Supporting Software Maintenance by Mining Software Update Records

Jelber Sayyad Shirabad
School of Information
Technology and Engineering
University of Ottawa
Ottawa, Ontario
K1N 6N5 Canada
jsayyad@site.uottawa.ca

Timothy C. Lethbridge
School of Information
Technology and Engineering
University of Ottawa
Ottawa, Ontario
K1N 6N5 Canada
tcl@site.uottawa.ca

Stan Matwin
School of Information
Technology and Engineering
University of Ottawa
Ottawa, Ontario
K1N 6N5 Canada
stan@site.uottawa.ca

Abstract

This paper describes the application of inductive methods to data extracted from both source code and software maintenance records. We would like to extract relations that indicate which files in, a legacy system, are relevant to each other in the context of program maintenance. We call these relations Maintenance Relevance Relations. Such a relation could reveal existing complex interconnections among files in the system, which may in turn be useful in comprehending them. We discuss the methodology we employed to extract and evaluate the relations. We also point out some of the problems we encountered and our solutions for them. Finally, we present some of the results that we have obtained.

Keywords

Software maintenance, reverse engineering, data mining, knowledge based systems

1. Introduction

Legacy systems are old systems which still need to be maintained [32]. Despite the fact that these systems were developed many years ago, they are still essential in controlling some of the most important aspects of modern society. According to Ulrich, in 1990, 120 billion lines of code were in existence, most of which were considered to be legacy software systems at the time [34]. The Y2K problem brought the issue of software maintenance to the attention of a wider audience; some estimates of fixing this problem were in the trillions of dollars.

There is an obvious need to conduct research which directly or indirectly benefits the software maintenance programmer, since progress in this area has the potential to reduce the overall cost of software, sustaining its quality or reducing its degradation rate over its service life time. As for the challenge imposed to potential researchers, software maintenance has been referred to as a severe problem [22], or even as being the most difficult of all aspects of software production [9, 31]. The need for further research in the area is even more apparent in the light of statistics suggesting papers related to software maintenance are less than 1% of the papers annually published on software topics[16].

We use the following definitions, based on the work of Chikofski and Cross [4]:

Reverse Engineering is the process of analyzing a system to:

- identify its components and the relations among them;
- create an alternative representation for the system at the same or higher level of abstraction.

Redocumentation, a sub-field of reverse engineering, involves providing alternative views of a system at relatively the same abstraction level. For example data flow, or control flow diagrams of a system.

Design recovery, another major area of reverse engineering, uses domain knowledge, external information, and deductive and fuzzy reasoning to identify higher-level abstractions than the ones that can be obtained by examining the system directly.

In the past 5 years, different researchers have proposed the incorporation of knowledge based techniques into reverse engineering tools [2, 6,15]. Such systems in effect will fall into the intersection of reverse engineering and Knowledge Based Software Engineering (KBSE).

1.1 Knowledge based vs. inductive methods applied to software engineering

In a broad sense, *Knowledge Based Software Engineering* (KBSE) is the application of artificial intelligence technology to software engineering [22]. Most KBSE systems *explicitly* encode the knowledge that they employ [22]. KBSE systems designed for assisting software engineers in low-level everyday maintenance tasks have the potential to represent and deduce the relations among components of a software system. This is usually done at the expense of requiring a fairly extensive body of knowledge and employing deductions and other algorithms that are sometimes computationally demanding. In other words most such systems are fairly *knowledge rich.*

Very often, KBSE systems employ expert systems or knowledge bases as their underlying technology. Some examples of these systems include [5, 9, 13, 20, 21, 26, 27, 29, 35, 36, 38].

Some of the problems that systems employing large knowledge bases face are:

- the great effort needed to build and maintain a knowledge base;
- the shortage of trained knowledge engineers to interview experts and capture their knowledge;
- the time consuming nature of the process which typically leads to lengthy development;
- the need for continued development of the knowledge base if the system is to be maintained in routine use and at a high level of performance.

In contrast to the considerable body of work which has applied *deductive* methods to different aspects of software engineering, *inductive* methods have been applied far less frequently in this domain.

Learning systems have the potential to outperform an expert by finding *new* relations among concepts. They do this by examining examples of successfully solved cases. This could allow the incorporation of knowledge into a system without the need for a knowledge engineer [37]. The quality of a model created for a particular domain may further improve, should there be an effective way to incorporate the knowledge of domain experts when creating the model.

