Fine Grained Indexing of Software Repositories to Support Impact Analysis

Gerardo Canfora
Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
canfora@unisannio.it

Luigi Cerulo
Research Centre on Software Technology
Department of Engineering - University of Sannio
Viale Traiano - 82100 Benevento, Italy
lcerulo@unisannio.it

ABSTRACT

Versioned and bug-tracked software systems provide a huge amount of historical data regarding source code changes and issues management. In this paper we deal with impact analysis of a change request and show that data stored in software repositories are a good descriptor on how past change requests have been resolved. A fine grained analysis method of software repositories is used to index code at different levels of granularity, such as lines of code and source files, with free text contained in software repositories. The method exploits information retrieval algorithms to link the change request description and code entities impacted by similar past change requests. We evaluate such approach on a set of three open-source projects.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Measurement, Experimentation

Keywords

Mining Software Repositories, Impact Analysis

1. INTRODUCTION

CVS and Bugzilla are two tools for configuration management used with success by the open source community for sharing knowledge. The quality of data, in particular free text, such as bug comments, bug descriptions, and feature proposal definitions, is a critical need in an environment in which no people meetings, no phone calls, and no coffee break discussions are possible [8]. This leads to consider such software repositories interesting data sources, useful for developing text mining techniques to assist project managers and developers in their maintenance activities.

Natural language is widely used in many software engineering artifacts and it is not unusual to find in the literature models based on text mining techniques, information retrieval algorithms, and natural language processing approaches. In [1] a probabilistic information retrieval model has been used to map source code artifacts with documentation. In [14], and [19] text mining techniques have been used in free text contained in software repositories for mining, respectively, concept keyword, and project information.

In this paper we take advantage of free text stored in software repositories to build a textual representation of code entities at different levels of granularity, such as lines of code and source files. This can help in the problem of impact analysis, that is the identification of the work products affected by a proposed Change Request (CR), either a bug fix or a new feature request. Developers can know what are the code entities he/she should work on to resolve a given change request. Project managers can have an estimation of what are the impacted code entities in the next release, useful to focus testing effort.

The method has been introduced in [3] considering a level of granularity restricted to source files. In a set of four case studies, we obtained a precision that ranges between 30% and 78%. In this paper we consider a finer level of granularity, lines of code, and we show that this introduces an improvement at least of 10%, at the cost of spending more time and space for the index.

The paper is organized as follows: next section provides an overview about related work in the field of impact analysis; section three describes the bug resolution process generally adopted by the open source community and what are the free text left by developers; section four introduces the concept of line history table showing how changes at line-of-code level can be recovered from a CVS repository; section five introduces the approach of impact analysis; section six shows the application and validation of the approach in three case studies; the final section concludes the paper with open issues and future works.

2. RELATED WORK

Traditionally, impact analysis has been faced by static analyzing the product [2]. Many approaches are based on traceability analysis and dependence analysis. Traceability analysis identifies affected software entities using explicit traceability relationship. Some methods use a traceability matrix to represent relationships and the impacted objects

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR'06, May 22–23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

are inferred by computing the transitive closure of the matrix. Dependence analysis attempts to assess the effects of a change based on semantic dependencies between program entities; a technique is to use static and/or dynamic slicing [11]. Expert judgement and code inspection are also used; however, expert predictions have been shown to be frequently incorrect [12], and source code inspection can be prohibitively expensive [15]. The availability of data on software process, such as those deriving from software and change repositories, can provide new opportunities for impact analysis. In particular, approaches to predict the impact and propagation of changes can be found in [20, 18]. They use heuristics, such as historical co-change and co-authorship, to derive the set of impacted source file or program entities. A method to evaluate the performance of change propagation heuristics has been introduced in [9]. These methods predict the impacted files starting from a given initial source file, while the approach we present in this papers starts from a change request description.

