Who Can Help Me with this Source Code Change?

Huzefa Kagdi[1], Maen Hammad[2], and Jonathan I. Maletic[2]

[1]*Department of Computer Science*
Missouri University of Science and Technology
Rolla Missouri 65409

[2]*Department of Computer Science*
Kent State University
Kent Ohio 44242

{hkagdi, mhammad, jmaletic}@cs.kent.edu

Abstract

An approach to recommend a ranked list of developers to assist in performing software changes to a particular file is presented. The ranking is based on change expertise, experience, and contributions of developers, as derived from the analysis of the previous commits involving the specific file in question. The commits are obtained from a software system's version control repositories (e.g., Subversion). The basic premise is that a developer who has substantially contributed changes to specific files in the past is likely to best assist for their current or future change. Evaluation of the approach on a number of open source systems such as koffice, Apache httpd, and GNU gcc is also presented. The results show that the accuracy of the correctly recommended developers is between 43% and 82%. New developers to a long-lived software project, or project managers, can use this approach to assist them in undertaking maintenance tasks, e.g., bug fix or adding a new feature. The approach can be realized as a plug-in to development environments such as Eclipse.

1. Introduction

It is a routine practice to seek advice from other developers, with more knowledge than oneself, when undertaking a maintenance task to a large software system. Among the, possibly hundreds of, developers on the project team a select few typically have deep knowledge about any one particular file or component. For a new team member just joining a project or a manager on a very large system, finding the best person to help out can sometimes be a tedious task. One typical solution is to email the project team (or a selection of the team) and ask for advice on who is the most knowledgeable about a given file.

This is particularly acute in collaborative open source projects where developers come and go from the team and no one project manager may exist. The knowledge of who to ask for advice leaves when a developer leaves. Even in long-lived development projects in industry or government there is a regular amount of developer turnover.

Fortunately, all this knowledge does not completely disappear when developers or managers leave a project. Version control systems keep an excellent record of who changed a file and when the change occurred. Here, we present an approach and tool, called *xFinder*, that recommends a ranked list of developers who are very likely to have good knowledge of the file(s) planned to be modified. This ranked list is obtained by mining the historical records found in the commits that are stored in software repositories of the project.

Our approach uses several heuristic criteria to infer developer expertise, change activity, and commit contributions in the context of a particular file. According to a recent study [5], the programmer activity (frequency and recent date) is a good indicator of the knowledge developers have in the code.

We evaluate our approach on a number of open source projects. For each project, we examine the set of commits for a given duration of time. We divide these commits into two portions. The first portion is the training-set and the second one is the evaluation-set. The training-set is mined for developer expertise and we see how well it can be used to recommend developers for the files that are changed in the commits in the evaluation-set. Results show that this is a viable and simple approach to recommending developers to assist in maintenance tasks.

The paper is organized as follows. Sections 2 and 3 detail our approach and the heuristics used, an example that demonstrates our prototype is in Section 4. The evaluation of the technique is given in Section 5 and Section 6 gives threats to validity. Related work is given in Section 7 and we end with conclusions and future work.

2. Contribution Measures

The basic premise of the approach is that the developers who contributed substantial changes to a specific part of source code in the past are likely to best assist for its current or future change. These past contributions are used to derive a mapping of the developers' expertise, knowledge, or ownership to particular entities of the source code - a *developer-code map*. Once a developer-code map is obtained, a list of developers who can assist in a given part of the source code can be obtained in a straightforward manner. Now, the question is where to find a historical account of source code changes and how to use them to form a developer-code map. Our approach uses the source code repository of a software system. More specifically, the commits in repositories that record source code changes, as submitted by developers, to the version-control systems (i.e., *Subversion*). The commits are used as both the historical archive of source code changes and to derive a developer-code map. Next, we describe the various aspects of commits in detail.

2.1. Subversion Commits

Source code repositories store metadata such as user-IDs, timestamps, and commit comments in addition to the source code artifacts and their differences across versions. This metadata explains the why, who, and when dimensions of a source code change. Modern source-control systems, such as *Subversion*, preserve the grouping of several changes in multiple files to a single change-set as performed by a committer. Version-number assignment and metadata are associated at the change-set level and recorded as a log entry.

Figure 1 shows a log entry from the *Subversion* repository of *kdelibs* (a part of *KDE* repository). A log entry corresponds to a single *commit* operation. This commit log information can be readily obtained by using the command–line client *svn log* and a number of APIs (e.g., *pysvn*). *Subversion*'s log entries include the dimensions *author*, *date*, and *paths* involved in a change-set. In this case, the changes in the files *khtml_part.cpp* and *loader.h* are committed together by the developer *kling* on the date/time *2005-07-25T17:46:20.434104Z.* The *revision* number *438663* is assigned to the entire change-set (and not to each file that is changed as is in the case with some version-control systems such as CVS). Additionally, a text message describing the change entered by the developer is also recorded. Note that the order in which the files appear in the log entry is not necessarily the order in which they were changed. Clearly, each commit stores the developer and the corresponding files changed.

