Supporting Change Request Assignment in Open Source Development

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

Software repositories, such as CVS and Bugzilla, provide a huge amount of data regarding, respectively, source code and change request history. In this paper we propose a study on how change requests have been assigned to developers involved in an open source project and a method to suggest the set of best candidate developers to resolve a new change request. The method is based on the hypothesis that, given a new change request, developers that have resolved similar change requests in the past are the best candidates to resolve the new one. The suggestion can be useful for project managers in order to choose the best candidate to resolve a particular change request and/or to construct a competence database of developers working on software projects. We use the textual description of change requests stored in software repositories to index developers as documents in an information retrieval system. An Information Retrieval method is then applied to retrieve the candidate developers using the textual description of a new change request as a query.

Case and evaluation study of the analysis and the methods introduced in this paper has been conducted on two large open source projects, Mozilla and KDE.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; D.2.9 [**Software Engineering**]: Management

Keywords

Mining Software Repositories, Maintenance Task Assignment, Information Retrieval

1. INTRODUCTION

The availability of data on software process, such as those deriving from software and change repositories, can provide new opportunities of data analysis, such as change propagation [19], fault analysis [10], software complexity [7], and software reuse [14].

In this paper we propose a method that exploit CVS [2] and Bugzilla [1] repositories to support the project manager in selecting the most appropriate developers to be considered in the resolution of a new bug or the implementation of a new enhancement feature, both named in this paper Change Request (CR). The main contribution of this paper are: a) support for the project manager in selecting the most appropriate developers to be considered in the resolution of a new bug or the implementation of a new enhancement feature; b) a start point for classifying developers among their competencies and capabilities for build, for example, a developer competence database; c) observation on how maintenance activity has been followed in two open source projects, Mozilla and KDE, since 2003.

The paper is organized as follows: the next section gives an overview about two software repositories widely used by the open source community, CVS and Bugzilla; the third section introduces our developer selection approach by using information stored in software repositories; the fourth section present a case study in which the developer selection approach is applied and analyzed in two open source projects, Mozilla and KDE; the final section concludes the paper and gives some suggestions for further development.

2. SOFTWARE REPOSITORIES

In open-source community project management is usually performed with two systems: a source version control system, such as CVS, and a bug tracking system, such as Bugzilla. They provide a huge amount of data regarding respectively source code and change request history.

CVS can handle revisions of textual files by storing the difference between subsequent revisions in a repository. In open source projects CVS is used to store every files regarding the software system under development: this usually comprises source code, project documentation, internationalization text, arts, and help text. Revisions are stored together with the user id that performs the commit, the date, and a textual block representing the user comment to the change. The complete history of file revisions can be retrieved.

Bugzilla is a database for bugs. Its main use is to report a bug and to assign these bugs to the appropriate developers. However, Bugzilla considers also requests for enhancement. In the open source community it is largely used for proposing

SAC'06 April 23-27, 2006, Dijon, France

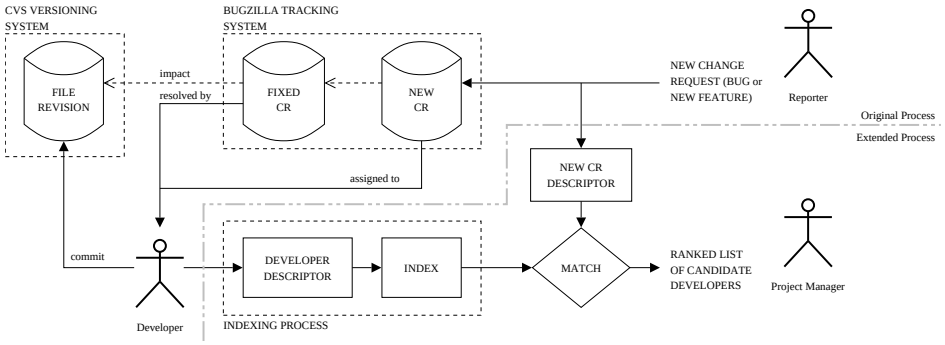

Figure 1: Candidate developer selection process

changes, assigning tasks, discussing different enhancement solutions and so on. Developers use Bugzilla to keep a to-do list as well as to prioritize, schedule and track dependencies. The Bugzilla database is accessible from http and some of its reports can be exported in XML. Bugzilla and CVS are often used in the open source community in a disciplined and standard way [9]. In this paper we refer to Change Request (CR) as an item tracked by Bugzilla that can be either a bug or a request for enhancement.

