Chapter 14
Novel and Applied Algorithms in a Search Engine for Java Code Snippets

Phitchayaphong Tantikul, C. Albert Thompson, Rosalva E. Gallardo-Valencia, and Susan Elliott Sim

Abstract Programmers often look for a "snippet," that is, a small piece of example code, to remind themselves of how to solve a problem or to quickly learn about a new resource. However, existing tools such as general-purpose search engines and code-specific search engines do not deal well with searches for snippets. In this chapter, we present a prototype search engine designed to work with code snippets. Our approach is based on using the non-code text on a web page as metadata for the snippet to improve indexing and retrieval. We discuss some implementation issues that we encountered, which lead to lessons learned for others who follow. These issues include: extracting snippets from web pages, selecting and indexing metadata, matching query terms with multiple metadata indexes, and identifying a text summary to be used in the presentations of results.

P. Tantikul (✉)
University of California, Irvine, Irvine, CA, USA
e-mail: ptantiku@gmail.com

C.A. Thompson
University of British Columbia, Vancouver, BC, Canada
e-mail: leetcat@cs.ubc.ca

R.E. Gallardo-Valencia
Intel Corporation, Santa Clara, CA, USA
e-mail: rgallardovalencia@acm.org

S.E. Sim
Many Roads Studios, Toronto, ON, Canada
e-mail: ses@drsusansim.org

S.E. Sim and R.E. Gallardo-Valencia (eds.), *Finding Source Code on the Web for Remix and Reuse*, DOI 10.1007/978-1-4614-6596-6_14,

14.1 Introduction

Searching for source code on the web has become an integral part of software development. We find evidence of this in the creation of search engines specifically designed to search for source code, such as Strathcona [5], Mica [12], Krugle,[1] Koders,[2] Google Code Search,[3] and Sourcerer [6]. All of these tools take the approach of gathering together as much source code as possible from open source hosting sites and making the repository searchable. Unfortunately, these repositories omit the large number of code snippets that are embedded in web pages throughout the Internet. Snippets usually consist of a handful of lines of code and do not necessarily compile. Since snippets differ from components in a number of ways, it stands to reason that they require a different kind of repository and search engine.

In this chapter, we describe "Juicy," a search engine for snippets of Java code and the lessons learned from its implementation. In the design of Juicy, we treated code snippets as first class objects. When the search engine returns a page of results, the items consist of an excerpt of the code snippet, a link the originating web page, and a brief text description. In implementing Juicy, we used many existing tools and algorithms. Our contribution is in the novel application of these resources and the resulting assemblage.

We used the **Rotation Forest** machine learning algorithm, as implemented by **Weka 3** to help us label sections of web pages from Java tutorial sites as either text or source code. The open source project, **Lucene**, was used as the repository for Juicy. We used the **Porter Stemming** algorithm to normalize words. The **Eclipse AST parser** was used to parse the code snippet. Finally, **Latent Dirichelet Allocation** was used to find the most relevant paragraph of text to be used as a short summary of the snippet.

In addition to leveraging these existing algorithms, we performed some small empirical investigations to inform our design decisions. We identified appropriate features to be used in classifying segments of tutorial pages. We found that it was necessary to filter out many duplicate pages and pages that did not contain code from our initial crawl of Java language tutorial sites. We found that the best text to use as metadata for a snippet is the text segment that appears above the snippet. We found that the best results for a general search were obtained by using only three indexes: web page title, code snippet, and text segment.

In the remainder of this chapter, we will describe how we used these existing algorithms and the design decisions that we made in doing so.

[1] http://www.krugle.com/.

[2] http://www.koders.com.

[3] http://www.google.com/codesearch.

14.2 Approach

Our approach was based on the following key insights:

1. Programming language tutorial pages contain an distinctive combination of source code snippets and natural language.
2. The natural language on the pages can be used as metadata for the code snippets.
3. Effective searches for snippets need to make use of both the source code and the natural language text.

The architecture of Juicy is divided into two parts, a back end that works offline and a interactive front end. These parts are depicted in Fig. 14.1 below.

