Applying Data Mining to Software Maintenance Records

Jelber Sayyad Shirabad, Timothy C. Lethbridge and Stan Matwin
School of Information Technology and Engineering
University of Ottawa, Ottawa, Ontario, K1N 6N5 Canada
{jsayyad, tcl, stan}@site.uottawa.ca

Abstract

In a system maintained over a long time period, as is the case for legacy software, there will be many unknown and non-trivial relationships among components. Finding such hidden relationships may help software engineers in their maintenance activities. In this paper we present an approach whereby we mine software update records to find relationships between files that are changed together. The generalized models we present as results are obtained by using features extracted from different sources of knowledge such as source code and problem reports. The predictive quality of some of the generated models suggest that they can be deployed to be used in a real world setting. The paper also includes the results of analyzing the structure of some of the best models obtained.

1. Introduction

Understanding a software system is key to its proper maintenance. However, more often than not, knowledge about the system and how to proceed with maintenance is scarce. This is certainly the case for legacy software systems that tend to be very large and complex. These systems also suffer from other factors such as high staff turnover, poor documentation and the long periods of time that they must be maintained. In such systems it is not surprising to find non-trivial relationships among different components of the system that were not previously known.

This lack of knowledge also contributes to delays, cost overruns, and other complications such as introduction of new errors during maintenance.

Research in reverse engineering and program comprehension has resulted in many tools and techniques to help software engineers recover lost knowledge and make latent knowledge explicit so software engineers may better understand software systems.

Discovering and understanding non-trivial relationships plays an important role in overall understanding of a given system. As suggested by other researchers [2, 5], using Artificial Intelligence (AI) techniques such as employing knowledge bases can further benefit reverse engineering and program comprehension Using large knowledge bases introduces various difficulties, however, some of whom are mentioned in an earlier paper by the authors[11].

In this paper we discuss the application of an alternative AI method to software maintenance, namely the inductive or *machine learning* methods.

Unlike knowledge based systems, inductive systems do not require a large body of knowledge. Instead, from past experience, they learn *models*, also known as *concepts*, that can be used in future predictions.

Data mining and inductive methods in general have been successfully applied to a variety of application domains. However, they have been largely neglected by the software engineering community and particularly in research involving source code level software maintenance.

In this paper we will present an application of inductive methods in the context of the following software maintenance issue:
When a software engineer (SE) is looking at a piece of code in a file F one of the questions he or she needs to answer is "*are there other files that may be affected by the changes applied to F?*" In other words "*are there other files in the system that he or she needs to be aware of in the process*

of maintaining the code in the file F?" Perhaps this problem can better be appreciated if one considers that a large legacy system can include thousands of files.

Our approach to help a software engineer to answer this question is to find relations among files tuples of which, when present, suggest that the files will both need to change together. We refer to this particular class of relations as *co-update* relations. We refer to these and similar relations in general as *Maintenance Relevance Relations* (MRR), since we are interested in the relevance of files to each other during software maintenance. We find an MRR by learning a classifier. The measure of relevance is the result of classification. The nature of relevance is defined by the particular relation, in our case the relevance of two files to each other in terms of them being updated together.

Problem tracking systems record problems or bugs reported against a software system. They also typically maintain a record of changes applied to the software to fix problems. Such systems can be seen as repositories of system maintenance experience. One can use classification learning and other data mining techniques to extract or learn relations such as co-update relations which are hidden in such repositories.

As data for our studies, we used maintenance records from a large legacy telephone switch system, i.e. a PBX, developed by Mitel Networks. This system is written in a high level programming language (HLL) and assembler. The version of the PBX software which we used in our experiments had close to 1.9 million lines of code distributed among more than 4700 files, out of which about 75% were HLL source files.

In the following sections we describe how we formulated and implemented the learning tasks, and discuss the results.

2. Casting the problem as classification

The machine learning techniques used in our experiments fall under the category of *classification learning*. This is a so called supervised learning method, because we learn from an existing pre-classified set of examples of a concept (what we want to learn). This set is referred to as *training set*. The classifier that is the result of the learning process is also known as a *model*. To learn the concept we apply an *induction algorithm* to the training set. To evaluate the generated model we test it on a *testing set*, which consists of a set of examples independent from those used in training. Each training example is a *known* case of the concept we want to learn, in that it is assigned an *outcome* or a *class*. An example is described in terms of the observed values for a set of *features* or *attributes*. Once a model or classifier is learned from a set of training examples it can be used to predict the class or outcome of unclassified or unsolved examples.