Figure 1 shows what happens in the resolution of a new CR, of an open source project tracked by Bugzilla. A reporter propose a new CR that can be a bug he/she discovered or an enhancement feature he/she likes to suggest. Before this CR is stored in the new CR database, a quality assurance person, which is usually the maintainer of the project, checks it in order to confirm it exists. The new CR is assigned a priority and a severity level and is stored in the database. On the basis of priority and/or familiarity with CRs stored in the database, a volunteer developer can pickup a new CR in order to resolve it. A new CR can, of course, be directly assigned by the maintainer of the project to a developer he/she know to be able to resolve it. In both cases an *assigned to* relationship links the developer with a new CR. The resolution of the CR evolves and in the most simple case it is concluded with a commit, in the CVS database, of the source code changes that resolves the CR (*impact* relationship). This relation cannot be recovered directly from Bugzilla database but needs to be derived by considering some heuristics [8]. Usually developers keep track of the files impacted by a CR by inserting in a CVS commit comment the id number of the CR tracked by Bugzilla. A *resolved by* relationship holds between the developer, author of the resolution change, and the fixed CR. Exception of this quite simple process can be happen if more complex resolution cases occurs, as for example the CR is blocked by other CRs, it is reassigned to another developer, or it is a duplicate of another CR reported/resolved previously.

3. DEVELOPER SELECTION APPROACH

In this section we present an approach to select the set of best candidate developers to resolve a new CR. The approach is based on the hypothesis that the best candidate developers are those that have resolved similar CRs in the past. For similarity we intend the likeness with respect to the CRs text content, that is a representation in natural language of the CRs semantic. Textual similarity is a critical part of our approach. Information Retrieval community dealt with text similarity for a long time. Given a set of text documents and a user information need represented as a set of terms or more generally as free text, the information retrieval problem consists of retrieving all documents relevant to the user [18]. The effectiveness of an information retrieval systems depends largely on the quality of text used to represent documents. In the open source community, CVS and Bugzilla are extensively used as tools for sharing knowledge during the software development process. The quality of textual information, such as bug comments, bug descriptions, and feature proposals definition is a critical need in an environment in which no people meetings, no phone calls, and no coffee break discussions are possible.

We have extended the basic schema of figure 1 by introducing a support for the project manager to select the most appropriate developer for a new CR. For doing that we have integrated an information retrieval process in the original Bugzilla tracking process. Developers are considered as documents and new CRs are considered as queries. As shown in figure, both for developers and new CR, a textual descriptor is builded. In particular a developer D_i is represented as: $D_i = \{CR^j_{sd}, CR^j_{ld}\}_{j \in resolvedBy(D_i)} + \{R^k_{notes}\}_{k \in commitedBy(D_i)}$, where CR^j_{sd} and CR^j_{ld} are respectively the short and the long description of the CRs the developer have fixed and R^k_{notes} is the notes the developer have included in CVS commit operations. A new CR is represented with its short and long descriptions, those inserted by the reporter of the CR when it was proposed. Developer descriptors are indexed and a matching algorithm compares the new CR descriptor with the developer descriptors stored in the index, resulting in a ranked list of candidate developers. We have used a probabilistic information retrieval model in which the relevance of a developer with respect to a new CR is computed by evaluating $P(R|d,q)$, that is the probability that a given developer d is relevant to a given new CR q. This approach assumes a representation of descriptors as vectors of terms. Let $T = \{t_1, t_2, \ldots, t_n\}$ denote the set of terms used in the collection of developer descriptors. Both developer d and new CR q are represented as a vector $(x_1, x_2, \ldots, x_n)$ with $x_i = 1$ if t_i belong to the descriptor and $x_i = 0$ otherwise. The vector representation is built through an indexing process in which some standard transformation are performed. Free text is first *tokenized*: in our case we have discarded terms consisting only of digits. Each term is then processed by a *stemmer*, that serves

