticated technique to describe the exception and its context in terms of few tokens. We capture the exception message containing exception class name returned by the IDE and collect five class and method names with the highest *Degree of Interests* from the stack trace [9]. We also extract five most frequent method calls and imported class names from the context code using an *ASTParser library*[4] (i.e., in case of compilable code) and a custom island parser (i.e., in case of uncompilable code) [13]. Then, we combine the extracted method and class names from both context to generate a unique list of query terms and add the list to the exception message. We note that the exception message itself returned by the IDE is a good descriptor of the exception; however, we filter the message and discard the irrelevant components such as absolute file path, URL and so on. Thus, the filtered exception message and the list describing the context of the exception develop the search query which we use to collect results from different search engines in proactive mode.

D. Performance Metrics

Given that our proposed approach is aligned with the research areas of information retrieval and recommendation systems, we use a list of performance metrics from those areas as follows.

1) Mean Precision (MP): While *precision* denotes the fraction of retrieved results that are relevant to the query, *Mean Precision* is the average of that measure for all queries.

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

2) Mean First False-Positive Position (MFFP): First false-positive position (FFP) is the rank of first false-positive result in the ranked list. *Mean First False-Positive Position* measures the average First false-positive position for each query in the query set.

3) Mean Reciprocal Rank (MRR): Reciprocal Rank is the multiplicative inverse of the rank of first relevant result. *Mean Reciprocal Rank* is a statistical measure that averages the *Reciprocal Rank* for each query in the query set.

4) Recall (R): Recall denotes the fraction of the relevant results that are retrieved. In our experiments, we consider *recall* as the percentage of the exceptions for which the solutions are recommended correctly.

E. Experimental Results on Proposed Approach

In our experiments, we conduct search with each exception in the dataset and collect the top 30 results. We consider both working modes– interactive and proactive, and analyze the results using different performance metrics (Section IV-D). Tables I, II and III show the results of the experiments conducted on our approach.

Table I shows a comparative analysis between *interactive* mode and *proactive* mode of search by the proposed approach. Here, we see that our tool performs relatively better in interactive mode than proactive mode in terms of different performance metrics such as Mean Precision (MP), Mean First False-Positive Position (MFFP), Mean Reciprocal Rank (MRR) and Recall (R). We also note that *proactive version* can recommend correct solutions for 56 exceptions in total whereas *interactive version* can recommend for 68 out of 75 exceptions, which gives a recommendation accuracy of 90.66% for our approach. Given that formulating search query is one of the decisive factors for the performance of *interactive approach*, we manually select a list of search-friendly keywords from the context of each exception as the search query. The query collects a richer set of initial results from multiple search engines than that of *proactive approach*, where the search keywords are not fine-tuned for search engines.

Table II investigates the impacts of different aspects– content relevance, context relevance, popularity and search engine confidence of the result link in the scoring of each result. It shows how the incremental association of different aspects can improve the search results in terms of performance metrics such as Mean Precision (MP), Recall (R) and so on. We note that the proposed approach provides the highest MP and the highest recall when all four aspects are considered during scoring rather than a single aspect such as *content-relevance*. For example, it performs the best (e.g., 90.66%) in terms of *accuracy* (e.g., TEF (No. of total exceptions fixed), R (% of exceptions fixed)) when all four dimensions of the result score are considered, which shows the potential of exploiting associated context information besides search query during search.

Table III compares the experimental results achieved against two different sets of exceptions– one with exception messages and stack traces (e.g., Set A) and the other with exception messages, stack traces and code context (e.g., Set B). Here, we see that Set B, that considers code context besides stack trace and error message of an exception, achieves higher accuracies (e.g., 97.29% and 86.48%) than Set A (e.g., 84.21% and 63.16%) in both working modes. It also gets better results in terms of other performance metrics such as Mean Precision (MP) and Recall (R). Given the findings, all indicate that a combination of context code and stack trace can better specify the context of the exception rather than stack trace only, and thus, by exploiting the context, the proposed approach can recommend solutions more for the set that captures the combination than the one that do not capture.