Examples of the application of inductive or data mining methods, in software engineering include [1, 7, 8, 22, 25, 28, 33].

In this paper we discuss our research that applies inductive methods to aid software maintainers in some of their daily activities. We believe that software maintenance can benefit from data mining research, and the intersection of the two fields is an area that deserves more attention from researchers in both communities.

2. Learning from past maintenance activities

Most average to large sized companies, that produce software, keep a record of changes applied to the source code as the system is maintained over many years. This, in effect, is a repository of past software maintenance experience. Depending on the degree of detail kept, there is a considerable amount of knowledge hidden within these records that reflects the connections and interactions between different parts of the software system.

Knowledge of existing relations between components of the system is in most cases the key to proper maintenance. As the size and complexity of a system grows, so does the likelihood that maintainers will lack knowledge of the relations among different components.

As mentioned earlier, legacy software systems are both large and old; these factors both contribute to the difficulty of understanding such systems, particularly for software engineers who are new to the project.

Source code is the most common starting point, and often the only starting point, for learning about an existing system. Such learning, or *comprehension*, of programs is a prerequisite for being able to properly apply changes required during software maintenance.

One of the important questions asked by a maintenance programmer, while looking at a piece of code, such as a routine or even a file, is:

"Which other files or routines should I know about, i.e. what else might be relevant to this piece of code?"

To answer this question, the programmer needs to be aware of some of the existing relations among components of the system.

In our research, we apply inductive methods to historical maintenance update records, and to a variety of other data sources. We will therefore use inductive techniques to extract from the past experience what we call a *maintenance relevance relation* among objects in the system under consideration. In machine learning terms, this relation is commonly known as the 'concept' that we are learning.

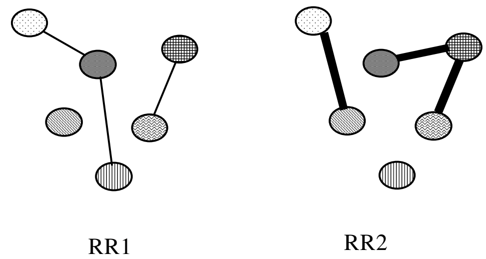

Figure 1. Two different Relevance relations RR1 and RR2 among software components of a system

In general, a *relevance relation* maps a set of objects into a value which shows how relevant these objects are to each other. As suggested by Figure 1, different relevance relations may relate different components of the system, depending on the definition of the relation. A *maintenance relevance relation* (MRR) is a special kind of relevance relation, which exists and is learned in the context of software maintenance. More specifically, we are interested in a special kind of MRR which is a function that takes two files, f1 and f2, and returns true or false indicating whether the two files are relevant to each other. Expressed another way, if a software engineer needs to understand f1, he or she will probably also need to understand f2.

Learning a relevance relation, such as an MRR, can be seen as a design recovery process. The hope is that the extracted concept will inform and assist the maintenance programmer with understanding existing complex relations or interactions within the target system. Such relations may reveal the intrinsic interconnections between different component of the subject system which:

- may or may not be documented
- are the consequences of the original design and implementation decisions, or a side effect of changes that the system has gone through over its life time

As discussed in [30] we have applied inductive methods to the data extracted from:

- Static source code analysis
- Historical maintenance records
- Logs generated by tools used by software engineers performing maintenance tasks (which implicitly contains expert knowledge)

In [30] we presented an overview of our research, including steps involved in creating our MRR, and some of the results we had obtained. However, there was a need to improve the *precision* and *recall* values of our relevance predictions. The current paper reviews the essential aspects of our research (the reader can follow this paper without reading out previous work). It then discusses some of the techniques we have used to improve our past results. In particular we have considerably improved precision and recall by using a better noise reduction technique.

In the next section we summarize some of the important aspects of the process of creating an MRR.

3. Learning a Maintenance Relevance Relation

The following are the steps involved in our approach to learn an MRR:

1. Instrument the TkSee program to log user interaction with the system. TkSee is a system that helps software maintainers explore source code [18, 19].

2. Learn the concept of Relevance based on the logged user interactions, plus information regarding changes to source files during the software maintenance process. The latter information is obtained from a configuration management system.

3. Evaluate the usefulness of the learned concept

4. While the level of usefulness is not high enough

    5. Update the attribute set used in defining the relation, and log other necessary information if required

    6. Learn the concept as mentioned in step 2

    7. Evaluate the usefulness of the learned concept

This procedure accommodates the inclusion of software engineers' expertise into the process of learning, by including information from the log of his or her interactions with the TkSee tool. In our experience, the level of usefulness of such data depends on the ability to connect the logged information to a well-defined maintenance task, such as solving a defined problem. As will be discussed below, we have chosen the action of selecting two files in a user work session as the basis for one of our relevance assignment heuristics.