to lead a term to its root. For example verb conjugation are leaded to the infinite verb, plural are leaded to singular, and so on. In our case we have used the Porter stemmer algorithm for English words [17]. Stemmed terms are then filtered on the basis of a *stopwords list*, that discards words that are not significative to represent a developer. 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 the words *bug*, *feature*, and words regarding the name of the system under consideration such as *kde*, *koffice*, and so on.

The probability $P(R|d,q)$ can be evaluated in different ways [18]. We have used the model introduced in [11]. It assumes that each term is associated with a topic, and that a document may be about the topic, or not. The following scoring function is a statistical measure to estimate the probability [6]: $S(d,q) = \sum_{t \in q} W(t)$. It 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 [6]. We have used the following [11]:

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

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 use the values of $k_1 = 1.2$ and $b = 0.75$, which are recommended values for generic English text.

We have developed a toolkit in order to support the indexing process and the extraction of textual information contained in software repositories. All steps are completely automated, including the download of data from CVS and Bugzilla repositories. In particular we have developed a java software that parses a cvs log file and a script that can submit a request to the Bugzilla repository and download the output of a set of fixed CRs. For doing that we have used the WebL scripting language [12] together with its information extraction features in order to parse the html Bugzilla output and generate an XML version of a CR. The java software correlates developers with each Bugzilla fixed CRs and generates the set of descriptors in XML ready to be parsed by the indexing engine. The indexing process and the matching algorithm have been implemented in java by using an information retrieval framework named Terrier [16].

4. CASE STUDY: MOZILLA AND KDE

Mozilla and KDE are two quite large open source projects started respectively in 1998 and 1996. The first in an internet suite that comprises a browser (Firefox), an email client (Thunderbird), and other tools regarding editing and authoring of web pages. The second is a graphical desktop environment for Unix workstations with many desktop applications such as Koffice, Kcal, Kpdf, Konqueror, etc.

In this case study we choose an interval of observation starting from 01-jan-2003 to 01-jan-2005. The choice is arbitrary as we obtained similar result with different observation intervals. The chosen interval is sufficient wide to comprise the average behaviors that happens in each project and it is sufficient far from the beginning to cut off the transition state of each project in order to take in account only the working rate. Table 1 shows for both project the number of involved developers and the number of fixed CRs since 01-jan-2003. Figure 2 shows the CRs fixing progress in the time interval under observation.

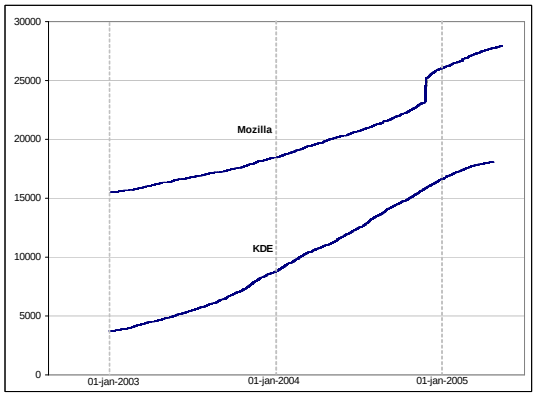

Figure 2: Number of fixed CR since 01-jan-2003

Table 1: Open Source projects

	involved developers since 01-jan-2003	fixed CRs since 01-jan-2003	project starts
Mozilla	637	12447	oct-1998
KDE	373	14396	oct-1996

In the first part of this case study we show how CRs are distributed among authors. We show in particular that the distribution follows with a good accuracy the Lotka's law [3]. Similar behavior has been observed in author participation on open source projects [15]. We suggest an interpretation about this behavior and use this as a factor for analyzing how CRs has been assigned in a period of time. In the second part we show how much the CR assignment adopted in Mozilla and KDE follows the selection process proposed in this paper. In doing that we use the metrics of precision and recall [18], widely adopted to measure the performance of an information retrieval algorithm.