A software system that is a long-lived would have gone through a numerous commits during its evolution. As can be seen from the above commit example, the relationship between a developer and the files in any given commit is one-to-one. However, a developer may contribute multiple commits with the same file. Also, multiple developers may change the same file in different commits. Therefore, commits give an opportunity to analyze for the exclusive and (the level of) shared contributions of developers to files. Next, we discuss, a few ways of gauging developer contributions from commits.

```
<?xml version="1.0" encoding="utf-8"?>
<log>
  <log entry revision="438663">
    <author>kling</author>
    <date>2005-07-
25T17:46:20.434104Z</date>
    <paths>
      <path
action="M">khtml_part.cpp</path>
      <path action="M">loader.h</path>
    </paths>
    <msg>
       Do pixmap notifications when
       running ad filters.
    </msg>
  </log entry>
</log>
```

Figure 1. Part of kdelibs subversion log message.

2.2. Commit Contribution

One measure of the developer contribution is the total number of commits, i.e., source code changes performed in the past. A developer who contributes a larger number of changes on specific parts of source code than some other developer can be considered as relatively more knowledgeable on those parts. We analyzed the commit contributions of the *koffice* developers and focused on the total number of commits performed by each developer on all the files. We considered commits performed in 304 days between 1/6/2006 and 31/3/2007. The total number of commits that have been extracted is 5642 and there were about 4991 different files in these commits. The number of developers that are involved during this time period is 60.

Table 1 shows the frequency distribution of developers according to their total number of commits, i.e., commit contribution. All the developers contributed more than one commit. Most of the developers have a small number of commits. For example, 24 developers contributed between 2 and 10 commits. The total number of commits of these 24 developers forms only 2% of the total commits in the considered period of *koffice*. On the other hand, we can see that one developer has more than 1040 commits and his commit contribution is about 19% of the total commits. Most of the commits are done by a very small number of developers. These results show that the number of developers contributing a substantially large portion of total commits (i.e., experts) in the open source projects is quite small. One should also note that many developers do not have significant commit contributions.

Table 1. Frequency distribution of developers over their commit contribution showing that only a small fraction of developers contribute a substantially large portion of total commits in a subset of *koffice* change history.

No. Of Commits	No. Of Developers	Commit Contribution
2 - 10	24	2%
11 - 20	7	2%
21 - 30	3	1%
31 - 40	4	2%
41 - 50	1	1%
51 – 60	1	1%
61 – 70	2	2%
71 – 80	2	3%
81 – 90	1	1%
91 – 100	1	2%
101 – 110	1	2%
121 – 130	1	2%
141 – 150	2	5%
161 – 170	1	3%
191 – 200	1	3%
251 – 260	1	4%
261 – 270	1	5%
271 – 280	2	10%
381 – 390	1	7%
521 – 530	1	9%
721 – 730	1	13%
1041 – 1050	1	19%
Total	60	100%

2.3. Developers and Files

The frequency and extent of changes to files committed by developers can be yet another measure of contribution. A developer who commits changes to files has (or acquires) of knowledge of these files. The more number of commits by a developer on a selected few files across a large number of commits could indicate frequent interactions and deeper knowledge of those files. On the other hand, a developer commit spanning across a large number of files may indicate that the developer has a wider knowledge of the system. So, there are two types of expertise that can be inferred: deep expertise and wide expertise. These two types of expertise are used in the approach to identify experts.

Another factor is the number of files that are updated exclusively by only a specific developer and no one else across a large number of commits. This could give an idea of the importance of a particular developer. For example, a developer who alone committed changes to 10 files over a long period of time most likely has more expertise concerning these files than anyone else.

Table 2 shows the frequency distribution of developers over the unique number of files they exclusively changed in a subset of the *koffice* change history. More than a half of the total developers contribute exclusively to quite a small number of files. This indicates that the importance of the majority of the developers and their expertise are restricted to a small part of the system. Only a few developers change a large number of files exclusively. This indicates the presence of developers whose impact and knowledge span across a substantial portion of the system.

Table 2. Frequency distribution of developers over the unique number of files they exclusively changed in a subset of *koffice* change history.

No. Of unique files	No. Of Developers
1-50	33
51-100	7
101-150	6
151- 200	2
201 - 250	1
251-300	3
401 - 450	2
451 - 500	1
501 - 550	2
551-600	1
950-1000	1
1001-1050	1
Total	60

2.4. Activity

Another consideration for developer contribution is the workdays, i.e., activity, involved in changes that are committed. The activity of a specific developer is the percentage of his or her workdays over the total workdays of the system. Here, a developer's workday is considered as a day on which (s)he submits at least one commit. A developer can submit multiple changes on a given workday. A system's workday is considered a day on which at least one commit is submitted. A day on which no commits are submitted is not considered a workday. Thus, the activity of a specific developer is his or her total workdays over the total number of the system's workdays.