TABLE I
RESULTS OF EXPERIMENTS ON PROPOSED APPROACH

Mode	Metric	Top 10[1]	Top 20[2]	Top 30[3]
Interactive	Mean Precision (MP)	0.1229	0.0736	0.0538
	MFFP	1.2400	1.2400	1.2400
	MRR	0.4604	0.4648	0.4669
	TEF[4]	59(75)	64(75)	**68**(75)
	Recall (R)[5]	78.66%	**85.33%**	**90.66%**
Proactive	Mean Precision (MP)	0.0866	0.0529	0.0380
	MFFP	1.2400	1.2400	1.2400
	MRR	0.4009	0.4048	0.4054
	TEF	51(75)	55(75)	56(75)
	Recall (R)	68.00%	73.33%	**74.66%**

[1] Metrics for the top 10 results
[2] Metrics for the top 20 results
[3] Metrics for the top 30 results
[4] No. of exceptions fixed
[5] Percentage of exceptions fixed

TABLE III
EXPERIMENTAL RESULTS ON MULTIPLE SETS

Mode	Metric	Set A[1](38)	Set B[2](37)
Interactive	Mean Precision (MP)	0.0404	0.0604
	MFFP	1.0000	1.4864
	MRR	0.2695	0.6697
	TEF	32	36
	Recall (R)	**84.21%**	**97.29%**
Proactive	Mean Precision (MP)	0.0263	0.0450
	MFFP	1.0000	1.4864
	MRR	0.2563	0.5585
	TEF	24	32
	Recall (R)	**63.16%**	**86.48%**

[1] Contains exception message and stack trace.
[2] Contains exception message, stack trace and code context.

F. Experiments with Existing Approaches

We compare the results of our proposed approach against two existing IDE-based recommendation systems– context-based recommendation system by Cordeiro et al. [9] and *Seahawk* by Ponzanelli et al. [13]. Both of them collect data from StackOverflow data dump and recommend StackOverflow posts taking the current context of the search into consideration. They select suitable tokens from either stack trace or context code to describe the problem context in the search query, and recommend solutions in a proactive fashion. We implement both of the existing methods and use them for experiments.

To implement the approach proposed by Cordeiro et al. [9], we extract the exception name from each exception message and collect 100 top voted StackOverflow posts discussing about that exception, and develop a corpus for the exception test case. We download the page source of each item in the corpus and create an *Apache Lucene Index* for the corpus. Then, we use the index and *Lucene search engine* to retrieve the relevant posts against the search query associated with the

TABLE II
EXPERIMENTAL RESULTS FOR DIFFERENT SCORE COMPONENTS

Score Combination	Metric	Interactive			Proactive		
		Top 10	Top 20	Top 30	Top 10	Top 20	Top 30
Content (R_{cnt})	MP	0.0899	0.0607	0.0481	0.0871	0.0528	0.0371
	TEF	43	55	65	46	54	56
	Recall (R)	57.33%	73.33%	86.66%	61.33%	72.00%	74.66%
Content (R_{cnt}) and Context (R_{cxt})	MP	0.11428	0.0699	0.0514	0.0785	0.0499	0.0376
	TEF	58	63	66	45	52	55
	Recall (R)	77.33%	84.00%	**88.00%**	60.00%	69.33%	73.33%
Content (R_{cnt}), Context (R_{cxt}), and Link Popularity (S_{pop})	MP	0.1157	0.0699	0.0519	0.0857	0.0542	0.0381
	TEF	57	63	66	49	55	56
	Recall (R)	76.00%	84.00%	**88.00%**	65.33%	73.33%	74.66%
Content (R_{cnt}), Context (R_{cxt}), Link Popularity (S_{pop}) and Result confidence (S_{sec})	MP	**0.1229**	0.0736	0.0538	0.0886	0.0529	0.0380
	TEF	59	64	**68**	51	55	56
	Recall (R)	78.66%	**85.33%**	**90.66%**	68.00%	73.33%	**74.66%**

TABLE IV
COMMON AND UNIQUE RESULTS FROM SEARCH ENGINES

Search Query	Common	Google Only	Bing Only	Yahoo Only
Content Only	32	09	16	18
Content and Context	47	09	11	10

test case. From *Lucene*, we also collect the *retrieval score* based on *Vector Space Model* for each retrieved post. We calculate the *structural score* and *lexical score* of each post considering their stack traces and then normalize them. Finally, we add all three scores for each post to get the final score.

In case of *Seahawk* proposed by Ponzanelli et al. [13], we collect the ten most frequent tokens (e.g., method reference, class name) from the context code as the search query, and retrieve relevant posts from StackOverflow containing suitable code examples or discussions helpful for current state of coding. Given that *Apache Solr* is a search service provider using *Apache Lucene* as the core search engine, we use *Apache Lucene* to collect relevant results against a search query. Basically, we reuse the previously developed indexes of StackOverflow posts and collect the relevant posts as well as their relevance scores.