4.1 Lotka's law

In 1926, Alfred Lotka performed numerical analysis of the frequency distribution of scientific productivity [3]. Briefly stated, Lotkas Law stipulates that, as the frequency of publications increases, the approximate number of authors with that frequency of publications may be predicted. Lotka calculated that approximately 60.79% of authors would have only one publication. We believe that developers production within a project can be modeled with the Lotka's law.

We notice that also the assignment of CRs to a developer, in a period of time, follows, with a good accuracy, Lotka's Law. To show that, we used the Zipf's equation [20] numerically and conceptually equivalent Lotka's original formulation [4]: $r^n \cdot y = const$. Zipf's equation deals with the distribution of English terms among a collection of documents. In that case the number of term occurrence is intended as

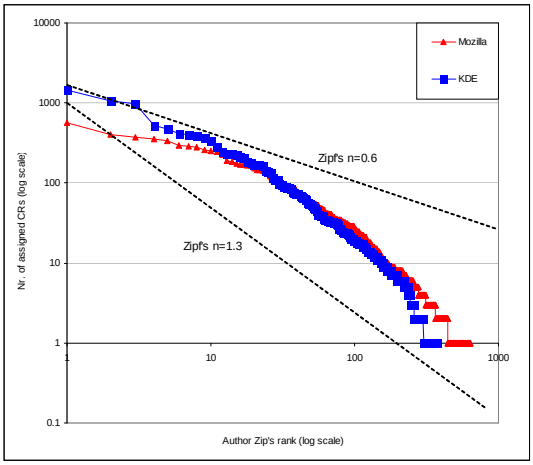

Figure 3: Assigned CRs distribution

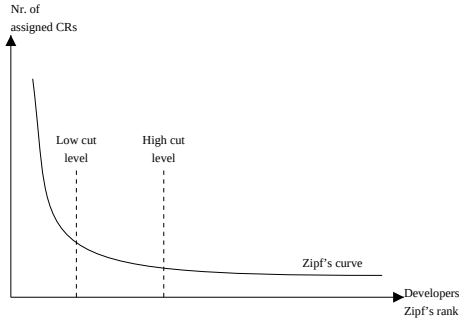

Figure 4: Low and high cut levels in Zipf's curve

the number of author publications in Lotka's Law. Similarly to Lotka, Zipf stated that in a document collection there are few terms with high occurrence and many terms with low occurrence. In Zipf's equation, r is the Zipf's rank, that is the position of the ith term (in our case an author) in the set of terms in decreasing order by its number of occurrence (in our case author set ordered by the number of assigned CRs); y is the number of term occurrence (in our case the number of assigned CRs); n and *const* are respectively the Zipf's exponent and a constant depending although on the particular collection considered. The number of term occurrence y (or published papers, or CRs assigned) may be predicted for a particular collection, although the predictive capability of these laws was debated [5]. Both Lotka's and Zipf's Laws are not intended to predict a particular authors productivity, or terms occurrence. Instead, they predicts aggregate behavior across a large number of authors/terms. For example, Zipf's Law is used to improve the indexing process of a document collection. In that case only the set of terms between two cut off levels in the Zipf's rank are used as index terms, because in has been shown that terms with high and low occurrence are non discriminant for a particular document [13]. This is the interpretation of the CRs assignment behavior we will refer.