The problem of learning a particular relevance relation among pair of files can be seen as a *classification* problem. Each pair of files used in the learning process, represents an instance, or *example*, of the concept. A set of *training* examples are used by an induction algorithm to generate a *classifier*. Since we are using a supervised machine learning method, we need to define appropriate *classes*, or *categories* of relevance between files, and assign one of these categories to each training example. The generated classifier can be used to classify future unseen instances of the concept i.e.; suggest a category of relevance between *any* two given files.

We used C5.0, a decision tree inducer by RuleQuest Research, to generate our classifiers. Decision tree learning algorithms have been widely studied by machine learning researchers and learners such as C4.5[1] have been successfully used in other projects. The training examples, which are the input to this program, are represented as a vector of the values of a set of predefined *attributes* or *features* describing the concept. For instance, the length of the common file name prefix for two files could be an attribute. Other attributes include the number of global variables shared by the two files, and the number of routines they both call. The complete set of attributes is shown in Table 1. As mentioned above, each training example is also assigned a class of relevance.

The output of C5.0 is a *decision tree* that has a condition at each node. The condition is a test of the value of one of the predefined attributes. At each leaf, there is a prediction for the category, or the class, of an example that satisfies the conditions on the path from the root of the tree to that particular leaf.

One of the benefits of using decision tree learners is that they produce *explainable* classifiers. In other words, a human can study the classifier e.g.; a decision tree, to gain further insight about the reason that it is suggesting a particular class for an example. In our case, such explainable models have the potential to present the more important relations among files in the system, which in turn could be used to assist software engineers in gaining a better understanding of the target software system. Such an understanding could potentially be useful to both reverse engineering and also to the day to day maintenance of the software system.

To summarize, we would like to be able to predict the relevance of any pairs of files in the system to each other. To do this, based on our past experience, we select candidate file pairs for training, and assign to each pair a class of relevance; e.g. *Releavnt* or *Not Relevant*. The concept of file pair relevance is described in terms of a set of features or attributes. For each training example, we calculate the vector of values corresponding to these attributes. These classified training examples are input to a program that outputs a decision tree classifier. Given the vector of attribute values for any pairs of files in the system, the generated classifier can be used to predict their class of relevance.

Attribute Name	Attribute Type
Number of shared files included	Integer
File 1 directly includes File 2	Boolean
File 1 is directly included by File 2	Boolean
Number of shared directly referred types	Integer
Number of shared directly referred non-type data items	Integer
Number of routines *Directly Referred* in File1 /*Defined* in File2	Integer
Number of routines FSRRG *Referred in File1*/*Defined* in File2	Integer
Number of routines *Defined* in File 1/*Directly Referred* in File2	Integer
Number of routines Defined in File 1/FSRRG *Referred* in File2	Integer
Number of shared routines directly referred	Integer
Number of shared routines among all routines referred	Integer
Number of nodes shared by FSRRGs	Integer
Common prefix length in a file name	Integer
The files have the same name (but different extensions)	Boolean
File Extension of File 1	Text
File Extension of the File 2	Text
The files have the same extension	Boolean

Table 1:Summary of attributes used in our experiments

When computing the values of attributes, one of the files in each pair is our main interest, and the other file is the one that may be Relevant (or Not Relevant, etc.) to the file that we are interested in. In Table 1, the file of interest is referred to as *File 1*, and the other file is referred to as *File 2*. The FSRRG (File Static Routine Reference Graph) mentioned in the table is the static call graph for a file, which includes all the routines referred to in that file, as well as, recursively, the FSRRG for these routines. A node in such a graph is the name of a routine.

The relevance categories with which we have experimented are:

- Relevant: The files were both *modified* in the same ‘update’ of the system. "An update is the basic unit of identifying actual (as opposed to planned) updates to the software." [3] An update associates problem

[1] C4.5 is the predecessor of C5.0

(reports) with a list of files being modified in response to those problems, allowing us to find out the set of files that were changed to address each problem.

- Potentially Relevant: The files were both *looked at* in the same TkSee session, but were not both modified, which is the case for Relevant category. This class allows us to take advantage of the expertise of software engineers who may have recognized a relationship between two files, even though both files did not need to be updated.