A co-update relation exists among pairs of files. We can cast the task of learning this relation as a classification problem where we classify two files as one of m distinct classes that show how likely it is that two files will change together. This approach results in a discrete classifier. The generated model or classifier is the co-update relation that we are trying to learn. As an alternative to a discrete classifier one can learn a classifier that returns a numeric value as the likelihood of two files being changed together

In this paper we will present a discrete co-update relation with two classes called Relevant and Not-Relevant. If two examples are classified as Relevant it means that a change in one will likely result in a change of the other. The Not-Relevant class indicates that a change in one file will likely not have an effect on the other e.g. when two files belong to two independent subsystems.

We used C5.0, a decision tree induction system by RuleQuest Research, to learn the co-update relation[1].

As shown in Figure 1, a decision tree has a condition at each node. The condition is a test T_i of the value of one of the predefined attributes used to describe the example. At each leaf, there is a prediction for the class C_j of an example that satisfies the conditions on the path from the root of the tree to that particular leaf. In other words each such path provides a conjunction of tests performed on the value of some of the features

[1] We have also experimented with a deliberately trivial learning algorithm called 1R and another approach called Set Covering Machines [7]. As expected, 1R did not produce satisfactory results due to the complexity of the data. Results obtained for SCM were not significantly better than the ones obtained with C5.0.

and the corresponding outcome if all the tests on the path result in a true value.

We have chosen decision tree learning because the algorithms are among the most-established techniques in machine learning and data mining. They produce explainable models that can be presented to software engineers. This is an important factor since software engineers can actually find the reason for the outcome of a prediction by following the path in the tree that is taken by the example or case. In Section 4 we provide an example of such analysis.

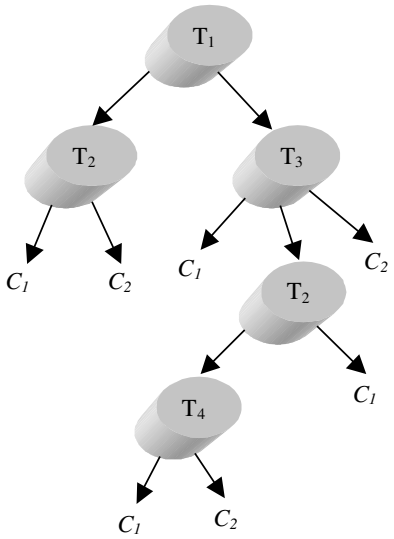

Figure 1. A decision tree

2.1. Training and testing repositories

As we mentioned earlier, to be able to learn a model or concept, we first need to find examples of it. Since the co-update relation is between pairs of files, we first need to find a set of file pairs where each pair is either labeled as Relevant or Not Relevant. Once we have a file pair and its corresponding relevance label we can create an example by calculating the value for a set of predefined features and assigning the example a class value similar to the file pair label.

At Mitel Networks a problem report is stored in a system called SMS (Software Management System). SMS provides a range of facilities including bug tracking and source code management. Problem reports typically include a textual description of the problem with possible additional information such as hardware status when the problem was observed, etc. If the problem causes change in the source code, an update report (simply referred to as an *update*) will be submitted and stored in SMS. An update records the problem reports that it intends to address and the name of the files changed as the result of applying the update. If the submitted update passes the testing and verification process, the update is considered as final or *closed*.

Using heuristics, we automatically extracted from SMS the following two kinds of examples of the co-update relation:

- Relevant: Two files that have been changed in the same update are Relevant to each other
- Not Relevant: During the period of time covered by our data, two files were not updated together.

To apply these heuristics, first we select a time period $T=[T_1\text{-}T_2]$ of certain number of years from which we want to learn. We find all the updates that were 'closed' in this time period. For each update, we find all the files changed by that update and create a list of *Relevant* file pairs. We refer to this heuristic as the *co-update heuristic*. Any other file pairs that are not labeled as Relevant by the co-update heuristic in a time period $T'=[T_3\text{-}T_2]$ where $T_3 \leq T_1$, are labeled as *Not Relevant*. We refer to this second heuristic as the *Nor-Relevant heuristic*. Due to complementary nature of these heuristics, the number of Not-Relevant file pairs is reduced as the number of Relevant file pairs increases. In other words we obtain the Not-Relevant labels by applying what is known in AI as the *Closed World Assumption*.