3. FREE TEXT IN SOFTWARE REPOSITORIES

The resolution of a new CR, in many open source projects tracked by Bugzilla and CVS, usually follows a very simple workflow. A reporter proposes a new CR that can be a bug he/she discovered or an enhancement feature he/she likes to suggest. The CR is stored in the new CR database, after a validation performed by the maintainer of the project to confirm it exists. A developer that has dealt with similar CRs in the past has a wide knowledge of the source code involved and can easily locate the code entities that should be changed. Otherwise, if he/she does not have such knowledge, it is usual to ask for the help of other developers that have resolved similar problems in the past. It is not rare to find in the discussion comments of a CR, topics regarding similar past behaviors resolved in other CRs. An *assigned to* relationship links the developer with the CR. The resolution of the CR evolves to a fixed CR with a commit, in the CVS database, of the source code changes that resolves the CR (*impact* relationship). This relation does not exist in Bugzilla database but, as suggested in [6], it can be derived because usually developers keep track of the impacted files by inserting in the CVS commit comment the id number of the CR. A *resolved by* relationship links the developer, author of the resolution change, and the fixed CR.

This process involves a lot of information both structured and not structured, e.g. composed of free text. A file revision is composed by a set of fields: *revision*, is a number that increases when new changes are committed by the developer; *date*, is the date and time of check-in; *author*, is an identificator of the person who did the check-in; *state* assumes one of the following values: 'exp' means experimental and 'dead' means that the file has been removed; *lines*, the number of lines added and deleted with respect to the previous version of the file; and a final block of free text that contains informal data entered by the developer during the check-in process.

A CR is in many cases represented in XML and it is enclosed generally in a *bug* or *issue* tag containing: *bug-id*, a unique identifier assigned by Bugzilla; *creation-ts*, the date and time of CR creation; *short-desc*, a short description; *product*, the product name; *component*, the component of

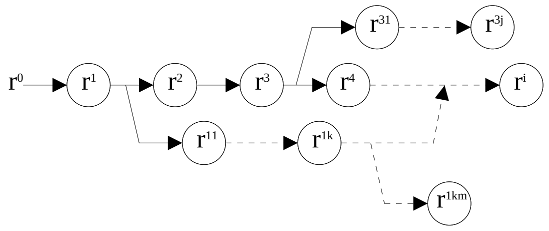


Figure 1: Source file revisions graph

Table 1: Line history table

r^1	...	$r^{ijk...l-1}$	$r^{ijk...l}$	line #
	...			1
	...			2
...	...	...	...	...
	...			n

the system; *reporter*, who has submitted the CR; *assigned-to*, who was assigned the CR for resolution; and *long-desc*, a structure comprising a long description of the CR, *thetext*, who submitted it, *who*, and when, *bug-when.*

4. FINE GRAINED ANALYSIS OF CVS REPOSITORIES

CVS handles revisions of textual files by storing the difference between subsequent revisions in a repository. It provides only information on files and differences, but not which code entities have been changed. For an analysis of fine-grained entities, another preprocessing step is required: each revision is compared with its predecessor and the changes are mapped to entities. In [21] syntactic entities, such as functions, methods, and variable declarations have been considered. In this paper we refer to lines of code entities and recover the history of source code modification in terms of lines that have been added, changed, and deleted during the evolution of a source file. In doing that we will introduce the concept of *line history table* as a tool to visualize source file evolution at the line-level granularity. A similar concept has been introduced in [4] for different purpose.

Figure 1 shows a typical source file revisions graph with different development trunks. Each revision is identified with a sequence of numbers positions: $ijk...l$, in which the last, l, is incremented by one every time a new revision is committed. Revision r^0 represents the empty file. A development trunk can be started from every revision and at some point it can be merged to the trunk from which it derives. When a new development trunk starts the revision identifier is incremented with another number position at the end, initially equal to 1.

Each revision is compared with its predecessor by using the *diff* tool. If a revision is a merge of multiple predecessors, it should get a special treatment, while a revision with no predecessors is compared against an empty file.