- Not Relevant: During the period of time covered by our data, two files were neither considered relevant nor potentially relevant to each other.

Each of the training examples (regarding a pair of files) is assigned one of these classes. After performing machine learning, the resulting classifier can be given an arbitrary pair of files, plus the computed values of the required attributes. It will use these to induce one of the above relevance categories.

In addition to working with the three classes, we also experimented with learning a two-class relevance concept that involves only the *Not Relevant* and *Relevant* categories as defined above.

The above heuristics allow us to automatically assign a class of relevance to each pair of files in the training examples. The particular MRR learned here defines the conditions under which two files will, may, and will not be updated together. In principle, our approach is not limited to generating only one kind of MRR. For example, given enough time and resources, the training examples could be manually classified by expert software engineers, based on the knowledge they have accumulated by maintaining the system over many years. The same learning strategy could then be applied to this manually classified data. It is very likely that such a relation will have some similarity with our MRR, as well as some differences.

3.1 Class conflicts among the examples

Class or category conflict, also known as *class noise*, may occur either when for some subset (of size $K > 1$) of the training set, the attribute values are the same, while the classes are different.

Such noise should be resolved if learning algorithms are to perform effectively. There are various strategies for resolving the noise. In [30] the strategy we used was as follows. For each set of examples with indistinguishable attributes, replace the set with N examples that have the relevance class that was predominant in the set. As a consequence of this, however, the machine learning program would be presented with N identical examples (i.e. N examples that have the same attributes and the same class). For the results reported in [30] we used N=1.

In the next section, we will describe a different method of resolving conflicts caused by noise.

4. Experiments

In this section we briefly provide background on the legacy software system that is the target of our research. We will discuss the data used in our learning process. Then we will present a new method that improves upon the results presented in [30].

4.1 Background on the target software system and learning data

SX 2000 is a large legacy telephone switching system.[2]. It was originally created in 1983 by Mitel corporation. The software is written in multiple programming languages; it contains approximately 1.9 million lines of code, distributed in more than 4700 files. These files are of different types including source, interface, and type declaration files

The class label assignments discussed in the previous section were applied to the data corresponding to the dates shown in Table 2

For files to be considered relevant to each other we chose updates in which between 2 and 20 files were modified. Limiting the set of updates to this range ensures that there is a strong relevance relationship among the members. Furthermore, the type of the first file in each pair was limited to the major programming language used in this system.

The set of Not Relevant pairs was created by subtracting the set of Relevant pairs in the desired time period from the set of all possible file pairings. The final set is created by applying any additional filtering required e.g.; removing pairs referring to files that have been subsequently deleted or renamed.

Class	Time Period
Relevant	1997/01/01-1999/12/16
Not Relevant	1995/01/01-2000/02/03
Potentially Relevant	1998/06-1999/02

Table 2: Time periods from which data was gathered

[2] Also know as a PBX

The distribution of each of the classes after resolving the label conflicts introduced by labeling heuristics is shown in Table 3.

Relevant	5,705	0.0008%
Not Relevant	6950612	99.999%
Potentially Relevant	1120	0.0002%

Table 3. Distribution of the classes of relevance of file pairs

4.2 Experiment setup

As can be seen in Table 3, the classes of file pairs are highly unbalanced. The more interesting cases only account for merely 0.001% of the examples.

Unbalanced data sets appear frequently in real world machine learning problems, and impose difficulties that are documented by other researchers [10, 12, 17].

To address the imbalance we extracted two groups of relatively balanced datasets from our full data set. The number of Relevant and Not Relevant examples in each of the datasets generated was the same. Datasets in both groups included all the examples of the Potentially Relevant class. In one group all the Relevant pairs were included in each dataset, and in another group the examples of the three classes were distributed in a 40-20-40 proportion, in which Potentially Relevant examples accounted for 20% of the data. In this paper, we refer to the first group of datasets, i.e. those using all examples of Relevant class, as **All-Relevant** and the second group is referred to as **40-20-40**. The Not Relevant pairs were sampled from the set of all Not Relevant pairs and were chosen in a way that the first file in each pair is the same as the first file in some Relevant pair. We allowed about 10% of Not Relevant examples to be chosen randomly, without regard to the existence of a Relevant pair with the same first file name in the pair.

We attempted to learn both a 2-class and 3-class version of an MRR. For the two-class problem, we generated the data sets by removing the Potentially Relevant pairs from the corresponding three-class data sets. Our experiments used two groups of balanced files, as discussed earlier. We generated 30 dataset for each of **All-Relevant** and **40-20-40** groups.