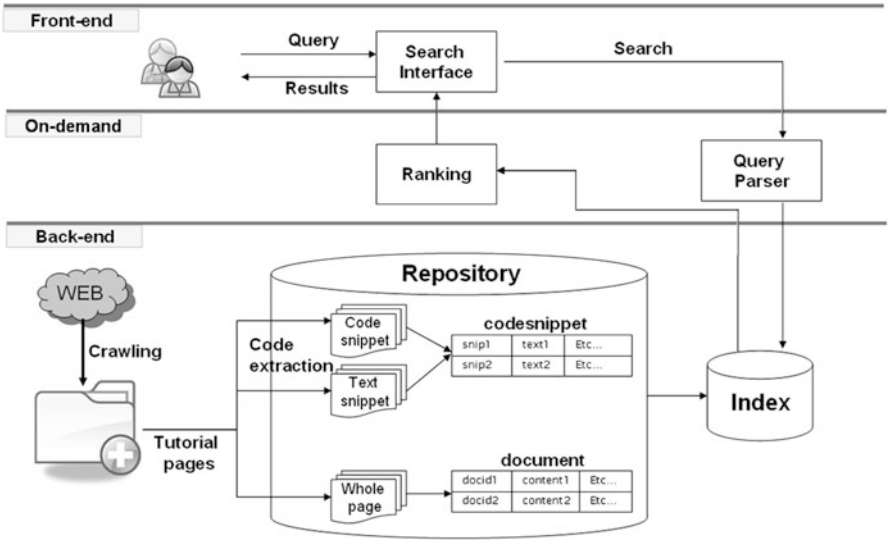


Fig. 14.1 Architecture of Juicy, a Java code snippet search engine

The back end consists of a repository built on top of Apache Lucene, a text search engine library written in Java [8]. Our contributions consist of the techniques for populating the repository with code snippets, and for creating metadata and indexes. The front end provides a user interface to the repository through a web interface.

14.3 Populating the Repository

We populated the repository by using a web crawler to collect web pages from the Internet. In populating the repository for a snippet search engine there are basically three issues that need to be considered: (1) what sites to crawl; (2) how to exclude pages that do not contain Java code; and (3) extracting code snippets from the web page. We will discuss each of these in this section.

14.3.1 Starting Points for the Crawler

The web crawler is a program to collect web-pages from the Internet. It takes an intial web page, or "seed URL" as input and places it in a queue. The program retrieves the page from the first URL in the queue and stores it locally. The contents of this page is further parsed for hyperlinks, which are added to the queue of web sites. The program then iterates through the remaining URLs in the queue. The crawler in this project was built upon HTML Parser.[4]

To construct our prototype repository, we used a set of 33 seed URLs. These were chosen, because they were identified as rich sources of information, and used a variety of page formats. The full list can be found in the appendix for this chapter.

14.3.2 Extracting Snippets

We used a three-step process to separate the source code snippets from the surrounding text on the web page. We used the HTML tags to divide the page into spans, or grouped content segments. These segments were then scored on the presence of features from a natural language (English) and a programming language (Java). These scores were input into a machine learning algorithm to classify either as natural language text or Java source code. The results are exported in an XML file to be read by downstream tools. We will discuss each of these steps further in this section.

14.3.2.1 Division into Grouped Content Segments

Every HTML document consists of a series of spans demarcated by matched pairs of begin/end tags. For instance, the pair of tags indicates a paragraph. The texts in between these tags are called content segments. In other words, content segments are the leaves in the tree representing the document object model.

Unfortunately, content segments can be difficult to classify, because they are short, containing only one or two words, which makes it difficult to label the segment as source code or text. For example, "public" is both an English language word and a Java keyword. For this reason, we took a tactic from lyrics classification and formed grouped content segments from content segments according to their nesting within <pre>and <code>tags. In other words, the content segments in the sub-tree rooted by a <pre>or <code>tag are aggregated.

[4] http://htmlparser.sourceforge.net.

14.3.2.2 Scoring of Natural Language Features

We used the following three features based on characteristics of natural language: number of words, ratio of non-dictionary words, and ratio of stop words. These choices were motivated by previous work on content retrieval. Also, we experimented with other features, such as length of the segment and counts of dictionary words, but these were less fruitful.

Number of Words This feature is simply the count of the number of words in a content segment. Numerals, i.e. digits, are not included. As well, white space of any kind does not affect this feature. Typically, text has a larger number of words per content segment than source code.