A line history table depicts a particular revision $r^{ijk...l}$ and contains a row for each of its source line, and a number of column for all previous revisions belonging to the path that reaches the revision r^0 (table 1). For example, the line history table of revision r^{31} contains the columns: r^1, r^2, r^3, and r^{31}. If merges will not be considered for each

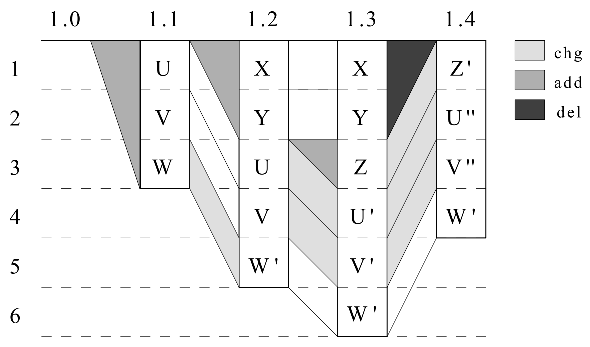


Figure 2: Revisions example

revision, only one line history table can exist. The i-*th* row of the table shows the history of the i-*th* line of the revision by using a marker, ‘a’ or ‘c’, in the column corresponding to the revision the line has been respectively added or changed. Some constrains hold for the relative position of markers:

- a line is added to revision only once, then each row contains one, and only one ‘a’;
- a line can be changed only if it has been added in a previous revision, then in each row, ‘a’ precedes each changes ‘c’.

A line history table can be built from the output of a *diff* command. When comparing two files, *diff* finds sequences of lines common to both files, interspersed with groups of differing lines called *hunks* [13]. There are many ways to match up lines between two given files. The algorithm inside *diff* tries to minimize the total hunk size by finding large sequences of common lines interspersed with small hunks of differing lines.

A diff command is performed between two revisions, called the right and the left revision and the output of a diff command is a sequence of tuples x, yTw, z, in which x, y and w, z are two line number intervals of respectively the left and right revision, and T can be: a, c, or d which means respectively that the left interval has been added, changed, or deleted in the right interval. The column r^i of a line history table of revision r^{i+k} is computed from the output of a diff command performed between revisions r^{i-1} and r^i. Each right interval is translated to line numbers of revision r^{i+k} by examining what happens to this interval in each subsequent revision r^{i+j} for $j = i+1, ..., k$.

Table 2 shows the output of a *diff* command for four revisions performed on a text file depicted in figure 2. The first revision (1.1) contains three lines. The second revision (1.2) adds two more lines on the top and changes the third line of the previous revision. The third revision (1.3) adds a new line in third position and changes the two lines that follows. The last revision (1.4) deletes the first two lines and changes the three lines that follows. Revision 1.0 means the empty file.

Table 4 shows the line history table of revision 1.4. Diff information between revision 1.0 and 1.1 shows an add of the first three lines (`0a1,3`). The add range will change if we look what happens in subsequent revisions 1.2, 1.3, and 1.4. In revision 1.2, as two more lines are added to the top (`0a1,2`), the range is shifted from 1,3 to 3,5. In revision 1.3,

Table 2: Diff information between revisions

rev1	rev2	diff
1.0	1.1	`0a1,3`

1.1	1.2	`0a1,2`
		`3c5`

1.2	1.3	`2a3`
		`3,4c4,5`

1.3	1.4	`1,2d0`
		`3,5c1,3`

Table 3: Line history table of revision 1.3

1.1	1.2	1.3	line #
	a		1
	a		2
		a	3
a		c	4
a		c	5
a	c		6

as one line is inserted in position 3 (`2a3`), a shift of size one places the lines in positions 4 to 6. In revision 1.4 two lines have been deleted and the final line positions are shifted up in the range 2,4 as shown in the first column of table 4. Other columns are computed in a similar way by shifting up and down line positions in order to take into account line additions and deletions in subsequent revisions. In this way, we obtain the diff information for each revision translated to line positions relative to the reference revision. Line changes are slightly different if the reference revision is 1.3, as shown in table 3.

In this example, for the purpose of simplicity, we have not considered cases in which the subsequent add and delete are inside the range to be transformed, nor the case in which changes have different left and right range sizes. They can be modeled by considering a range split in the first case and by transforming the change in an add/del + change operation in the second case. The add/del are computed in order to have change subpart of the same size for left and right ranges.

A line history table can be used in a number of ways. Given a system release, the line history table of each revisions belonging to that release can be computed. As an example, for each source line, the number of past revisions until its last change is an indicator of its age, while the number of changes explains its stability.