We applied 10 fold cross validation for each data set. In k-fold cross validation, each dataset is divided into k parts. The classifier is trained using the data from k – 1 parts, and it is tested on the data in the single remaining part which was not used during training. This process is repeated k times, so that each example is used at least once in testing the classifier. We measured the macro averaged[3] precision and recall of the prediction for each class, along with the average decision tree size and error, and the standard error of these values.

Precision is a standard metric from information retrieval, and used in machine learning, that tells one the proportion of returned results that are in fact valid (i.e. assigned the correct class). *Recall* is a complementary metric that tells one the proportion of valid results that are in fact found.

4.3 Results

Tables 4 and 5 show the initial results obtained for the two groups of datasets, without removing the class noise introduced after computing the attribute values. The tables are divided into two main parts. The upper part shows the class distribution and macro averaged precision and recall measures along with the average size of a data set for each of the **All Relevant** and **40-20-40** groups. The precision of the prediction of Relevant class, which is an important factor in our evaluation, is shown in bold. The bottom part of each table shows the average size and error rate of the generated decision trees and their corresponding standard error

The following observation can be made from these two tables:

- The error rate of the decision trees obtained is much better than a baseline system that randomly assigns the relevance class to examples with the same probability for each class. Such a system would have a 66.6% error rate for the 3-class problem, and an error rate of 50% for the 2-class problem.
- The 2-class problem generates better results than the 3-class problem.
- The **All Relevant** data sets generate better results for most measures, except for the precision and recall of the Potentially Relevant class.
- **40-20-40** data sets generate smaller trees.
- In the case of the 2-class problem, there is not much advantage gained by using **All Relevant** files in measures other than tree size. The average size of the generated trees for **All Relevant** datasets is 2.3 times the size of the generated trees for the **40-20-40** data sets

[3] This means that we accumulated the counts of correct and incorrect classifications in all test datasets, and produced the final precision and recall value based on these accumulated values.

The results also show that the obtained trees are fairly large, and that there is more to be done to improve the precision and recall measures.

The approach we took to remove noise is very close to the idea of *Robust-C4.5* [14]. In Robust-C4.5, after creation and pruning of a decision tree, examples in the training dataset that were not correctly classified (outliers) are removed from the training set, and a new tree is built using the new training dataset.

This process is repeated until no example in the training set is misclassified. Our approach is similar, except that instead of repeating the process until there are *no* misclassified training instances, we stop when successive iterations yield negligible improvement.

Tables 6 and 7 show the results at the end of the fourth iteration, for the two and three class problems, respectively. The entries for precision, recall, tree error and size also include the changes in the values of these measures compared to the corresponding values shown in Tables 4 and 5.

We stopped in the fourth run, because the amount of improvement from one run to the next drops to the third digit after decimal point.

The results of Tables 6 and 7 show that by using the outlier removal approach mentioned above, we could create classifiers that are equally good for the two and three class problems. This is in contrast to our findings in [30], which showed that a better result could be obtained for the two-class problem. In [30] the class noise is removed by replacing each set of conflicting training examples with one example. The attribute values for this replacement example is the same as the rest of the examples in the conflicting set, and its class is the same as the majority class in the set.

The differences between corresponding values in the different parts of Tables 6 and 7 are very small. For both problems, it can be seen that the classifiers generated using the **40-20-40** group of data sets generate trees which are smaller than the ones generated from the **All-Relevant** data sets. Note that the average size of a data set in the **40-20-40** group is much smaller than the average size of a dataset in **All-Relevant** group.

Table 6 shows a sharp decrease in the proportion of the Potentially Relevant class after first four repetitions. This could be indicative of a higher degree of noise in the examples of this class. The minimal constraint imposed when we assign pairs of files to the Potentially Relevant class is that two files have been browsed in the same session. It is possible that a programmer looks at two files for completely independent reasons. It is also possible that, after studying two files, the programmer comes to the conclusion that the files are Not Relevant to each other. Without knowing the reason for a programmer's action, it is very likely that files that are Not Relevant to each other would be mistakenly paired as Potentially Relevant. Unfortunately, recording the reasons for a programmer's action in a transparent manner and without influencing his or her regular work pattern is a very difficult task.

It is worth noting that the uniform improvement in the values of precision and recall in Tables 6 and 7 was accompanied by a major reduction in the average size of the trees generated.