Ratio of Non-Dictionary Words Another feature is the number of words in the content segment that does not appear in the Merriam-Webster's 9th Collegiate dictionary [7]. Again, we excluded numerals. Usually, source code has a larger proportion of non-dictionary words than natural language, because identifier names are invented to suit the context and are not limited to dictionary entries.

Ratio of Stop Words In natural language processing, there is the concept of "stop words." These are words that are filtered out prior to processing, because they appear so frequently that they add little information to the input stream. Stop words typically include articles, pronouns, prepositions, and common verbs, such as "to be" [4]. In our work, we do not filter out stop words, but instead use a stop word ratio. We used a published list of English stop words [2], but removed Java programming language keywords. Normally, text would have a higher stop word ratio than source code.

14.3.2.3 Scoring of Programming Language Features

We use the following set of five features derived from programming languages: ratio of keywords, ratio of indentation, number of comments, ratio of separators, and ratio of operators. For each of these, we will discuss how they apply to programming languages in general, and then specifically for Java.

Ratio of Keywords In source code, the words that appear most frequently are programming language keywords. By counting these keywords we can get a good idea if something is source code. Since the number of distinct keywords in a programming language is relatively small, it would not be difficult to adapt this feature to a particular language, such as Java.

Ratio of Indentation In general, indentation is used only to improve readability. Some programming languages, e.g. Fortran and Python, prescribes indentation or assigns a role to a line position. In either case, extensive use of tabs or whitespace is an indicator that a content segment is source code.

Number of Comments Every programming language has syntax for comments. The syntax for comments can vary from language to language, but their presence indicates that a content segment is source code. In some languages, comments are

easy to identify. For example, a letter 'C' or hash mark or single quotation mark in the first position on a line indicates a comment. Java comments are more complex and can take one of two forms. A comment that spans only one line is prefixed by a pair of backslashes '\\'. A comment that spans one or more lines begins with '/*' and ends with '*/'. Care must be taken when working with single-line comments to ensure that double backslashes in URLs do not produce false positives.

Ratio of Separators and Ratio of Operators Programming languages have special punctuation that is used to separate or delimit identifiers or operands. The distinction between separators/delimiters and operators is arbitrary and language specific. We calculate ratios for these two features disjointly.

We relied on the Java Language Specification [3] to define these two classes of special punctuation. For the separators, we included parentheses, curly braces, square brackets, and the semicolon, but did not include the comma and full stop, because the latter two appear frequently in text. For the operators, we used the full Java set, but excluded the hyphen.

14.3.2.4 Classification of the Segments

We used the Rotation Forest algorithm as implemented in Weka 3,[5] an open source framework written in Java, that implements more than 60 different machine learning algorithms.

Rotation Forest [10] is a classifier ensemble method. Its main heuristic is the application of feature classification to subsets of M features using principal component analysis (PCA) separately on each subset and reconstructing a full feature set for each of the L classifiers in the ensemble. Using Weka, we ran the algorithm for subsets of three features (M = 3), 10 classifiers in the ensemble (L = 10), and using a J48 decision tree. The algorithm is named Rotation Forest, because it uses a simple rotation of the coordinate axes from the PCA and the base classifier model is a decision tree.

The scores for the features are input to a Rotation Forest classifier. The output is two values: the likelihood that the grouped content segment is text and the likelihood that it is source code. We take the higher of the two values and apply the appropriate label to the segment.

14.3.2.5 Evaluating the Algorithm

We created a corpus of web pages that contained Java source code examples. We compared the output generated by each algorithm for each web page against a hand-built gold standard. Metrics such as accuracy, recall, precision, and F_1 were calculated based on the comparison.

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

We created a corpus of web pages to evaluate our algorithms by using results returned by Google search. We issued 16 queries each containing the term "java" and one of the following keywords from the Java programming language: abstract, class, double, final, for, if, import, int, interface, long, private, protected, public, static, void, and while. We downloaded and archived the first 50 results from each of the searches from Google. We removed 52 duplicate pages from the repository and 41 pages, because the pages did not contain HTML, e.g. PDF and word processor documents. Our final corpus contained 707 diverse web pages, both with and without Java source code examples. In these pages, there were 471,536 content segments and 9,796 grouped content segments. For each of these pages, we created by hand a "gold standard," or oracle for correct classifications.