In the context of this paper we are interested to use the line history table of the current system release, and represent each line of code with free text related to revisions in which the line has been added, or changed. This comprises revision comments and short and long descriptions of CRs that impact those revisions.

Table 4: Line history table of revision 1.4

1.1	1.2	1.3	1.4	line #
		a	c	1
a		c	c	2
a		c	c	3
a	c			4

5. IMPACT ANALYSIS APPROACH

Our approach to impact analysis is shown in figure 3. A descriptor builder process links free text contained in software repositories with source code entities and an indexing process generates the index used by an information retrieval algorithm to retrieve the ranked list of code entities impacted by a new CR. The hypothesis is that revision comments and impacted CRs are a good descriptor of code entities, such as source files and lines of code, to support impact analysis of new CRs. This is granted by the fact that CVS and Bugzilla are extensively used as tools for knowledge sharing during the software development process with textual data of acceptable quality. We use textual similarity to compute the similarity between a new CR descriptor (i.e. *short-desc*, or *short-desc* + *long-desc*) and the set of source code entities descriptors. The most similar code entities are retrieved and presented to the developer as a first ranked list of probable impacted code entities from which change propagation can start. Textual similarity is a critical part of our approach. The Information Retrieval community dealt with text similarity for a long time. Given a set of text documents and a user information needs represented as a set of words, or more generally as free text, the information retrieval problem is to retrieve all documents relevant to the user [17]. In information retrieval, queries and documents are described by a set of index terms. Let $T = \{t_1, t_2, \ldots, t_n\}$ denotes the set of term used in the collection of documents. Both a document d and a query q are represented as a vector $(x_1, x_2, \ldots, x_n)$ with $x_i = 1$ if t_i belong to the document/query and $x_i = 0$ otherwise. In our approach we have used a probabilistic information retrieval model in which the relevance of a document with respect to a query is computed by evaluating $P(R|d, q)$, that is the probability that a given document d is relevant to a given query q. Different probabilistic models have been proposed in literature to evaluate this probability. We have used the model introduced in [10]. It assumes that each term is associated with a topic, and that a document may be about the topic, or not. Statistic measures about the term occurrences in documents are used to estimate the probability. In particular, a document d is scored with respect to a query q by using the following scoring function:

$$S(d, q) = \sum_{t \in q} W(t) \qquad (1)$$

that sums the weight of each query term with respect to the document d on the basis of a weighting function W. An overview of weighting functions can be found in [5]. We have used the following [10]:

$$W(t) = \frac{TF_t(k_1 + 1)}{k_1\left((1 - b) + b\frac{DL}{AVDL}\right) + TF_t} \log \frac{N}{ND_t}$$

where TF_t is the frequency of term t in the document, DL is the document length (i.e. number of terms), $AVDL$ is the average document length in the collection, N is the number of documents in the collection, and ND_t is the number of documents in which the term t appears. The constant k_1 determines how much the weight reacts to increasing TF_t, and b is a normalization factor. We have used the values of $k_1 = 1.2$ and $b = 0.75$, which are recommended values for generic English text.

The vector representation is built through an indexing process composed by a number of standard steps usually performed to improve the retrieval performance. The first step, *term tokenizer*, regards the subdivision of free text in a sequence of index terms. A token is a sequence of alphanumeric characters separated by non-alphanumeric characters. In our case we have discarded tokens consisting only of digits. The second step, *stemmer*, serves to lead a term to its root. For example verb conjugation is led to the infinitive verb, plural is led to singular, and so on. Terms are stemmed in order to collapse terms with the same meaning into a single term. In our case we have used the Porter stemmer algorithm for English [16]. The third step, *stopper*, serves as a filter of common words that are not discriminant for the document. We have used a common stop word vocabulary used in the context of English text retrieval enriched with a set of words picked up from the software system domain. For example, we have discarded words such as *bug*, *feature*, and words related to the system under consideration such as *argouml*, *gedit*, and so on. The set of so obtained terms are counted for each document and stored within the document identifier in two data structures, namely direct and inverted indexes. The first stores term occurrences within a document, while the second stores term occurrence among the collection.