4.4 A note on discretizing numeric attributes

As can be seen in Table 1, the majority of attributes used in our experiments are numeric, some of them ranging between 0 and 2000. Usually such attributes result in large and complex decision trees. We prefer smaller trees because it makes the task of interpreting the generated model easier for humans. Smaller trees also speed up the process of classification once the classifier is actually deployed in the field.

By *descretizing*, or *grouping*, a numeric attribute a range of its values is replaced by a single value. In [30], we used a discretization method called *Entropy-based* grouping [11], which proved to be very effective in the reduction of the size of the learned decision trees. The class noise removal method employed in [30], which was discussed above, also has the property that it reduces the size of the learned tree. However, as opposed to our current method, the previous method did not rely on a particular concept learning algorithm such as. C5.0.

Replacing a range of values for an attribute with a single value has the potential to introduce class noise among examples that were unique, and therefore not conflicting. Consequently, a class noise removal operation should be performed on any discretized data set. In the past we experimented with the following two combinations of class noise removal and attribute value discretization (grouping):

1. Group/Resolve

2. Resolve/Group/Resolve

Our experiments showed that the second combination generated a better error rate and smaller decision trees. Comparing our current result and the corresponding result from [30] reveals that while the trees generated in our previous method are relatively smaller than the ones reported in this paper, our previous values for precision and recall measures are much lower, especially for the Not Relevant and Potentially Relevant classes. In our opinion the precision, recall, and error rates obtained by employing the ideas from the robust C4.5 method, as discussed in this

	All Relevant			40-20-40		
Class	***Proportion***	***Precision***	***Recall***	***Proportion***	***Precision***	***Recall***
notRelevant	45.5%	0.721	0.842	40.0%	0.657	0.824
potRelevant	9.0%	0.496	0.233	20.0%	0.547	0.372
relevant	45.5%	**0.761**	0.712	40.0%	**0.683**	0.618
Avg data set size	12530			5600		
	Tree Size	Error		Tree Size	Error	
Mean	498.8	**27.1%**		343.5	**34.9%**	
SE	1.8	0.1%		1.3	0.1%	

Table 4 Results before removing outliers (3 class)

	All Relevant			40-20-40		
Class	***Proportion***	***Precision***	***Recall***	***Proportion***	***Precision***	***Recall***
notRelevant	50.0%	0.763	0.852	50.0%	0.747	0.845
relevant	50.5%	**0.832**	0.736	50.0%	**0.822**	0.713
Avg. data set size	11410			4480		
	Tree Size	Error		Tree Size	Error	
Mean	236.5	**20.6%**		103.4	**22.1%**	
SE	1.3	0.1%		1.0	0.1%	

Table 5 Results before removing outliers (2 class)

	All Relevant			40-20-40		
Class	***Proportion***	***Precision***	***Recall***	***Proportion***	***Precision***	***Recall***
notRelevant	64.3±**0.2**%	0.997+**.276**	0.997+**.155**	70.9±**0.6**%	0.998 +**.341**	0.997+**.173**
potRelevant	0.6±**0.0**%	0.974+**.478**	1.000+**.767**	2.1±**0.0**%	0.968 +**.421**	0.996+**.624**
relevant	35.1±**0.2**%	**0.994**+**.233**	0.995+**.283**	27.0±**0.6**%	**0.996**+**.313**	0.996+**.378**
Data Size	7736±**44.4**			2315.1±**33.9**		
	Tree Size	Error		Tree Size	Error	
Mean	60.5	**0.4%** **-26.7**		20.1	**0.3%** **-34.6**	
SE	0.3	0.0%		0.1	0.0%	
Change in Tree size	**-88%**			**-94%**		

Table 6 Results Obtained After Round 4 of outlier removal (3 class)

	All Relevant			40-20-40		
Class	***Proportion***	***Precision***	***Recall***	***Proportion***	***Precision***	***Recall***
notRelevant	61.8±**0.9**%	0.997 +**.234**	0.991 +**.139**	63.8±**0.4**%	0.996 +**.249**	0.997+**.152**
relevant	38.2±**0.9**%	**0.986**+**.154**	0.996+**.260**	36.2±**0.4**%	**0.995**+**.173**	0.992+**.279**
Data Size	8174±**193**			3082±**34**		
	Tree Size	Error		Tree Size	Error	
Mean	58.7	**0.7%** **-19.9**		21.2	**0.5%** **-21.6**	
SE	0.4	0.1%		0.3	0.0%	
Change in Tree size	**-75%**			**-79%**		

Table 7 Results Obtained After Round 4 of outlier removal (2 class)

paper, outweighs the slightly smaller trees generated in our previous work.