The F_1 statistic is the weighted harmonic mean of precision and recall. In our evaluation, we calculated it separately for both text and source code, but here we show the generic formula we used to calculate both:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{14.1}$$

Classification accuracy indicates the percentage of segments that the algorithm correctly classifies contents in text and source code.

$$Accuracy = \frac{number\ \ of\ \ correctly\ \ classified\ \ segments}{total\ \ number\ \ of\ \ segments} \tag{14.2}$$

We found that F_1 Text = 0.968, F_1 Code = 0.767, and Accuracy = 0.959. It took 14.19 s to train the algorithm and 0.147 s to classify a typical page.

14.3.3 Summary

After completion of this processing, we have a repository of web pages where each web page has been factored into code snippets and text that can be used as metadata. The number of pages available after each step is summarized in Table 14.1.

Total pages downloaded	34,054
Pages with no Java code	21,162
Pages with Java code and text	12,892

Table 14.1: Number of pages after filtering

14.4 Indexing the Repository

Indexing is the process of identifying a set of keys for looking up a document in a repository. With text documents, it is common to index all of the terms, excluding stop words. Choosing what to index and how is an important design decision, because these keys determine how effectively a document is retrieved from the repository. Often, adding metadata to the index, such as the URL, tags, or author of the page, can improve the performance of a search. Simply treating source code snippets as text documents is not sufficient for a number of reasons. Some terms in source code are structurally significant, such as identifiers. Also, source code typically contains few keywords that tell you about what the code does. Consequently, additional processing is needed to ensure that the index contains the appropriate information, so that the most relevant code snippets are returned in response to a user's query.

Our index contains metadata from three different sources: web page, code snippet, and text as shown in Table 14.2. For web pages, we included two metadata fields: url and page title. For code snippets, we included 11 metadata fields: 10 for different identifier types and 1 for a summary of keywords found in a specific code snippet. For text, we included one metadata field that has the summary of keywords found in the text segment associated with a code snippet.

Web page	Code snippet	Text
URL	Keywords from code snippet	Keywords from text
Page title	Package	
	Import	
	Class declaration	
	Class used	
	Extending and implementing class	
	Return type	
	Method declaration	
	Method invocation	
	Variable declaration	
	Comments	

Table 14.2: List of indexable metadata

Information for all the metadata fields were indexed and stored in seperate columns in Lucene. We indexed words from 43,306 snippets, which were compressed into indexes in Lucene with a total size around 71 MB.

14.4.1 Indexing Text Segments

Our approach centers on the idea of using the text surrounding code snippets as metadata, because source code tends to have few words that describe its functionality or what it does. As a result, we need to find the chunk of text that contains the terms that best describes a snippet.

To answer this question, we sampled 200 pages from our repository and manually identified the most relevant text. We obtained this sample by taking the top 50 pages returned by a search using the following four keywords: binary tree, database, hashmap, and socket.

Out of 200 pages, 81 pages were identified to be related to the keywords. Figure 14.2 shows the number of relevant pages for each keyword. Looking more closely, we found that some keywords are too common, such as database, hashmap, and socket. Therefore, the words could appear in web pages that have a topic that is irrelevant to the search query.