In the next two subsections we consider the descriptor building process of code entities at two different levels of granularity, source files and lines of code, and in the next section we show that indexing finer grained code entities gives, in most cases, a better performance in retrieving the impacted source files. Moreover, fine grained indexing can give more rich information as a result because, within the source file, the set of impacted lines of code is also returned.

5.1 File indexing and file retrieval

Source files indexing is performed on descriptors built for each source file belonging to a system release. A source file descriptor, $D(sf)$, is defined as:

$$D(sf) = \sum_{cr \in impact(sf)} D(cr) + \sum_{r \in revision(sf)} D(r)$$

Where $D(cr)$ is the descriptor of the change request cr that impact the source file sf obtained by the concatenation of its short and long descriptions; $D(r)$ is the descriptor of the revision r of the source file sf obtained from its commit comment; and $+$ is the operator of string concatenation.

Source file retrieval is performed by computing the score value of each source file sf with respect to the new CR descriptor by using the equation 1, $S(sf, CR)$. The set of source files in descending order with the score value is returned to the user. Source files with a score value less than a threshold constant t are discarded, usually $t = 0$.

5.2 Line of code indexing and file retrieval

Lines of code indexing is performed on descriptors built for each line of code belonging to each source file of a system release. A line of code descriptor, $D(lc)$, is defined as:

$$D(lc) = \sum_{cr \in impact(lc)} D(cr) + \sum_{r \in revision(lc)} D(r)$$

Where $D(cr)$ is the descriptor of the change request cr

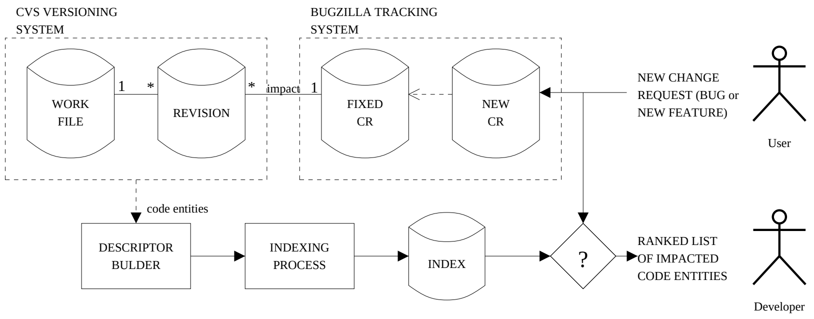


Figure 3: Impact analysis process

that impact the line of code lc; $D(r)$ is the descriptor of the revision r in which the line of code lc has been added or changed; and $+$ is the operator of string concatenation. While $impact(sf)$ and $revision(sf)$ can be derived directly from software repositories, those relative to a line of code, $impact(lc)$ and $revision(lc)$, can be derived by using the line history table of the current system release and considering only those revisions in which the line has been added or changed.

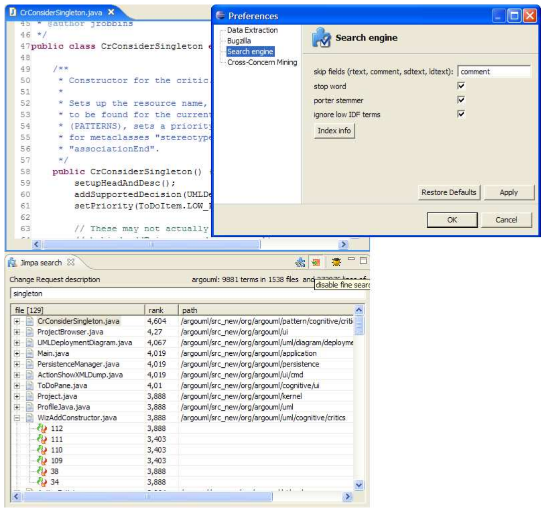
Since, $impact(lc) \subseteq impact(file(lc))$, and $revision(lc) \subseteq revision(file(lc))$, then $D(lc) \subseteq D(file(lc))$, where $file(lc)$ is the source file lc belongs to. When indexing the line-of-code level, the score value of each line, lc, with respect to the new CR descriptor is computed by equation 1, $S(lc, CR)$. We score a source file, sf, by computing the maximum score of lines belonging to it.

$$S(sf, CR) = \underset{lc \in sf}{MAX}\,(S(lc, CR))$$

Other score functions can be defined but this one has given good results. Lines of code indexing is more expensive, in space and time, than source file indexing, by a factor depending on the average length of source files.