5. Conclusion and future work

The work presented here is the continuation of our research towards learning a useful and accurate *Maintenance Relevance Relation*. We have been able to increase the precision and recall measures for all classes while producing reasonably small trees. This has been accomplished largely by changing our method for handling noise. In future, we intend to take into account different theoretical evaluation methods that could better reflect the usefulness of the produced results.

Obviously the extent of what can be learned from past software maintenance activities depends on the existence and the nature of the data recorded. In our research, we can learn relations that predict whether two files may or may not get updated together because our industrial partner records the name of each file changed by an update. The granularity of the stored data, is another factor that limits what can or can not be learned. It should also be noted that, while most medium and large size companies tend to save and track changes to software, because this is not done with data mining in mind, many times there is need for fair amount of programming and data processing to transform the existing data to a form that can be used by a learning algorithm.

To gain a better understanding of the usefulness of the generated classifiers, we plan to deploy our improved classifier at Mitel, and collect and analyze the data on its usage and benefits to software engineers at work.

Acknowledgements

This work is sponsored by CSER and supported by Mitel and NSERC. The Authors would like to thank the members of SX 2000 team for their support and help with our project.

References

[1] M.A. Almeida and S. Matwin. Machine Learning Method for Software Quality Model Building. *Proceedings of the Eleventh International Symposium on Methodologies for Intelligent Systems (ISMIS)*, 565-573, Warsaw, Poland, 1999.

[2] B. Bellay and H. Gall. An Evaluation of Reverse Engineering Tools. *Technical Report TUV-1841-96-01*, Distributed Systems Group, Technical University of Vienna 1996.

[3] C. Bryson and J. Kielstra. *SMS - Library System User's Reference Manual.* DT.49, Version A18, Mitel Corporation, 1998.

[4] E.J. Chikofski and J.H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software, vol. 7 no. 1*, 13-17, January 1990.

[5] D. Chin and A. Quilici. DECODE: A Cooperative Program Understanding Environment. *Journal of Software Maintenance, vol. 8 no.1*, 3-34. 1996.

[6] R. Clayton, S. Rugaber, and L. Wills. On the Knowledge Required to Understand a Program. *Proceedings of the Forth Working Conference on Reverse Engineering*. Honolulu, Howaii, 69-78, October 1998.

[7] W. Cohen and P. Devanbu. A Comparative Study of Inductive Logic Programming Methods for Software Fault Prediction. *Proceedings of the Fourteenth International Conference on Machine Learning*, Vanderbilt University, Nashville, TN, USA, July 8-12 1997.

[8] W. Cohen and P. Devanbu. Automatically Exploring Hypotheses about Fault Prediction: a Comparative Study of Inductive Logic Programming Methods. *International Journal of Software Engineering and Knowledge Engineering,* 1999.

[9] P. Devanbu and B.W. Ballard. LaSSIE: A Knowledge-Based Software Information System. *Automated Software Design*. Edited by M.R. Lowry and R.D. McCartney, AAAI Press, 25-38, 1991.

[10] K.J. Ezawa M. Singh, and S.W. Norton. Learning Goal Oriented Bayesian Networks for Telecommunications Management. *Proceedings of the Thirteenth International Conference on Machine Learning*, 139-147, 1996.

[11] U.M. Fayyad and B.K. Irani. Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, 1022-1027, 1993.

[12] T. Fawcett and F. Provost. Combining Data Mining and Machine Learning for Effective User Profile. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, OR USA. 8-13, 1996.

[13] M.T. Harandi and J.Q. Ning. Knowledge-Based Program Analysis. *IEEE Software, vol. 7 no. 1*, 74-81, January 1990

[14] G.H. John. Robust Decision Trees: Removing Outliers from Databases, *Proceedings of the first international Conference on Knowledge Discovery and Data Mining*, 174-17, 1995.

[15] K.A. Kontogiannis and P.G. Selfridge. Workshop Report: The Two-day Workshop on Research Issues in the Intersection between Software Engineering and Artificial Intelligence (held in conjunction with ICSE-16). *Automated Software Engineering v. 2*, 87-97, 1995.

[16] O.C. Kwon, C. Boldyreff, and M. Munro. Survey on a Software Maintenance Support Environment, *Technical Report 2/98*, Centre for Software Maintenance, School of Engineering and Applied Science, University of Durham, 1998.