4.2 Analysis of CRs data

Figure 3 shows, in logarithm scale, the CRs distribution among Mozilla and KDE developers. Developers are ordered by their number of assigned CRs, that is the Zipf's rank. The figure shows two Zipf's curves with exponent $n = 0.6$ and $n = 1.3$, they are approximatively the interval in which the Zipf's exponent of the real distribution is included. For KDE there are few developers (about 1%) for which a great number of CRs has been assigned (more than 500 CRs). Inversely there are many developers (about 58%) for which a small number of CRs has been assigned (less than 10 CRs). This happens also for Mozilla in which more than 300 CRs has been assigned to the 1% of developers, while less than 10 CRs has been assigned to the 72% of developers. This behavior can be interpreted in many ways. Here we attempt to give an interpretation based on the term significance of the homologue behavior of English terms following the Zipf's distribution. As for terms distribution two cut off levels can be fixed, a low cut level and a high cut level, as show in figure 4. These will subdivide developers in three categories:

1) developers belove the low cut level. These are developers for which a great number of CRs has been assigned. These are a small number and are usually *maintainers* of the project. Usually they don't resolve directly the CRs but integrate contributions and/or suggestions from volunteers not registered as developers. They also work on maintaining the consistency of Bugzilla database and are responsible of a wide area of the project.

2) developers between the low and the high cut levels. These are developers for which a reasonable number of CRs has been assigned. They can be considered as *regular* developers because they usually work on a localized and specific part of the project, also with regularity in time.

3) developers above the high cut level. These are developer for which few number of CRs have been assigned. They can be considered as *occasional* developers because they have provided sometimes volunteer support in resolving a specific CR.

Cut levels are not intended to be precise because boundary behavior can be considered fuzzy and levels may depend on the collection considered. We used this developer categorization to show how the assignment of CRs in a period of time follows the selection approach presented in this paper in each of the category. We show, in particular for KDE, that regular developers are those for which the assignment of CRs follows more our developer selection approach based on past CRs similarity. This is congruent with the consideration that regular developers deals usually with a localized and specific domain of software maintenance and evolution described by similar CRs text content.

4.3 CRs assignment in Mozilla and KDE

Precision and Recall are two metrics we used to measure how much the assignment of CRs among developers in the two case studies considered has followed the selection approach presented in this paper. In information retrieval evaluation, *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, while *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 [18]. Computation of precision and recall is performed at different cut levels [18] in order to take in account the position of the document in the ranked list. A cut level N (top N) is the list of the first N documents ranked by the retrieval algorithm. The ranked list is then

Table 2: KDE: top N recall averaged on the first m authors in Zipf's rank

first m authors	top N average recall					
in Zipf's rank	1	2	3	4	5	10
5	0.59	0.73	0.79	0.82	0.85	0.93
10	0.58	0.72	0.78	0.81	0.83	0.91
100	0.34	0.45	0.50	0.53	0.55	0.62
all	0.12	0.16	0.18	0.19	0.21	0.24

Table 3: Mozilla: top N recall averaged on the first m authors in Zipf's rank

first m authors	top N average recall					
in Zipf's rank	1	2	3	4	5	10
5	0.32	0.45	0.53	0.58	0.62	0.75
10	0.24	0.34	0.42	0.47	0.51	0.64
100	0.20	0.28	0.34	0.37	0.40	0.49
all	0.05	0.07	0.08	0.09	0.10	0.12

compared with the oracle that in our case is the developer assigned to the CR (the number is always one, because the bug tracking system assigns each CR to one developer, otherwise the CR is splitted in sub CRs each with one assigned developer). In every case, for the first cut level, the average precision and recall will coincide, while for cut levels greater that one, average precision will always decrease and average recall may increase. As precision for cut levels greater than one depends on recall (it is given by recall divided by N), we do not consider it in the results. For a given set of CRs a top N average recall of $X\%$ means that, X CRs on 100 has been assigned to one of the top N authors retrieved by the selection algorithm.

We have observed the CRs assignment of Mozilla and KDE from three point of view: (**O1**) how the assignment of CRs have followed the developer selection algorithm with respect of developer Zipf's rank; (**O2**) how the assignment of CRs have followed the developer selection algorithm in a period of time; (**O3**) how the developer selection algorithm depends on the author's index size.