5.3 Tool support

We have developed an Eclipse plug-in, named Jimpa, in order to support both indexing and source file retrieval. All steps are completely automated, including the download from the CVS and Bugzilla repositories. As shown in figure 4, the user can write a short explanation of the impacted source files he/she wants to search for. The user can choose the index to use, either fine or coarse grained, respectively lines of code and source files. The search is performed among the current project and the set of source files, ranked by their relevance with change request description, is returned by the search engine and shown in the bottom. The list shows, for each source file, the project relative path location and the relevance weight for the change request description. For fine grained index, the set of finer code entities, such as ranges of impacted lines of code, are shown within the source file. The tool provides the support for setting information retrieval properties such as stop word list, stemmer algorithm, and fields to be included or excluded from the indexing process. Moreover, parameters to access a Bugzilla site can be set in the preference dialog of each Eclipse project. Jimpa runs under Eclipse 3.1 and is hosted on an Eclipse update site at the following URL: http://cise.rcost.unisannio.it/updates/.



```
 * @author jrobbins
 */
public class CrConsiderSingleton e

    /**
     * Constructor for the critic.
     *
     * Sets up the resource name,
     * to be found for the current
     * (PATTERNS), sets a priority
     * for metaclasses "stereotype
     * "associationEnd".
     */
    public CrConsiderSingleton() {
        setupHeadAndDesc();
        addSupportedDecision(UMLDe
        setPriority(ToDoItem.LOW_P

        // These may not actually
```

skip fields (rtext, comment, sdtext, ldtext): comment
stop word ✓
porter stemmer ✓
ignore low IDF terms ✓

Change Request description: singleton
argouml: 9881 terms in 1538 files and

file [129]	rank	path
CrConsiderSingleton.java	4,604	/argouml/src_new/org/argouml/pattern/cognitive/criti
ProjectBrowser.java	4,27	/argouml/src_new/org/argouml/ui
UMLDeploymentDiagram.java	4,067	/argouml/src_new/org/argouml/uml/diagram/deployme
Main.java	4,019	/argouml/src_new/org/argouml/application
PersistenceManager.java	4,019	/argouml/src_new/org/argouml/persistence
ActionShowXMLDump.java	4,019	/argouml/src_new/org/argouml/ui/cmd
ToDoPane.java	4,01	/argouml/src_new/org/argouml/cognitive/ui
Project.java	3,888	/argouml/src_new/org/argouml/kernel
ProfileJava.java	3,888	/argouml/src_new/org/argouml/uml
WizAddConstructor.java	3,888	/argouml/src_new/org/argouml/uml/cognitive/critics
112	3,888	
111	3,403	
110	3,403	
109	3,403	
38	3,888	
34	3,888	

Figure 4: Tool snapshot

Table 5: Open-Source projects

project	files	lines	lines/files	fixed CRs	age
Gedit	117	47913	409.5	116	9 years
ArgoUML	1538	272076	176.9	670	7 years
Firefox	89	42580	467.4	591	4 years

6. CASE STUDY

We have applied the impact analysis approach in three case studies with different characteristics (Tab. 5). The first, Gedit, is a general purpose text editor of the GNOME desktop environment written in C. The second, ArgoUML, is an UML modeling tool written in Java. The third, Firefox, is an Internet browser written in C++.