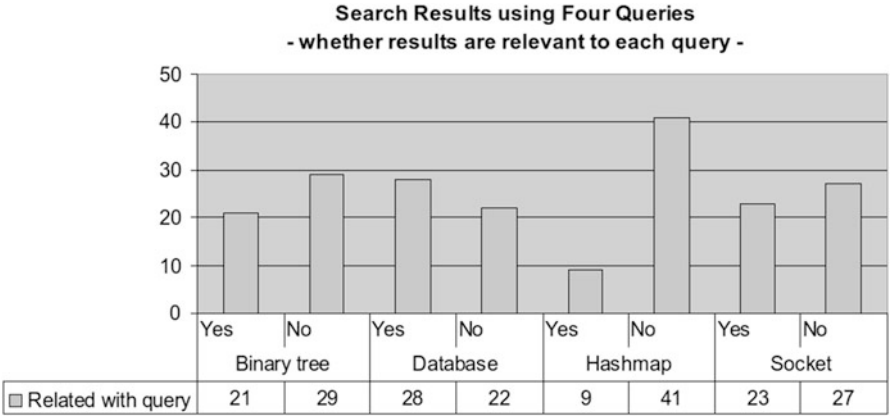
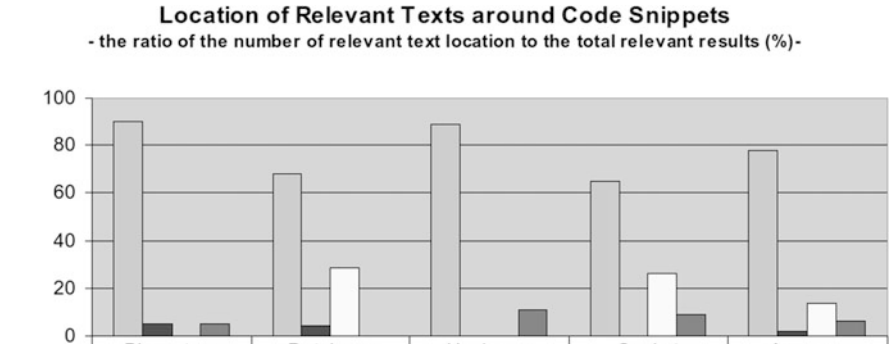


Fig. 14.2 Number of relevant results in each query

We focused on the remaining 81 relevant pages in our study to find the location of a text segment that is most relevant to a specific code snippet. We manually inspected the code snippets and nearby text segments. We found that for 78 % of code snippets, the best text description appeared above them. For 2.25 % of the snippets, the most descriptive text appeared below of them. The best text description appears both above and below 13.75 % of the snippets. Finally, we could not find any relevant text description in the same page for 6.25 % of code snippets. Figure 14.3 summarizes these findings.

Based on these findings, we decided to pair the code snippet with the text segment that appears above it on the page in the repository. The words in the text segment can help describe a code snippet. We have a total of 43,306 pairs of text and code snippets in our repository.



	Binary tree	Database	Hashmap	Socket	Average
above	90	68	89	65	78
below	5	4	0	0	2.25
both	0	29	0	26	13.75
no text	5	0	11	9	6.25

Fig. 14.3 Location of relevant text around code snippets

Each text segment that we collected was parsed using simple word delimitors (e.g. white space, new line) in order to extract all words from the text segment. Due to the fact that many extracted words are very common and not very helpful for searching, (e.g. 'a', 'an', 'the'), these words should be removed from the collection of extracted words. We use a list of stop words [6] to filter them out. The remaining words are changed to lower case and stemmed using the Porter Stemming Algorithm [9]. By ignoring capitalization and reducing each word to its simplest form, we increase the chances of words being matched with the terms in a user's query.

14.4.2 Indexing Code Snippet Segments

Within an integrated development environment (IDE), programmers often search for variables, functions, classes, and other programming constructs by name [11]. Code-specific search engines, such as Krugle, Sourcerer, Google Code Search, and Koders, also provide this functionality. It stands to reason that a snippet search engine should provide this functionality as well. Snippets tend not to be complete syntatically correct, nor can they be compiled and linked. Parsing out programming language constructs is only the beginning. Identifiers usually are not plain English words, but rather are improvised compounds. In addition, comments can be a useful source of metadata and deserve further analysis.

Instead of using fuzzy parsers, such as those used in syntax highlighters, but we tried an approach that has not been used extensively, an incremental compiler.

[6] http://www.ranks.nl/resources/stopwords.html.

To parse out identifiers and comments from the code snippets, we use the Eclipse abstract syntax tree parser.[7] We chose to use this API in Juicy, because it is a robust incremental compiler. Within the Eclipse workbench, the AST parser is capable of parsing code as it is typed and compiling classes as they are saved. For Juicy, it provides two features that are particularly helpful: input type selection and error handling. Input type selection allows the AST parser to be called with a flag that specifies whether the input code is complete source code, a block, or a line of code. This allows the parser to produce more accurate output. If the flag is set incorrectly for an input snippet, the parser will produce errors and we can try again with a different flag.

We collected ten types of structural information from code snippets which are shown in Table 14.3. The first nine types are mainly identifier declarations and invocations, which can be generalized into terms of package, class, variable, and method information.