In principle, widening T' (increasing the time-period covered) results in a higher degree of confidence regarding two files not being changed together. This is because the evidence to the contrary does not exist in a larger time period. On the other hand, choosing a very large time period T' means applying the co-update heuristic to that time period, which can result in the inclusion of changes applied to the system in a distant past that may no longer reflect the current state of the system. In the experiments discussed in this paper we have used a time period T'=T or in other words $T_3 = T_1$.

We created the set of Relevant and Not Relevant file pairs by processing the updates for the 1995-1999 time period. There were 1213 problems reports in this time period for which we found 1401 closed updates. An update can change

more than one file. We define the *update group size* as the number of files changed by it. To apply the co-update heuristic we need updates with a group size greater than or equal to two. Out of all closed updates in the above time period, 43% changed two or more files. However it turned out that some updates change a large number of files. For instance we encountered updates that changed more than 280 files. It makes sense to assume that updates that change a smaller number of files capture relations that exist between files that are more closely related. Further analysis of the updates showed that 40% of them have a group size between 2 and 20 inclusively. These updates constitute 93% of updates with a group size larger than 1. In the experiments reported in this paper we have limited the update group size to 20.

Our version of the co-update relation will only focus on the files in a particular high level language (HLL). This is because our experience with the Assembler parsers available to us showed that the information generated by them was not as accurate as we had wished. This can be mostly traced back to the less structured nature of Assembler code. The number of Relevant HLL file pairs changed by updates with a group size limit of up to 20 in the 1995-1999 time period was 4547. This number includes repeated file pairs if two files were changed together in more than one update.

For N source files we can create $\binom{N}{2}$ pairs of files. This formula takes into account the symmetric nature of the co-update relation and the fact that we do not pair a file with itself. To create the set of Not-Relevant pairs we need to create a *set of potential file pairs* and then from it remove the set of Relevant file pairs (SR). Using the above formula, the number of potential file pairs when all the HLL source files in the system are used is close to six million pairs. This immense number of file pairs or examples brings to light a major difficulty with learning the co-update relation, namely the very skewed distribution of classes.

To reduce the number of potential file pairs we have constrained the first file in a potential file pair to be one of the first files in some Relevant file pair in SR. We refer to this approach as the *Relevant Based* method of creating Not-Relevant pairs. By applying the Relevant Based method we are pairing each file f_i with files that are Relevant to it and files that are Not-Relevant to it. Using this method, we paired the first file f_i of Relevant file pairs $(f_i, f_j) \in$ SR with other HLL files to create the set of potential file pairs. By removing SR from this set i.e. applying the closed world assumption, we create the *default Not-Relevant file pairs set*. We can further refine this set by removing file pairs that from other sources of information we know should not be labeled as Not-Relevant. An example of this is the case where a software engineer may suggest that some of the Not-Relevant files created by the Not-Relevant heuristic should be removed from the default Not-Relevant file pairs set. In our case we used the data from SMS for this purpose. We removed all the file pairs that were changed together by updates in the 1995-1999 time period. This set of file pairs of course includes all the file pairs in SR which was created from updates in the same time period but with a maximum group size of 20.

To evaluate our classifiers, we randomly split the set of file pair examples into three parts. Two parts are used to create a training repository, and the remaining part is used to create a testing repository. This approach is known as the *hold out* method, and results in a testing set that is independent of the training repository. As we will discuss below, we use the training repository to create training sets to learn the co-update relation.

Table 1 shows the distribution of file pairs in the training repository and the testing set.

Table 1. Training repository and the testing set class distribution

	Relevant	***Not Relevant***	***#Relevant / #Not Relevant***
All	4547	1226827	**0.00371**
Training	3031	817884	**0.00371**
Testing	1516	408943	**0.00371**

As can be seen in Table 1, although the number of Not-Relevant file pairs has been greatly reduced, the set of file pairs is nevertheless very imbalanced. Such imbalanced data frequently appear in real world applications and pose difficulties to the learning task [1]. In Section 3 we discuss how we learn from less skewed training sets to circumvent this problem.