The results have been assessed using two widely accepted information retrieval metrics, namely, *Precision* and *Recall* [17]. *Precision* is the ratio between the number of relevant documents retrieved for a given query and the total number of documents retrieved for that query. *Recall* is the ratio between the number of relevant documents retrieved for a given query and the total number of relevant documents for that query. In our case study recall and precision indicates how many of the right impacted files have been correctly predicted (recall) and how many of the predicted impacted files are right (precision). We use the same methodology used for evaluating an information retrieval algorithm, that

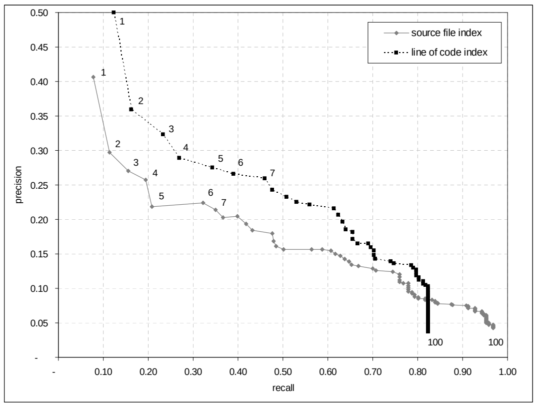


Figure 5: Gedit results

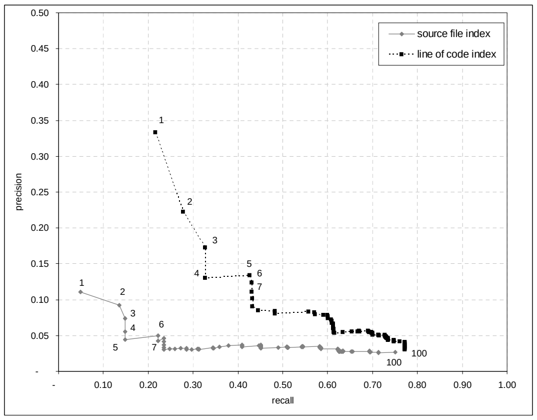


Figure 6: ArgoUML results

is, the retrieved ranked list of documents is considered at different cut levels [17]. A cut level N is the list of the first N ranked documents. For each cut level the behavior of precision and recall is analyzed and traced on a graph.

We have conducted the evaluation by using the leave-one-out assessment technique [7]. For a given CR we have predicted the set of impacted files by using an index without descriptors regarding that CR. The predicted set of files is then compared with the oracle set, that is the files actually impacted by that CR, recovered by considering the presence of the Buzilla id number in the revision comments of the files [6]. We think this is a good oracle as CRs managed in Bugzilla follow basically some accepted guidelines, and one of these is to indicate in the check-in comment the Bugzilla id that identifies the relevant CR.

The performance has been evaluated by using both source file and line of code indexes. Figures 5, 6, and 7 show the results for each of the three systems considered. The curves have been traced by observing the top 100 files ranked by the scoring function and averaging the precision and recall on the number of CRs considered for each system. Each figure contains two curves, relative to source file index and line of code index. A curve contains 100 points, one for each cut level starting from 1 to 100. The first set of points should be read as a measure of overall precision, while the last set of points as a measure of overall recall.

Indexing lines of code produces improvements ranging between 10% and 20% of top 1 precision for each case considered. Top 100 recall is better for source file index in two cases, ArgoUML and Firefox. An evident improvement is reported for ArgoUML for which the top 1 precision is almost 20% better for lines of code index than for source files index. Is this related to the fact that ArgoUML is written in Java?

Why different performance behaviors occurs for different systems needs to be further investigated. For sure it depends on how software repositories are used in software projects. Usually, projects share a common usage practice driven by the configuration management system but with a some slightly deviation driven by the members of the project.

Table 6 shows the time and the disk space needed to build both source file and line of code indexes for each case considered. Data explains that the increment of the cost, in terms of time and space required to index lines of code, grows percentually more than the increment of performance gained. However this is not a drawback at all because the indexing process takes place only one time to set-up the environment and successive index updates are performed incrementally. Regarding file retrieval response time there is a no evident cost increment as shown in table 7.