The workday information can be easily obtained from the date component of the commit as seen in Section 2.1. For example, if we consider a period of 100 workdays from the lifetime of a system, and one developer committed five revisions in five different days during

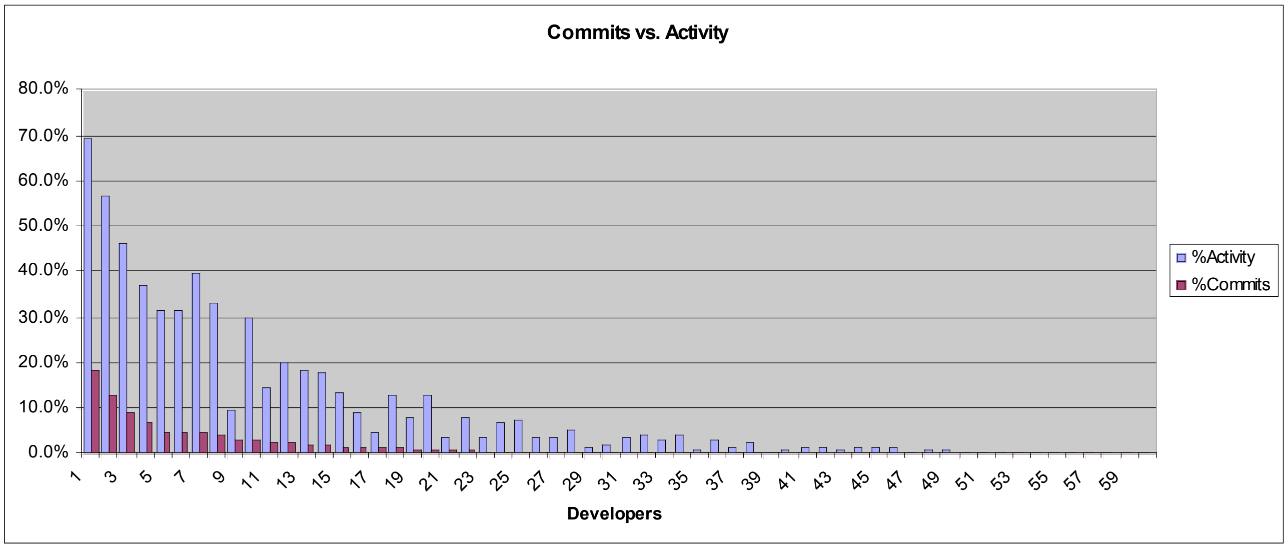


Figure 2. Frequency distributions of developers over activity and commit contributions in a subset of *koffice* development history. It shows that the most active developers also tend to have the most commit contributions

these 100 days, then his activity is 5%. In our analysis of the *koffice* project, we found that most of the developers have just a few workdays. The total number of workdays for *koffice,* in this case, is 304 days.
Table 3 shows the distribution of developers over the percentage of workdays (activity) and their total commit contributions. We can see that most of the developers have a small activity period. This observation is in a way similar to the analysis of the commit contributions. Only a small number of developers are active or have major commit contribution. As shown in Table 3, there are 37 developers whose work activity is less than 5% of the total workdays in the considered subset of *koffice* development history.

It is interesting to investigate the relationship between the activity and commit contribution of the developer. For instance, does the most active developer have the largest commit contribution? Figure 2 shows this relationship for a subset of the *koffice* project (also supplementary data in Table 1 and Table 3). As we can see, the most active developers (top three) also have the most commit contributions. In general the commit contribution and the activity of developers tend to be similar.

Now that we have described some of the ways of inferring developer contributions, we next describe how to use them to recommend developers to assist for a change in a source code file.

3. Developer Expertise and Recommendation

We use a combination of the three contribution measures to infer candidate developers who could best assist for a change in a given source code file. The contribution measures that are used are the commit contribution, the most recent activity date, and the number of active workdays. All these measures are obtained from the commits in the source code repositories of the system. We use these measures to determine developers that are likely to be experts in a specific source code file, i.e., developer–code map.

Table 3. Frequency distribution of the developers over the number of workdays they are active in a subset of *koffice* development period along with their commit contributions.

Developer Activity	No. Of Developers	Commit Contributions
1% - 5%	37	8%
6% - 10%	7	8%
11% - 15%	4	7%
16% - 20%	2	4%
21% - 25%	1	3%
26% - 30%	1	3%
31% - 35%	3	14%
36% - 40%	1	7%
41% - 45%	1	5%
46% - 50%	1	9%
56% - 60%	1	13%
66% - 70%	1	19%
Total	60	100%

The developer-code map is represented via the developer-code vector DV for the developer d and file f, as shown below,

$DV_{(d,f)} = <C_f, A_f, R_f>$, where:

- C_f is the number of commits, i.e., commit contributions that include the file *f* and are committed by the developer *d*.
- A_f is the number of workdays in the activity of the developer *d* with commits that include the file *f*.
- R_f is the most recent workday in the activity of the developer *d* with a commit that includes the file *f*.

Similarly, the change contributions to the file *F* can be represented via the file-change vector *FV*, as shown below,

$FV_{(f)} = <C'_f, A'_f, R'_f>$, where

- C'_f is the total number of commits, i.e., commit contributions, that include the file *f*.
- A'_f is the total number of workdays in the activity of all developers with commits that include the file $_f$.
- R'_f is the most recent workday with a commit that includes the file *f*.