Extracted identifier types		
Package	Import	Class declaration
Class used	Extending and implementing class	Return type
Variable declaration	Method declaration	Method invocation
Comment		

Table 14.3: Extracted indentifier types from parser

Another metadata field was added to store all the English words in the identifiers, which were found by dividing the identifiers according to internal capitalization. The scheme is also known as "camel case," because uppercase letters in an identifier are taller than the lowercase ones, giving the identifier the appearance of camel-like humps. As specified in the Sun Java Coding Convention, camel case is the recommended standard for aggregating English words to form a meaningful identifier. For this metadata field, we excluded Java keywords such as 'class,' 'for,' 'new,' and 'void.'

The last piece of information to be parsed from code snippets is code comments. Search engines can benefit from comments because they provide information about the context of the code snippets and also contribute potential matches to search terms. Having more keywords associated with a piece of source code will allow users to have more changes that a keyword included in a query matches with a keyword related to a piece of source code. Therefore, javadoc comments, line comments, and block comments were collected and treated as textual information in Juicy.

[7] http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.

14.5 Retrieving and Presenting Search Results

Juicy has both a general and an advanced user interface, as seen in Fig. 14.4. In this section, we will discuss how we use the indexes to obtain matches to the user's query, rank them, and display them.

14.5.1 Matching and Ranking

The basic user interface consisted of a simple text box, just like a typical search engine. With the indexes and metadata available, it was necessary to find a combination that would provide a set of useful results. Furthermore, when using multiple search indexes, we would need to find a way to combine the results.

We found that matching the search terms to all the metadata fields in our indexes at the same time produced many irrelevant results, but did so quickly. After experimenting with different combinations, we found that using three of our indexes produced the largest proportion of good results early in the list. These indexes were page title, keywords from code snippet, and keywords from text. Page title gives a general concept of the whole page. Keywords from code snippet contained identifier names that could explain what the code snippet does. Finally, keywords from text above code snippets, which proved to be relavant to the code snippet, could also

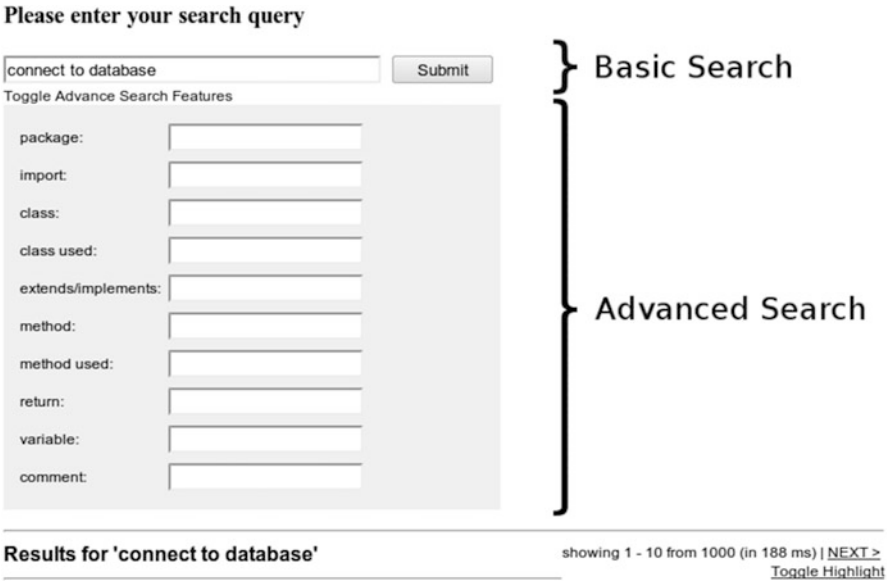


Fig. 14.4 User interface of Juicy

contain words explaining the behavior of the code snippet. In other words, these three metadata fields provide a idea of the purpose of the code snippet.

Using these three indexes gave us three different sets of search results that must be combined into a single ranked list. But the sets are not identical and the snippets are ranked differently. The approach that we used to combine the results revolved around the snippet ID. The final result set consisted of only snippets that appeared in all three results set, which ensures a high degree of relevance to the query. The new ranking was calculated by averaging the ranking from each index. We tried different combinations of weights for aggregating the ranking scores, but found that using equal weights yielded the best results.