O1. In the first point of view we will show how the assignment of CRs follows our developer selection approach with respect of developer categories pointed out by the Zipf's Law. We have built an index of developers by considering the first 50% of fixed CRs since 01-jan-2003, which approximately corresponds to mar-2004 for KDE and oct-2004 for Mozilla as KDE have a fixing rate greater than Mozilla even though less developers are involved. The second half of fixed CRs has been used as test set and the top N recall has been averaged on the first m authors in the Zipf's rank. Tables 2 and 3 show, respectively for KDE and Mozilla, the assignment strategy, in terms of top N recall, followed for the first $m = 5, 10, 100, all$ authors in Zipf's rank. Figure 5 shows the same behavior as a graph that is the cumulative top 1 recall average in author's Zipf's rank. The first 5 KDE developers in Zipf's rank are in 93% of cases in the top 10 set of developers retrieved by our selection approach, while in 59% of cases are in the top 1 set of developers retrieved. The first 5 Mozilla developers in Zipf's rank are in 75% of cases in the top 10 set of developers retrieved by our selection approach, while in 32% of cases are in the top 1 set of developers retrieved. At first sight, the results show that

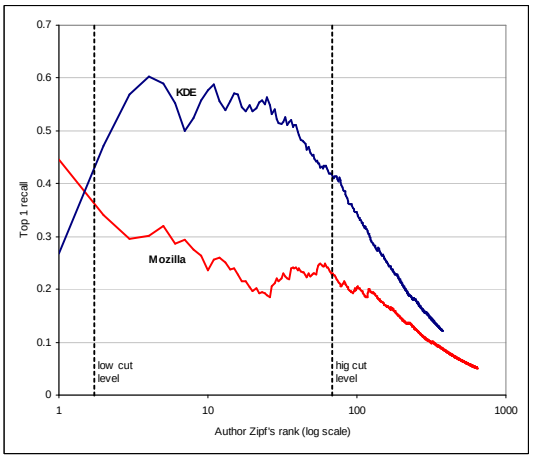

Figure 5: Top 1 recall: cumulative average on author Zipf's rank

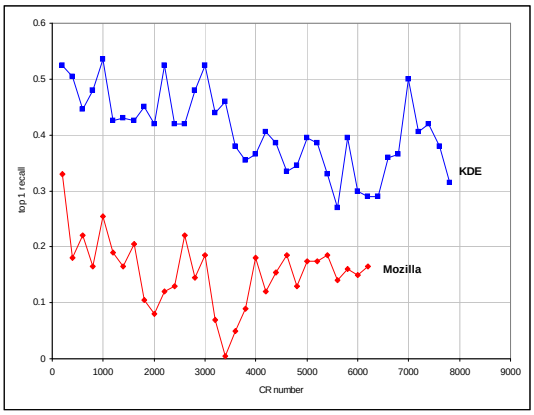

Figure 6: Assignement averaged on 200 fixed CRs

in KDE the strategy of CRs assignment follows more our developer selection approach than in Mozilla. As pointed out in figure 5 KDE confirms more the hypothesis that *regular* developers are those for which our selection approach explains better the actual data. A maximum around the first 4 developers can be observed in the graph. Otherwise for Mozilla the curve is almost decreasing and our selection approach is able to explain the data for the firsts developers, those we have classified as *maintainers* and *regular*.

O2. In the second point of view we have observed, by using the same previous index of developers, the assignment of the second half of fixed CRs by calculating the top 1 recall averaged progressively among a set of consecutive fixed CRs. In particular Figures 6 and 7 show for both Mozilla and KDE respectively the average among 200 fixed CRs and among the set of CRs fixed in a time interval of 60 days. Different values of this slice intervals has been chose but with similar results. It can be noticed that KDE adhere more to our selection approach than Mozilla. For KDE the top 1 recall is around 30% and 50%, instead for Mozilla it is around 10% and 20%.