The contribution or expertise factor, termed *Xfactor*, for the developer *d* and file *f* can be computed using a similarity measure of the developer-code vector and the file-change vector. Here, we use the Euclidean distance to find the distance between the two vectors. Distance is an opposite of similarity, this means that lesser the value of the Euclidean distance, greater the similarity between the vectors. The *Xfactor* can be given as follows,

$$Xfactor_{(d,v)} = \frac{1}{DV(d,f) - FV(f)}$$

$$Xfactor_{(d,v)} = \frac{1}{\sqrt{(C_f - C'_f)^2 + (A_f - A'_f)^2 + (R_f - R'_f)^2}}$$

We use the *Xfactor* as a basis of the recommendation method used to suggest a ranked list of developers to assist with a change in a given file. The developers are ranked based on their *Xfactor* values. The developer with the highest value is ranked first. Note that we discard all the developers for which the developer-code vector $DV_{(d, f)}$ for any given file is zero (case that would give an undefined *Xfactor* value). Of course, there maybe some files that have not been changed in a very long time, or this is the first change where a file is added. As a result, there will not be any recommendation. To overcome this problem, we look for developers who are experts in the package that contains the file, and recommend them instead. The package here means the immediate directory that contains the file.

We define the *package expert* as the one who updated the largest number of different (unique) files in a specific package. We feel the package experts are a reasonable choice and a developer with experience in several files of a specific package can most likely assist in updating a specific file in that package. As a final option, if no package expert can be identified, we turn to the idea of the system expert. The system means a collection of packages. It can be a subsystem, a module, or a big project (e.g. kspread, *koffice*, gcc …etc). The *system or project expert* is the person(s) who are involved in updating the largest number of different (unique) files in the system. The person who updated more files should have more knowledge about the system. In this way we move from the lowest, most specific expertise level (file) to the higher, broader levels of expertise (package then system). According to this approach, we guarantee that the tool always gives a recommendation (unless this is the very first file added to the system). The procedure for the suggested approach is given in Figure 3.

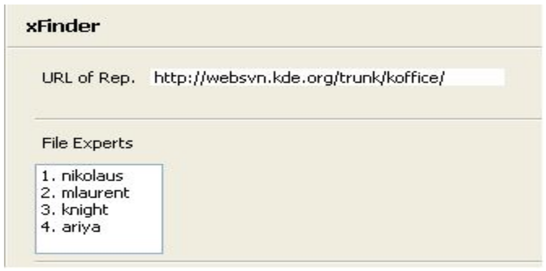
```
Recommender (f, p, maxFileExperts,
             maxPackageExperts, maxSysExperts, h)
Begin
// f: the file name
// p: the package name that contains f
// h:  the period of history

For each developer d appeared in h do
  Begin
    if Xfactor(f,d) > 0 then add d to fileList
    descendingSort(fileList) by the Xfactor values
    show fileList [1 …maxFileExperts]
  End for

If fileList.size( ) >= maxFileExperts, then Exit

For each developer d appeared in h do
  Begin
    //no. of files in package p updated by d
    if fileCount (p,d) > 0 then add d to packageList
    ascendingSort(packageList) by fileCount values
    show packageList [1 …maxPackageExperts]
  End for

If packageList.size( ) >= maxPackageExperts,
                                        then Exit
For each developer d appeared in h do
  Begin
    //no. of files in the whole system updated by d
    if fileCount (d) > 0 then add d to sysList
    ascendingSort(sysList) by fileCount values
    show sysList [1 …maxSysExperts]
  End for

End
```

Figure 3: The procedure to give a ranked list of developer candidates.

The three integer parameters; *maxFileExperts*, *maxPackageExperts*, and *maxSysExperts* are determined by the user of the tool. As we will see in the validation section, three seems to be a reasonable heuristic for both maxFileExperts and maxPackageExperts. Additionally, we suggest that the values of these parameters follow the property maxSysExperts >= maxPackageExperts >= maxFileExperts. To help understand the process we now present a detailed example/scenario of using our approach.

4. xFinder

We realized our approach in the form of a tool, namely *xFinder* (for expert finder). We now give a detailed example of using our approach and *xFinder* on part of the *koffice* open source system. Consider a situation where a developer needs help in updating the file *kspread/Canvas.cpp,* which is a file in the *koffice* project. Suppose that four is the maximum number of the recommended developers (maxFileExperts). *xFinder* will give the list of developers shown in Figure 4, which is part of the *xFinder* tool interface.


xFinder

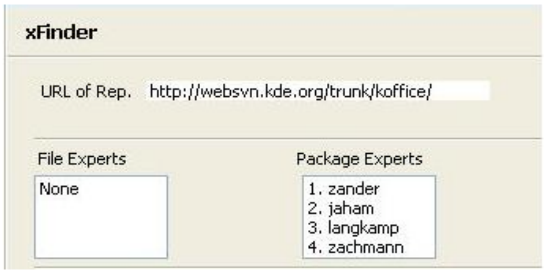
URL of Rep. http://websvn.kde.org/trunk/koffice/