Table V (left part) shows a comparative analysis between the results of two existing approaches– Cordeiro et al. [9] and Ponzanelli et al. [13], and our proposed approach. The working principles of the existing approaches are similar to that of *proactive* version of our tool, and therefore, we compare them to the *proactive* version. Here, we can see that both of the existing approaches perform poorly in terms of all performance metrics compared to our approach. In the best case, they can recommend solutions for 24.00% and 18.92% of the exceptions respectively. Given the findings, it is evident that depending on a single information source for exception is not a good choice, and the combination of stack trace and code context is a preferable choice to either any of them for reflecting the exception context during search.

G. Experiments with Existing Search Engines

We compare the results of our proposed approach against three available search engines– Google, Bing and Yahoo, and one Q & A website– StackOverflow. The interactive mode of our approach allows the developer to provide a search query which resembles with working principles of the search engines. We develop search query for each exception using suitable tokens from the context, and use them to collect results from the search engines as well as from the proposed approach. It should be mentioned that the search query is simply used to develop the corpus in case of the proposed approach, and final ranking of the results are determined with the help of ranking algorithms using the automatically extracted context information from IDE. We collect the top 30 results from each search provider and look for expected solutions identified previously (Section IV-A).

Table V (right part) shows the comparative analysis of results from different search engines and our proposed approach. Here, we see that existing search engines can recommend solutions for at most 58 (77.33%) out of 75 exceptions, where the proposed approach can recommend for 68 (90.66%) exceptions. We also note that *Google* performs slightly better than our approach in terms of Mean Precision (MP), but recommends correct solutions for only 57 exceptions. Given that *selection of appropriate query terms* is an essential precondition for successful search, we conduct another experiment with those search engines using two scenarios–keywords from only exception message, and keywords both from exception message and exception context. Table IV shows the results of those two scenarios. Here, we see that the keyword-based query that considers the exception context, provides more relevant results than the one that does not consider. Therefore, the performance of the traditional search engines is subjected to the selection of search keywords, and the appropriateness of this selection entirely depends on the developer's skill. In our case, we choose the search keywords carefully which provides the better results (e.g., precision) for Google, but it cannot be taken for granted given the uncertainty in query selection. On the other hand, it is interesting to note that our approach, for the same set of queries, can recommend correct solutions for more exceptions with a little compromise in the precision, and the developers get rid of the burden of choosing appropriate tokens for the context. They can select an encountered exception for search, and the plugin itself captures the problem context to recommend relevant results, whereas Google depends entirely on the developers for the context-based information.

Given the precise results from Google search engine, and the correlation between Mean Precision (MP) and Recommendation Accuracy (e.g., R) observed at Table II, one may

TABLE V
RESULTS OF EXPERIMENTS ON EXISTING APPROACHES AND SEARCH ENGINES

Proactive Mode

Recommender	#TE[1]	Metric	Top 10	Top 20	Top 30
Cordeiro et al. [9]	75	MP	0.0202	0.0128	0.0085
		TEF[2]	15	18	18
		R	20.00%	24.00%	**24.00%**
Proposed Approach	75	MP	0.0886	0.0529	0.0380
		TEF	51	55	**56**
		R	68.00%	73.33%	**74.66%**
Ponzanelli et al. [13]	37	MP	0.0243	0.0135	0.0099
		TEF	7	7	7
		R	18.92%	18.92%	**18.92%**
Proposed Approach	37	MP	0.1000	0.0621	0.0450
		TEF	30	**32**	**32**
		R	81.08%	86.48%	**86.48%**

Interactive Mode

Search Engine	#TE	Metric	Top 10	Top 20	Top 30
Google	75	MP	**0.1571**	0.0864	0.0580
		TEF	57	57	57
		R	76.00%	76.00%	**76.00%**
Bing	75	MP	0.1013	.0533	0.0364
		TEF	55	58	58
		R	73.33%	77.33%	**77.33%**
Yahoo	75	MP	0.0986	0.0539	0.0369
		TEF	54	57	57
		R	72.00%	76.00%	**76.00%**
StackOverflow	75	MP	0.0226	0.0140	0.0097
		TEF	14	17	17
		R	18.66%	22.66%	**22.66%**
Proposed Approach	75	MP	**0.1229**	0.0736	0.0538
		TEF	59	64	**68**
		R	**78.66%**	**85.33%**	**90.66%**