O3. In the third point of view we show how the index of developers size impact the performance of the selection approach. It is expected that developers indexed with more

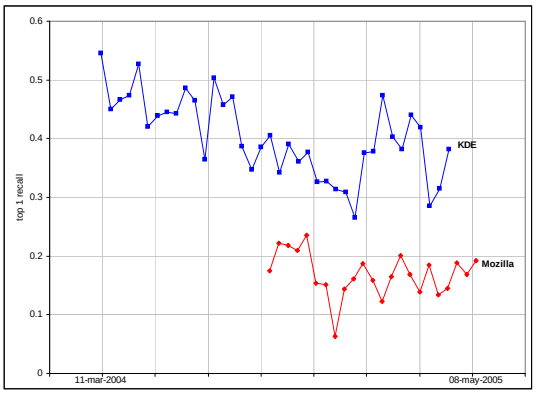

Figure 7: Assignement averaged on 60 days

Table 4: Top 1 recall averaged on the first m authors in Zipf's rank with different index size

first m authors	Mozilla		KDE	
in Zipf's rank	25%	50%	25%	50%
5	0.20	0.32	0.50	0.59
10	0.13	0.24	0.45	0.58
100	0.12	0.20	0.24	0.34
all	0.03	0.05	0.08	0.12

CRs are better described than developers indexed with less CRs. Then, with a fixed behavior tested on given index size, the prediction performance of that behavior should decrease if the index size decreases. This is shown in table 4 where the results of the first point of view (index size 50%) is compared with the results obtained with the same experiment but with an index of developer builded with the first 25% of CRs ordered by their fixed date. The results show a decrease of about 10 percent point for both systems.

5. CONCLUSIONS

Software and change repositories give new opportunities to analyze how software changes and evolves. In this paper an approach to select the best candidate set of developers able to resolve a given change request has been proposed. We show how this approach explains the actual data in a time slice of two years in two large open source projects, Mozilla and KDE. We believe that this approach can be useful for project managers when maintenance tasks have to be assigned to developers. Of course, to verify this assumption an empirical validation is needed. Our next objectives is to observe the assignment to different developers categories. For example, most vs less active authors, volunteers vs employed authors, project joined far vs early, intra vs extra project participation.

The method exploits the huge amount of historical data available for a mature software project. Its drawback is that for immature projects it will fail as pointed with the experiment of CRs assignment performed with different index of developers size. The drawback can be resolved by using data of other similar projects developers have participated. Furthermore, developer competence classification, is another direction of investigation we will take in consideration. In this way tracks left by developers in software repositories can be a valuable source of information to reconstruct their competence profiles.

6. REFERENCES

[1] Bugzilla. http://www.bugzilla.org/.
[2] Cvs. http://www.cvshome.org/.
[3] L. A.J. The frequency distribution of scientific productivity. *Journal of the Washington Academy of Sciences*, 16:317–323, 1926.
[4] A. Bookstein. Informetric distributions, part i: unified overview. *Journal of the American Society for Information Science*, 41(5):368–375, 1990.
[5] A. Bookstein. Implications of ambiguity for scientometric measurement. *Journal of the American Society for Information Science*, 52(1):74–79, 2001.
[6] 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.
[7] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.
[8] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM' 03*, Amsterdam, 2003.
[9] K. Fogel and M. Bar. *Open Source Development with CVS*. Coriolis, 2001.
[10] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.
[11] 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.
[12] T. Kistler and H. Marais. WebL — a programming language for the Web. *Computer Networks and ISDN Systems*, 30(1–7):259–270, Apr. 1998.
[13] H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2:159–165, 1958.
[14] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE' 00*, pages 167–176. ACM Press, 2000.
[15] G. B. Newby, J. Greenberg, and P. Jones. Open source software development and lotka's law: bibliometric patterns in programming. *J. Am. Soc. Inf. Sci. Technol.*, 54(2):169–178, 2003.
[16] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and D. Johnson. Terrier information retrieval platform. In *ECIR*, pages 517–519, 2005.
[17] M. F. Porter. An algorithm for suffix stripping. 1997.
[18] B. Ribeiro-neto and Baeza-yates. *Modern Information Retrieval*. Addison Wesley, 1999.
[19] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining revision history. *IEEE Trans. Softw. Eng.*, 30:574–586, Sept. 2004.
[20] G. K. Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philology*, 15:1–95, 1929.