File Experts

1. nikolaus
2. mlaurent
3. knight
4. ariya

Figure 4: The recommended developers generated by *xFinder* for the *kspread/Canvas.cpp* source file

The four developers IDs that appear in the "File Experts" list are the IDs of all developers who previously updated this file and are ranked according to their *Xfactor* in this file (*kspread/Canvas.cpp*). Only four IDs have been shown because the (maxFileExperts) parameter is set to four.


xFinder

URL of Rep. http://websvn.kde.org/trunk/koffice/

File Experts

None

Package Experts

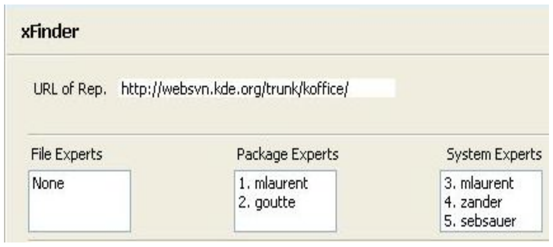
1. zander
2. jaham
3. langkamp
4. zachmann

Figure 5: The recommended developers generated by *xFinder* for the *commands/KWPageInsertCommand.h* header file

As another example, consider the header file *commands/KWPageInsertCommand.h,* which is also part of *koffice*. The recommendations for this file are shown in Figure 5.

Note that there are no file experts here. That is, there is no historical information for this file in the period of time specified by the user. Here, the user may have limited the mining period to one year previous from the current date. As the approach suggests, when there are no file experts or their number is less than a predefined value (maxFileExperts), the package experts are recommended. The package experts are ranked according to their experience in the package that contains the files. In the above example, the maximum number of the package experts (maxPackageExperts) is four.

One final example, if the file *kounavail/kounavail.h* is used as input to the tool. The three parameters maxFileExperts, maxPackageExperts, and maxSysExperts are all set to three. The tool generates the recommendation lists that are shown in Figure 6.


xFinder

URL of Rep. http://websvn.kde.org/trunk/koffice/

File Experts

None

Package Experts

1. mlaurent
2. goutte

System Experts

3. mlaurent
4. zander
5. sebsauer

Figure 6: The recommended developers generated by *xFinder* for the *kounavail/kounavail.h* header file

The system expert list appears here because the recommended package experts are less than three (the maxPackageExperts parameter). One important note here, the priority in the ranking is for file experts, then package experts, and finally the system experts. From the previous example, the top ranked developer in the third list has the rank order three.

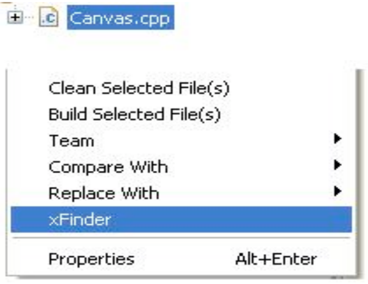
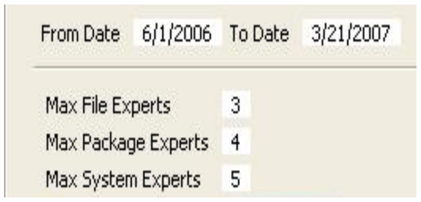


Figure 7: Invoking *xFinder* to find the experts in Canvas.pp source file

xFinder has been implemented using Java as a plug-in tool for *Eclipse* IDE. The tool can be invoked by clicking on the *xFinder* command that appears in the right click menu of the file (Figure 7). The tool has a set of input parameters (Figure 8) that need to be set by the user. These parameters are the mining history and the three integer parameters that control the sizes of the three expert lists.


From Date 6/1/2006 To Date 3/21/2007

Max File Experts 3

Max Package Experts 4

Max System Experts 5

Figure 8: Snapshot of *xFinder* input screen where its parameters are set by the user

5. Evaluation

The main goal of the evaluation is to assess the accuracy of our approach in correctly recommending the developers for a change in a given source code file. We use the source code version history as the data source for both training and evaluating our approach. The general methodology is to take a test-pair of file and developer, (*f*, *d*), from a commit, *c*. We trigger our approach to recommend a ranked list of developers for the file *f* by mining the commits that occurred before the commit *c*. If the developer *d* is in the recommended list, we consider that the pair *(f, d)* was correctly recommended. We repeat this process for a number of test pairs and report the accuracy in terms of percentage.

Since the (maximum) number of developers recommended can be more than one, we also evaluate the accuracy with regards to the ranked order at which the correct developer *d* from the test pair is recommended. This is an important factor, as there is a very little meaning in claming an accurate recommendation, if the correct developer is listed towards the bottom in a long litany of candidates suggested. In other words, one could simply say that the correct developer is one of all the developers who contributed to this project, and claim accuracy. We report the accuracy results for each individual rank (e.g., at rank 1, rank 2, and rank 3) in the recommended list.