The advanced interface allows users to search for a terms in specific indexes, such as class names, method names, and imported packages. Search terms are entered into a text box and these keywords are matched in the corresponding index. The matches are aggregated and ranked using the normalized TF-IDF score produced by Lucene [8].

14.5.2 Presentation

When returning matches, the search engine needs to provide not only the snippet, but also the title of matched web page and a short text summary (Fig. 14.5). It is not obvious what text on a web page is the best summary of the snippet. To solve this problem, we used LDA (Latent Dirichelet Allocation). This algorithm is little known in this space.

LDA, also known as Topic Modeling, can be run offline to create a static set of topics for all the web pages, text segments, and code snippets in our repository. When a search is executed, we use these topics to choose the best text summary among those that were returned and use this in the results presentation.

To evaluate the effectiveness of this approach, we conducted an experiment with a sample of ten queries (Table 14.4). The keywords in the queries were randomly selected from a list of 555 common search terms found in an analysis of a log from the Koders search engine [1].

Query keywords				
1. smtp	2. quicksort	3. list	4. stringbuffer	5. date
6. webservice	7. signature	8. xpath	9. download file	10. base64

Table 14.4: Keywords used for LDA experiment

We took the first 20 results returned by each query and looked at the number of topics in common with the following four candidate sources of text for the summary.

Fig. 14.5 Results presentation in Juicy

1. Best Matched Paragraph. This is the paragraph with the highest frequency of matched topic keywords between the paragraph and its related code snippet. This paragraph can appear anywhere on the web page.
2. Text Segment Above Snippet. Text segments usually contain several paragraphs and are bounded by two code snippets. These are identified during the snippet extraction process.
3. Last Paragraph. We considered using the the last paragraph of a text segment, which appears immediately above a code snippet.
4. Page Title + Text Segment. This candidate included the page title and the text segment. This group provides the largest set of data related to a code snippet. We considered this combination, because it was used to index the snippet.

Figure 14.6 shows the percentage of the 200 examples that had one or more topics in common between the code snippet and the candidate text. The two candidates that contained the most text also had more topics, and consequently had a higher percentage of matches. Eighty-three percent of the code snippets had topics in common with Page Title + Text Segment, while 80.54 % of the Text Segment did. The individual paragraphs, Best Match and Last Paragraphs, fared less well, because they contained less text and fewer topics. The Best Matched Paragraph has a 66.81 % of matched topics and the Last Paragraph has a 59.26 %. In the end, we elected to use the Best Matched paragraph, because it had a reasonable combination of descriptiveness and brevity.

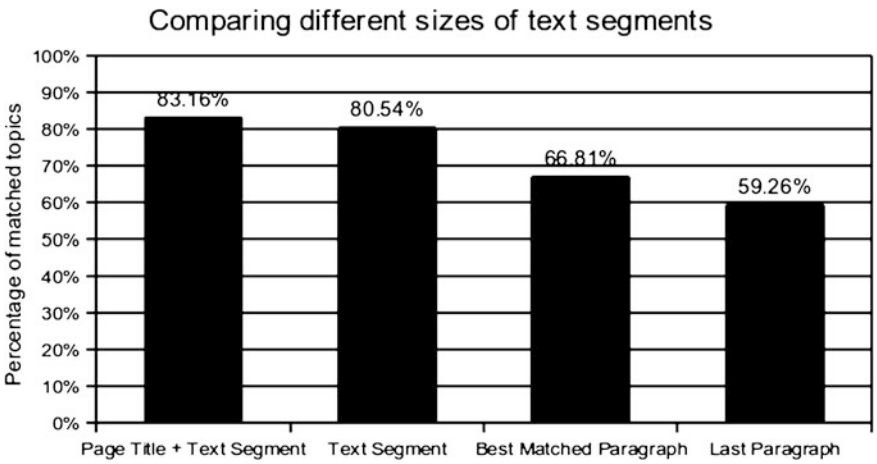


Fig. 14.6 Comparison of Page Title + Text Segment, Text Segment, Most Matched Paragraph, and Last Paragraph