[17] M. Kubat, R. Holte and S. Matwin. Learning When Negative Examples Abound. *Proceedings of the Ninth European Conference on Machine Learning,* Prague, 1997.

[18] T.C. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case

Study. *Technical Report TR-97-07*, Department of Computer Science, University of Ottawa, 1997.

[19] T.C. Lethbridge. Integrated Personal Work Management in the TkSee Software Exploration Tool. *Second International Symposium on Constructing Software Engineering Tools CoSET*, 31-38, June 2000.

[20] Z.Y. Liu, M. Ballantyne, and L. Seward. An Assistant for Re-Engineering Legacy Systems. *Proceedings of the Sixth Innovative Applications of Artificial Intelligence Conference*. AAAI, Seattle, WA, 95-102, 1994.

[21] Z.Y. Liu. Automating Software Evolution, *International Journal of Software Engineering and Knowledge Engineering*, *v. 5 no 1*, March 1995.

[22] M. Lowry and R. Duran. Knowledge-based Software Engineering. *The Handbook of Artificial Intelligence v. 4,* Edited by A. Barr, P. Cohen and E.A. Feigenbaum. Addison-Wesley Publishing Company, 1989.

[23] R. McCartney. Knowledge-Based Software Engineering: Where We Are and Where We Are Going. *Automated Software Design*. Edited by M.R. Lowry and R.D. McCartney, AAAI Press, 1991.

[24] E. Merlo and R. De Mori. Artificial Neural Networks for Source Code Information Analysis. *Proceedings of International Conference on Artificial Neural Networks*, *v.2 part 3*, Sorrento, Italy, 895-900, May 1994.

[25] C. Montes de Oca and D.L. Carver. Identification of Data Cohesive Subsystems Using Data Mining Techniques. *Proceedings of International Conference on Software Maintenance ICSM*, Bethesda, Maryland, 16-23, November 1998.

[26] M. Moore and S. Rugaber. Using Knowledge Representation to Understand Interactive Systems. *Proceedings of the Fifth International Workshop on Program Comprehension*, Dearborn, MI, 60-67, May 1997.

[27] S. Palthepu, J.E. Greer and G.I. McCalla. Cliché Recognition in Legacy Software: A Scalable, Knowledge-Based Approach. *Proceedings of the Forth Working Conference on Reverse Engineering*, Amsterdam, The Netherlands, 94-103, October 1997.

[28] A. Porter. Using Measurement-Driven Modeling to Provide Empirical Feedback to Software Developers. *Journal of Systems and Software, v. 20 no. 3*, 237-254, 1994.

[29] C. Rich and R.C. Waters. *The Programmer's Apprentice*. ACM Press. 1990.

[30] J. Sayyad Shirabad, T.C. Lethbridge, and S. Matwin. Supporting Maintenance of Legacy Software with Data Mining Techniques. *Proceedings of CASCON 2000*, 137-151, November 2000.

[31] S.R. Schach. *Software Engineering*. Richard D. Irwin, Inc, and Aksen Associates Inc., 1993.

[32] I. Sommerville. *Software Engineering*. Addison-Wesley, 1996.

[33] K. Srinivasan and D. Fisher. Machine Learning Approaches to Estimating Software Development Effort. *IEEE Transactions on Software Engineering. vol. 21 no. 2*, 126-137, February 1995.

[34] W.M. Ulrich. The Evolutionary Growth of Software Reengineering and the Decade Ahead. *American Programmer*, v. 3 no 10, 14-20, 1990.

[35] S. Walczak. ISLA: an Intelligent Assistant for Diagnosing Semantic Errors in Lisp Code. *Proceedings of the Fifth Florida Artificial Intelligence Research Symposium*, 102-105, 1992.

[36] M. Ward, F.W. Calliss, and M. Munro. The Use of Transformations in "The Maintainer's Assistant". *Technical Report 9/88*, Centre for Software Maintenance, School of Engineering and Applied Science, University of Durham, 1988.

[37] S.M. Weiss and C.A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufmann Publishers Inc., 1991.

[38] N. Wilde, S.W. Dietrich and F.W. Calliss. Designing Knowledge-Base Tools for Program Comprehension: A Comparison of EDATS & IMCA. *Technical Report SERC-TR-79-F*, Software Engineering Research Center, University of Florida, CSE-301, Gainesville, FL 32611, December 1995.