Another goal is to compare the accuracies of recommendations made with individual component, i.e., commit contributions, activity, most recent workday, package expertise, and system expertise, and their combination that are used in our approach.

We use a subset of the version histories of eight open source projects, shown in Table 4, for the evaluation process. These projects were selected to include diversity in the sizes, programming languages, and application domains, development organizations, and processes. A set of commits is extracted from the version history of each project. The name of the file is in the format *package-name/file-name*.

The extracted set of commits is divided into two sets: a *training set* and a *testing set*. The testing set contains commits that occur at a later period than the commits in the training set. The number of commits in the training set is larger than the number of testing set. The training-set was used as the data source for our approach to generate a list of developers for the data in the testing set.

For each commit, we extracted the committer ID, the date of the change, and the names of all the files in this revision. From the commits in the test set, we extracted the files and the developers who updated these files (actual developers) and they are represented in the form of a test pair *(file name, committer ID)*. The file name is used as a help request. The recommendation given by the tool for that request is compared with the committer ID in the test record.

For example, we extracted the commits in a time period of about 10 months from the *koffice* project. The revision information of the first 297 days was used as a training set. There were 5565 commits performed during these days. The commits of the last 7 days are used to generate the test set. From the commits of these seven days, we gathered 272 test pairs for evaluation. For each record the file name is used as an input to the tool. The resulting list of recommended developers is compared

Table 4: Recommendation correctness over eight projects. The training set size is the number of commits and the test set size is the number of tested (file, developer) pairs.

System	Training Set (Commits)		Test Set (file, developer)		File Expert		Package Expert	System Expert	Total
	Size	Period	Size	Period	Rank 1	Rank 2-3	Rank 1-3	Rank 1-3	
kdelibs	5885	11 months	145	5 days	26%	14%	7%	1%	48%
kdenetwork	863	11 months	75	10 days	28%	4%	29%	0	61%
kdebase	5579	11 months	188	5 days	41%	7%	12%	1%	61%
kdemulimedia	473	11 months	52	1 month	23%	6%	14%	0	43%
Kdesdk	724	11 months	165	1 month	13%	6%	48%	0	67%
Apache- Httpd	535	14 months	57	1 month	23%	25%	17%	14%	79%
gcc	7236	2 years	106	14 days	27%	21%	10%	2%	60%
koffice	5565	10 months	272	7 days	52 %	11 %	18 %	1 %	82%

with the second element of the record (committer). If the recommended developer is the actual developer who appeared in the record, then the recommendation is considered correct. The rank of the actual developer within the recommendation list is also taken into account in the evaluation of the accuracy. Here, we set the maximum number of developers recommended parameter to three at file expertise, package expertise, and system expertise levels.

Table 4 shows the recommendation accuracy of the projects considered in our evaluation. In case of the *koffice* project, for 52% of the test records, the correct developer was the first one in the file expert level. In 11% of the test cases, the correct occurred at the recommended ranked second or third. In 18% of the test case, the correct developer was recommended as one among the top three ranked developers at the package expert level, i.e., there were no (correct or incorrect) recommendation at the file expertise level. Finally, if the correct developer did not appear at both the file expertise and package expertise levels, he was recommended among the top three at the system expert level in 1% of the test cases. In total, 82% of the test cases in *koffice* have an accurate recommendation of the developers at file, package, or system expertise level, and are always among the top three ranked candidates. The accuracies among the considered projects range from 43% to 82% with a number of them falling between 60% and 70%.

Table 5: Comparing the recommendation correctness of two measures for package experts

System	No. Of Updated Files [Rank 1-3]	Commit Contribution [Rank 1-3]
kdelibs	7%	6%
kdenetwork	29%	24%
kdebase	12%	12%
kdemulimedia	14%	10%
Kdesdk	48%	43%
Apache - Httpd	17%	14%
gcc	10%	11%
koffice	18 %	17%

Now, we examine the impact including the package and system expertise measures on the accuracy. Once again, the result in Table 4 show the accuracy is increased in all the projects with the inclusion of these measures. In case of some projects the accuracy is more than doubled. While most of the accuracy gain can be attributed to the package expertise measure, we found at least one project (*Apache httpd*) in which the system expertise measure contribute almost as much as the package expertise measure. This shows that including the package and system level expertise measures is worthwhile and improves accuracy substantially.

Another question here is why we used a seemingly orthogonal measure for package and system expertise contributions. We defined package expertise of a developer based on the number of unique files updated by that developer in a given package. A measure similar to commit contribution that is used for file expertise could have been used for package level expertise. That is, package expertise of a developer could be defined as the number of commits that are committed by the developer in a given package. Table 5 shows the results of the comparison of these two different definitions of package expertise on the same data sets as previously discussed. In seven out of eight projects, the definition of package expertise (Column 2) measure used in our approach outperforms the other (Column 3). In one project (*gcc*), the commit contribution measure of package expertise measure is slightly better than our definition of package expertise.