14.6 Conclusion

In this chapter, we describe some design issues for the algorithms used in "Juicy", a search engine that capable of search for Java code snippets. This search engine is designed to help developers who are looking for a small chunk of source code to use as a reminder or to learn unfamiliar syntax. Juicy has been populated with over 34,000 Java tutorial pages that have been crawled from the web. In the repository, the code snippets are treated as primary documents and the surrounding text treated as metadata. Users can search the repository using a basic or advanced interface, using both terms from the source code and the metadata. When presenting the results of a search, Juicy provides a brief description for each code snippet in order to give its users more clues on what the code snippet could mean. By providing this information for each code snippet in search result, users could form better understanding of each code snippet and make better decision when picking them to incorporate with their project.

Our research is a starting point for the work necessary to build a robust snippet search engine. Additional work is needed to improve the usability of the search engine, to enable the back end to work with other programming languages, and to incorporate other kinds of resources (such as emails and forums) in the repository. Finally, the effectiveness and helpfulness of a snippet search engine needs to be evaluated. Nevertheless, Juicy is a proof of concept tool that sheds light on issues in the design and construction of a snippet search engine.

Acknowledgements This material is based upon work supported by the NSF under Grant No. IIS-0846034 and by the UCI Summer Undergraduate Research Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessary reflect the views of the NSF.

Appendix: Seed URLs Used to Create Repository

1. http://java.sun.com/docs/books/tutorial/
2. http://learnola.com/
3. http://www.zetcode.com/
4. http://forum.codecall.net/java-tutorials
5. http://www.dickbaldwin.com/java/
6. http://www.learn-java-tutorial.com/
7. http://www.developer.com/java/
8. http://pages.cpsc.ucalgary.ca/~kremer/tutorials/Java/
9. http://www.beginner-tutorials.com/java-tutorials.php
10. http://www.javabeginner.com
11. http://www.javacoffeebreak.com/
12. http://www.cafeaulait.org/javatutorial.html
13. http://www.javaworld.com/
14. http://en.wikiversity.org/wiki/Java_Tutorial
15. http://leepoint.net/notes-java/index.html
16. http://www.javafaq.nu/java-example.html
17. http://www.java-tips.org
18. http://www.java2s.com/Tutorial/Java/CatalogJava.htm
19. http://www.java2s.com/Code/Java/CatalogJava.htm
20. http://www.java2s.com/Article/Java/CatalogJava.htm
21. http://www.java2s.com/Code/JavaAPI/CatalogJavaAPI.htm
22. http://www.java2s.com/Product/Java/GUI-Tools/CatalogGUI-Tools.htm
23. http://www.tech-recipes.com/category/computer-programming/java-programming/
24. http://www.exampledepot.com/egs/
25. http://www.devdaily.com/java/
26. http://www.roseindia.net/java/
27. http://en.wikibooks.org/wiki/Java_Programming/
28. http://www.codetoad.com/java/
29. http://danzig.jct.ac.il/java_class/
30. http://www.java-samples.com/showtitles.php?category=Java&start=1
31. http://www.algolist.net/Algorithms/
32. http://www.javapractices.com/
33. http://home.cogeco.ca/~ve3ll/jatutor0.htm

References

[1] S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 111–120. IEEE Computer Society, 2009.

[2] C. Fox. A stop list for general text, 1989.
[3] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.
[4] T. Grotton. Combining content extraction heuristics: The combine system. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 591–595, 2008.
[5] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 237–240. ACM, 2005.
[6] Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2009.
[7] Merriam-Webster. *Merriam-Webster's 9th Collegiate Dictionary*. Merriam-Webster. Springfield, MA, USA, 1992.
[8] Michael McCandless, Erik Hatcher, and Otis Gospodnetić. *Lucene in Action*. Manning Publications, second edition, 2010.
[9] M.F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.
[10] J. J. Rodriguez, L. I. Kuncheva, and C. J. Alonso. Rotation forest: A classifier ensemble method, 2006.
[11] Susan Elliott Sim, Charles L. A. Clarke, and Richard C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the Sixth International Workshop on Program Comprehension*, page 180, Los Alamitos, CA, 1998. IEEE Computer Society.
[12] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding api components and examples. In *IEEE Symposium on Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006*, pages 195–202, Brighton, United Kingdom, 2006. IEEE.