[1] No. of exceptions used for the experiment.
[2] No. of total exceptions fixed.

argue that only Google results should be considered for corpus development in the proposed approach. In our research, we investigate whether such corpus is likely to contain the correct solutions for more exceptions or not. We develop corpus for each of 75 exceptions collecting the top 100 results from Google search API against the selected exception, and apply the proposed ranking algorithms. From Table V (right part) we find that the Google corpus-based approach can recommend correctly for at most 57 exceptions. Moreover, Table IV shows that each search provider contains some unique recommendations which cannot be exploited if we consider only one search engine. Therefore, the idea of accumulating search results from multiple search engines for corpus development is promising, and it ensures the maximum *Recall (R)* for our approach by leveraging the existing search services.

We also investigate into why the proposed approach provides slightly less precise results compared to Google. Given that our approach involves scraping of semi-structured data from the result web page, it may sometimes fail to extract the exception context information properly if the page does not contain the information in the expected tags (e.g., *code, pre, blockquote*). From our manual analysis with ten cases having the most precise and the least precise recommendations, we find that recommended pages containing context information (e.g., stack trace, context code snippets) relevant to the query exception are likely to rank higher than those which do not contain such information. In case of the least precise results, the recommended pages contain that information either in an unstructured way which is difficult to extract or they do not contain it at all. Given that the proposed approach emphasizes on the context of the problem discussed in a result page, it does not perform well for those test cases. Therefore, improvement of the context information extraction techniques from web page can help to enhance the *precision* of the proposed approach, which we consider as a scope for future study.

V. THREATS TO VALIDITY

During the research, we identify a few threats to validity which we discuss in this section. First, the proposed approach still does not provide the search results in real time. Given that the approach involves into scraping web page content for context information about the discussed problem, it takes 30-35 seconds on average to return the recommendations. We applied Java based multi-threading to speed up the computation; however, the approach can be made returning results in real time by more extensive parallelization on the web server and we have already designed it for multi-core systems.

During the experiments, we note that the existing search engines evolve rapidly, especially Google, within days and weeks, and the recommendations from the search engine vary over time for the same query. Therefore, the statistics from the experiments with the search engines are very likely to change. Given that our approach exploits the *live API services* from them, it would also evolve, and it is also subjected to the strengths and weaknesses of the search engines. However, adoption of meta search based approach is likely to aggregate the strengths and mitigate the weaknesses of each individual search engine as we showed the effectiveness.

Most of the programming errors and exceptions we selected for the experiments are frequently encountered by the developers, and their solutions are also widely discussed in the web. One may argue whether the wide availability of those solutions contributes to the better performance of the proposed approach or not. Our approach does not differentiate between frequent and rare programming exceptions, and it returns the relevant recommendations as long as sufficient data are collected from the search engines. However, the approach is subjected to the availability of the appropriate context information (e.g., stack traces, context code) in the web page for relevance checking.

VI. RELATED WORKS

Existing studies related to our research focus on integrating commercial-off-the-shelf (COTS) tools into Eclipse IDE [14], recommending StackOverflow posts and displaying them within IDE environment [9, 13], recommending previously visited web pages [17] and open source code [6], embedding traditional web browser inside the IDE [8] and so on. Poshyvanyk et al. [14] integrate Google Desktop Search