In our approach we use three different developer contributions (measures) to determine file experts. Now, we examine as to how using any of these three measures independently compare with the use of their combination. In order words, is any one of these measures as 'good' a parameter as, or better than, the combination used in our approach? To answer this question, we examined the same data sets by using these three measures independently. We are concerned about examining the file level experts (file expert list), because a different measure is used to determine package experts (number of unique files updated).

In two out of eight projects (*kdemultimedia* and *Apache httpd*), the accuracy obtained via the developer-code vector, i.e., *Xfactor*, is better than, any other measure. In only one project (*gcc*) the developer-code vector does not give the best result. In the remaining five projects the accuracy of the developer-code vector is good as the maximum accuracy of the three measures. Thus, in seven out of eight projects the developer-code vector gives at least the maximum accuracy of the three other measures. These results show that the combination of the measures, i.e., developer-code vector is possibly a better recommender. Even if the accuracy may not increase in some cases, we can be sure, with high confidence, that we got the maximum accuracy among the three individual measures.

6. Threats to Validity

There are some threats that may affect the validity of the accuracy of results. We used different history periods to get the training set. Obviously, it is difficult to use one fixed period across different projects due to the variation in their different evolution aspects. For example, some projects have more activity in one month than another does in one year. As a result the sizes of the data sets also vary. In the validation process we used the committer ID which represents the developer's identity. We do not know exactly who changed the file, but only who committed from the repository data. Also, if the developer has more than one ID [12], the accuracy of the result will be affected. The validation has been applied on a subset of only eight projects. Another area of investigation is the use of other similarity measures for vectors (e.g. *Cosine* or *Manhattan* distance) for computing the *XFactor* values and their impact on the recommendation accuracy. Finally, all the projects that we used for testing are open source projects. We do not claim that the results presented here would hold equally on other types of projects (e.g. closed source).

Our tool needs version history in order to give a recommendation. If there is not a "good" portion of development history, our tool will most likely not be able to function with a high accuracy (or in the worst case not provide any recommendation at all). The accuracy of the recommended list seems to improve with an increase in the training set size, however this has certain limits. This could be attributed to the fact that when open source projects evolve, their communities also evolve [11], so the relationship between the length of the historical period of time and the accuracy of the recommendation is not very decisive. This is an interesting issue of future investigation.

7. Related Work

McDonald and Ackerman [8] developed a heuristic based recommendation system called the Expertise Recommender (ER) to identify experts at the module level. Developers are ranked according to the most recent modification date. When there are multiple modules, people who touched all the modules are considered. Vector based similarity are also used to identify technical support. For each request three query vectors are created; symptoms, customers, modules. This vector is then compared with the person's profile. This approach depends on user profiles that need to be additionally and explicitly collected upfront. This approach has been designed for the specific organizations and not tested on open source projects.

Mino and Murphy [9] produced a tool called Emergent Expertise Locator (EEL). Their work is adopted from a framework to compute the coordination requirements between developers given by Cataldo et al. [4]. EEL helps in finding the developers who can assist in solving a particular problem. The approach is based on mining the history of how files have changed together and who has participated in the change. In our approach we also include the activity, i.e., workdays, of the developers and identify experts at the package and system levels and not only at the file level.

Expertise Browser (ExB) [10] is another tool to locate people with a desired expertise. The elementary unit of experience is the Experience Atom (EA). Experience is measured by the number of these EAs in a specific domain. The smallest EA is the code change that has been made on a specific file. In our approach, the number of EAs corresponds to the commit contribution. Again we included more than one parameter in deterring file experts. We also used two different measures to identify experts: one measure for file experts and another for package and system experts.

Anvik and Murphy [2] did an empirical evaluation of two approaches to locate expertise. As developers work on a specific part of the software, they accumulate expertise. They term this expertise *implementation expertise*. The two approaches are based on mining the source and bug repositories. The first approach examines the check-in logs for the modules that contain the fixed source files. Recently active developers who did the changed are selected and filtered. In the second approach, the bug reports from bug repositories are examined. The developers are selected from the CC lists, the comments, and who fixed the bug. They found that both approaches have relative strengths in different ways. In their first approach, the most recent activity date is used to select developers. This study focuses on identify experts to fix bugs or to deal with bug reports. Our approach uses only source code repositories and no other repositories (e.g., bug repositories).

A machine learning technique has been used in [1] to automatically assign a bug report to the right developer who can resolve it. The classifier obtained the machine learning technique analyzes the textual contents of the report and recommends a list of developers. Another text-based approach is used in [13] to build a graph model called ExpertiseNet for expertise modeling. Our approach uses expertise measures that are computed in a straightforward manner from the commits in source code repositories (and does not employ a machine learning like technique).

There are also works on using MSR techniques to study and analyze developer contributions. German [7] described in his report some characteristics of the development team of PostgreSQL. He found that in the last years only two persons have been responsible for most of the source code. Tsunoda et al. [14] analyzed the developers' working time of open source software. The email sent time was used to identify developers' working time. Bird et al. [3] mined email archives to analyze the communication and co-ordination activities of the participants. Yu and Ramaswamy [16] mined CVS repositories to identify developer roles (core and associate). The interaction between authors is used as clustering criteria. The KLOC and number of revisions are used to study the development effort for the two groups. Weissgerber et al. [15] analyze and visualize the check-in information for open source projects. The visualization shows the relationship between the lifetime of the project and the number of files and the number of files updated by each author. German [6] studied the modification records (MRs) of CVS logs to visualize who are the people who tend to modify certain files.