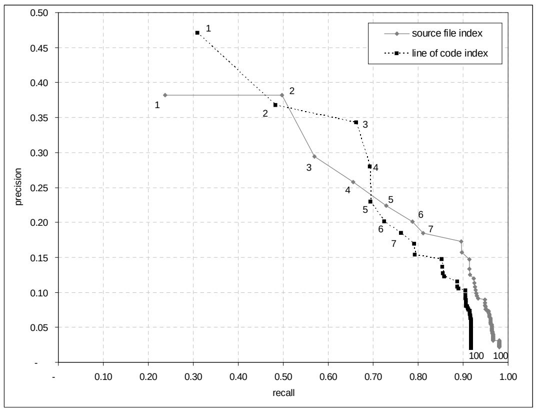


Figure 7: Firefox results

Table 6: Time and space needed to build the indexes

project	source file index		line of code index	
	time	space	time	space
Gedit	2 *sec*	360 *KB*	59 *sec*	8.5 *MB*
ArgoUML	19 *sec*	1.82 *MB*	390 *sec*	52.5 *MB*
Firefox	3 *sec*	300 *KB*	63 *sec*	3.3 *MB*

7. CONCLUDING REMARKS

Software and change repositories give new opportunities to support the software development process. In this paper

Table 7: Average file retrieval response time

project	source file index	line of code index
Gedit	16.2 *msec*	31.1 *msec*
ArgoUML	266.3 *msec*	375.5 *msec*
Firefox	15.1 *msec*	30.4 *msec*

an approach to predict impacted files from a change request definition has been presented. The approach exploits information retrieval algorithms performed on code entities, such as source files and lines of code, indexed with free text contained in software repositories. We show, in particular that indexing fine grained entities, improves precision, at the cost of indexing a much higher number of code entities.

The empirical validation conducted on three open source projects has given promising results. However, quality of text and project maturity are two factors that strongly impact the performance of the approach. Sometime CVS comments are used for communication rather than for description purpose and in almost all projects there is an initial period of transition that generates noise in both CVS and Bugzilla repositories. Indexes can be build, effectively, only for mature projects for which a huge amount of historical data is available. For young and immature projects this approach fails.

We feel that a direction of improvement should be the introduction of a filter that selects the text to index. The filter should be able to select only the text that well describes the indexed code entities. As a very simple example, CVS comments regarding maintenance and merged revisions should be discarded because, usually, they have not useful information for indexing.

The open source community uses other repository for knowledge sharing, such as: mailing lists, newsgroups, and IRC conversations. They are rich of free text and it should be interesting to investigate how this information can be used in conjunction or as an alternative to CVS and Bugzilla.

8. REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[2] R. S. Arnold and S. A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301. IEEE Computer Society, 1993.

[3] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium.* IEEE Computer Society, 2005.

[4] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *ICSM '01: Proceedings of 17th IEEE International Conference on Software Maintenance*, page 364. IEEE Computer Society, 2001.

[5] F. Crestani, M. Lalmas, C. J. V. Rijsbergen, and I. Campbell. Is this document relevant?...probably: a survey of probabilistic models in information retrieval. *ACM Comput. Surv.*, 30(4):528–552, 1998.

[6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of 19th IEEE International Conference on Software Maintenance*, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society.

[7] K. Fogel and M. Bar. *Cross-Validatory Choice and Assessment of Statistical Predictions (with Discussion)*, volume 36. J. the Royal Statistical Soc., 1974.

[8] K. Fogel and M. Bar. *Open Source Development with CVS.* Coriolis, 2001.

[9] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[10] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808, 2000.

[11] M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Syst. Softw.*, 31(3):197–214, 1995.

[12] M. Lindvall and K. Sandahl. How well do experienced software developers predict software change? *J. Syst. Softw.*, 43(1):19–27, 1998.

[13] W. Miller and E. W. Myers. A file comparison program. *Software Practice and Experience*, 15(11):1025–1040, 1985.

[14] M. Ohba and K. Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *IEEE 27th International Conference on Software Engineering - The 2nd International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[15] S. L. Pfleeger. *Software Engineering: Theory and Practice.* Prentice-Hall, Upper Saddle River, NJ, 1998.

[16] M. F. Porter. *An algorithm for suffix stripping.* Morgan Kaufmann Publishers Inc., 1997.

[17] B. Ribeiro-neto and Baeza-yates. *Modern Information Retrieval.* Addison Wesley, 1999.

[18] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Transactions on Software Engineering*, 30:574–586, Sept. 2004.

[19] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *IEEE 27th International Conference on Software Engineering - The 2nd International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[20] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, 2004.

[21] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *IEEE 26th International Conference on Software Engineering - The 1st International Workshop on Mining Software Repositories*, pages 2–6, 2004.