API into Eclipse environment to facilitate customized search for information within the IDE, which can be leveraged by different software maintenance activities. Cordeiro et al. [9] propose an IDE based recommendation system for runtime exceptions. They extract the question and answer posts from StackOverflow data dump, and suggest posts proactively relevant to the occurred exceptions considering the stack trace information generated by the IDE. In contrast to StackOverflow data dump, our research exploits the existing web search and StackOverflow API services to collect filtered and relevant data from multiple sources, and consider context code, popularity and search engine confidence of the result link besides the exception stack trace. Ponzanelli et al. [13] also propose an Eclipse IDE based recommendation system, *Seahawk*, that considers the current state of coding and recommends relevant StackOverflow posts containing code examples and discussions helpful to the coding. However, it does not consider the stack trace as a component of problem context, and its recommendation may not work for programming exception related issues. Brandt et al. [8] embed a custom code search engine, *Blueprint*, in Eclipse IDE and conduct a user study in the laboratory environment to investigate whether IDE-based browser can help developer productivity compared to stand-alone web browser. They conclude that the tool helped the developers significantly to write better code and find code examples, and task-specific search interface can greatly influence the web search usage. Our research is related to it in the sense that we also try to address the context-switching issues through IDE based web search features and suitable user interfaces. Sawadsky et al. [17] propose *Reverb*, a tool that considers the code under active development within the IDE, and proactively recommends previously visited and relevant web pages from the browsing history. Bajracharya et al. [6] propse *Sourcerer*, an open source code search engine that considers both *TF-IDF* and the structural relationships among the code elements to recommend Java classes from 1500 open source projects. Both Sawadsky et al. and Bajracharya et al. exploit lexical and structural features of the source code for recommendation. In our research, we apply similar set of features of the code with the focus on matching local code context of an encountered exception in the IDE against that of the exceptions and programming problems discussed in the web pages for relevant recommendation within the IDE.

VII. Conclusion and Future Works

To summarize, we propose a novel IDE-based web search solution that exploits three reliable web search engines and a programming Q & A site through their API endpoints. The approach facilitates to search for helpful information in the web from within the IDE when the developers face different programming errors and exceptions, and it considers both the problem content and problem context during the search. It also considers the popularity and the impression of each result link to different search engines during ranking of the results. We conduct experiments on our approach with 75 programming errors and exceptions related to Eclipse plug-in development framework and standard Java applications. We also conduct experiments on the existing approaches by Cordeiro et al. [9] and Ponzanelli et al. [13], three traditional search engines and StackOverflow with the same dataset. Experiments show that our approach outperforms the existing approaches, search engines and StackOverflow search feature in terms of *recall* and other performance metrics. Experiments also show that inclusion of all types of context information during search can speed up the accuracy of a recommendation system in the IDE. In future, we would attempt to increase the precision of our recommendation system, and validate its applicability with a comprehensive user study.

References

[1] Alexa Page Rank API. URL http://data.alexa.com/data?cli=10&url=domain.name.

[2] Web Search Query. URL http://en.wikipedia.org/wiki/Web_search_query.

[3] SurfClipse Experiment Data. URL http://homepage.usask.ca/~mor543/sc/info.

[4] SurfClipse Web Service. URL https://srlabg53-2.usask.ca/wssurfclipse.

[5] A. Arif, M.M. Rahman, and S.Y. Mukta. Information Retrieval by Modified Term Weighting Method Using Random Walk Model with Query Term Position Ranking. In *Proc. ICSPS*, pages 526–530, 2009.

[6] S. Bajracharya, T. Ngo, E. Linstead, P. Rigo, Y. Dou, P. Baldi, and C. Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Proc. OOPSLA*, pages 25–26, 2006.

[7] 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.

[8] 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.

[9] J. Cordeiro, B. Antunes, and P. Gomes. Context-based Recommendation to Support Problem Solving in Software Development. In *Proc. RSSE*, pages 85 –89, June 2012.

[10] Max Goldman and Robert C. Miller. Codetrail: Connecting Source Code and Web Resources. *J. Vis. Lang. Comput.*, 20(4):223–235, August 2009.

[11] B.T. S. Kumar and J.N. Prakash. Precision and Relative Recall of Search Engines: A Comparative Study of Google and Yahoo. *J. Lib. and Info. Mgmt.*, 38(1):124–137, 2009.

[12] S.M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What Makes a Good Code Example?: A Study of Programming Q & A in StackOverflow. In *Proc. ICSM*, pages 25–34, 2012.

[13] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.

[14] D. Poshyvanyk, M. Petrenko, and A. Marcus. Integrating COTS Search Engines into Eclipse: Google Desktop Case Study. In *Proc. IWICSS*, pages 6–, 2007.

[15] M.M. Rahman, S. Yeasmin, and C.K. Roy. An IDE-Based Context-Aware Meta Search Engine. In *Proc. WCRE*, pages 467–471, 2013.

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

[17] N. Sawadsky, G.C. Murphy, and R. Jiresal. Reverb: Recommending code-related web pages. In *Proc. ICSE*, pages 812–821, 2013.

[18] T. Usmani, D. Pant, and A. K. Bhatt. A Comparative Study of Google and Bing Search Engines in Context of Precision and Relative Recall parameter. *J. CSE*, 4(1):21–34, 2012.