8. Conclusions and Future Work

We presented an approach to identify expertise in open source projects. The approach recommends a ranked-list of experts in a specific source file. The analyzed contribution includes commit contribution, activity, recent activity date, and the number of files updated. This combination of contribution measures can be computed efficiently by only examining commit logs and yet still achieves high accuracy. This differs from previous approaches to this problem that examine a wide range of artifacts from a number of different repositories or employ more computationally expensive techniques.

We validate the approach using a subset of eight open source projects. The validation shows that the approach gives a reasonable accuracy results and it can be a base for or part of other expertise identification tools.

In future, we plan to evaluate the approach on more open source projects with very large historical information. We also plan to extend the approach to identify experts in syntactic entities (e.g., class and method). So we will have then lower levels of expertise. We will also investigate other contribution/expertise measures to include in the approach (e.g., the interaction between developers).

9. References

[1] Anvik, J., Hiew, L., and Murphy, G. C., "Who Should Fix This Bug?" in Proceedings of 28th international conference on Software engineering (ICSE '06), Shanghai, China, 20-28 May 2006, pp. 361 - 370.

[2] Anvik, J. and Murphy, G., "Determining Implementation Expertise from Bug Reports", in Proceedings of Fourth International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, May 20-26 2007.

[3] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A., "Mining Email Social Networks", in Proceedings of 2006 International Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23 2006, pp. 137-43.

[4] Cataldo, M., Wagstrom, P., Herbsleb, J., and Carley, K. M., "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools", in Proceedings of 20th anniversary conference on Computer supported cooperative work (CSCW'06), Alberta, Canada, 2006, pp. 353 - 362.

[5] Fritz, T., Murphy, G., and Hill, E., "Does a Programmer's Activity Indicate Knowledge of Code?" in Proceedings of 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07), Cavtat, Croatia, September 3-7 2007, pp. 341 - 350.

[6] German, D. M., "An Empirical Study of Fine-grained Software Modifications", Empirical Software Engineering, vol. 11, no. 3, September 2006, pp. 369-393.

[7] German, D. M., "A Study of the Contributors of PostgreSQL", in Proceedings of 2006 International Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23 2006, pp. 163 - 164.

[8] McDonald, D. and Ackerman, M., "Expertise Recommender: A Flexible Recommendation System and Architecture", in Proceedings of 2000 ACM Conference on Computer Supported Cooperative Work (CSCW '00), Philadelphia, PA, December 2-6, 2000, pp. 231-240.

[9] Minto, S. and Murphy, G., "Recommending Emergent Teams", in Proceedings of Fourth International Workshop on Mining Software Repositories (MSR '07), Minneapolis, MN, May 20-26 2007.

[10] Mockus, A. and Herbsleb, J., "Expertise Browser: a Quantitative Approach to Identifying Expertise", in Proceedings of 24th International Conference on Software Engineering (ICSE '02), Orlando, FL, May 19-25 2002, pp. 503-512.

[11] Nakakoji, K., Yamamoto, Y., Nishinaka, Y., Kishida, K., and Ye, Y., "Evolution Patterns of Open-Source Software Systems and Communities", in Proceedings of International Workshop on Principles of Software Evolution (IWPSE'02), Orlando, Florida, 2002, pp. 76-85.

[12] Robles, G. and Gonzalez-Barahona, J. M., "Developer Identification Methods for Integrated Data From Various Sources", in Proceedings of 2005 international workshop on Mining software repositories (MSR'05), St. Louis, Missouri, 2005, pp. 1-5.

[13] Song, X., Tseng, B., Lin, C., and Sun, M., "ExpertiseNet: Relational and Evolutionary Expert Modeling", in Proceedings of 10th International Conference on User Modeling (UM'5), Edinburgh, UK, Jul. 24-29 2005.

[14] Tsunoda, M., Monden, A., Kakimoto, T., Kamei, Y., and Matsumoto, K.-i., "Analyzing OSS Developers' Working Time Using Mailing Lists Archives", in Proceedings of 2006 International Workshop on Mining Software Repositories (MSR '06), Shanghai, China, May 22-23 2006, pp. 181 - 182.

[15] Weissgerber, P., Pohl, M., and Burch, M., "Visual Data Mining in Software Archives to Detect How Developers Work Together", in Proceedings of Fourth International Workshop on Mining Software Repositories (MSR'07), Minneapolis, USA, May 20-26 2007.

[16] Yu, L. and Ramaswamy, S., "Mining CVS Repositories to Understand Open-Source Project Developer Roles", in Proceedings of Fourth International Workshop on Mining Software Repositories (MSR'07), Minneapolis, USA, May 20-26 2007.