2.2. Feature sets

Corresponding to each Relevant or Not-Relevant file pair, there is an example that is labeled Relevant or Not-Relevant. Since the concept that we are trying to learn is about a pair of files, the attributes or features describing the example can be based on the properties of one of these files or a function of both of these files. The feature sets that we have used in our experiments can be divided into two groups:

- syntactic attributes
- text based attributes

In this section we present and discuss each one of these attribute sets.

2.2.1. Syntactic features. Syntactic attributes are based on syntactic constructs in the source code such as function calls, or data flow information such as variable or type definitions. These are the attributes extracted by static analysis of the source code. Computing the value of some of these attributes involves steps similar to the ones taken to measure well known static software product metrics such as fan in, fan out, and cyclomatic complexity [8]. In addition to these we have also used attributes based on names given to files. Table 2 shows the set of syntactic attributes used in the experiments reported in this paper.

Many of the entries in this table are self explanatory. Interesting software unit (ISU) denotes the first file in a pair and the other software unit (OSU) denotes the second file. Conceptually, in the Relevant Based method we want to learn what makes the second file in a pair (OSU) to be Relevant or Not Relevant to the first file in the pair (ISU). We say a subroutine is directly referred to if it appears in the main executable portion of a source file, e.g. main function in a C program. Subroutines that are referred outside the main executable part of a source file or non root subroutines in the static call graph of the source file are said to be *indirectly* referred to.

2.2.2. Text based features. Most source files include comments. These comments provide a textual description of what the program is intended to do.

Problem reports describe, in English, a problem or potential bug in the software. They may also include other information deemed to be helpful to the person who will eventually fix the problem. Examples of such additional information include program traces, memory dumps, hardware status information etc.

Table 2. Syntactic attributes

Attribute Name	*Attribute Type*
Same File Name	Boolean
ISU File Extension	Text
OSU File Extension	Text
Same Extension	Boolean
Common Prefix Length	Integer
Number of Shared Directly Referred Types	Integer
Number of Shared Directly Referred non Type Data Items	Integer
Number of Routines Directly Referred in ISU and Defined in OSU	Integer
Number of Routines Directly and Indirectly Referred in ISU and Defined in OSU	Integer
Number of Routines Defined in ISU and Directly Referred in OSU	Integer
Number of Routines Defined in ISU and Directly and Indirectly Referred by OSU	Integer
Number of Shared Routines Directly Referred	Integer
Number of Shared Routines Among All Routines Referred in the Units	Integer
Number of Shared Direct and Indirectly Referred Routines	Integer
Number of Shared Files Included	Integer
Directly Include You	Boolean
Directly Include Me	Boolean

Both problem reports and source file comments provide additional sources of knowledge that can be used to extract attributes. The idea here is to associate a set of words with each source file. Therefore instead of looking at a source file as a sequence of syntactic constructs, we can view them as documents.

We have adapted the *vector* or *bag of words representation* [9] to the task of learning the co-update relation by associating a vector of Boolean features with each *pair of files*. The bag of words representation is frequently used in information retrieval and text classification. The idea is that each document in a set of documents is represented by a bag of words appearing in all the documents in the set. In the Boolean version of this representation the vector corresponding to a document consists of Boolean features. A feature is set to *true* if the word corresponding to that feature appears in the document, otherwise the feature is assigned a *false* value.

Since our examples describe the relation between pairs of source files (f_i, f_j), we need to create a *file pair feature vector* for each such pair. We do this by finding the intersection of the individual file feature vectors associated with f_i, and f_j. This is in effect similar to applying a logical AND operation between the two file feature vectors. By doing so we find similarities between the two files. In the intersection vector, a feature is set to true if the word corresponding to the feature is in both documents that are associated with the files in the pair.

An important issue in using a bag of words approach in text classification is the selection of the words or features. We used an acronym definition file at Mitel Networks to create a set of *acceptable words.* Since these acronyms belong to the application domain, one would expect that they will include some of the most important and commonly used words. From this initial set we removed an expanded version of the set of stop words proposed in [6]. We further refined the set of acceptable words by filtering a set of problem reports against them. We repeated this process a few times and at each iteration updated the set of acceptable words by analyzing the rejected words using word frequencies, and the intuitive usefulness of words in the context of the application domain.

Although this acceptable wordlist was created by a non-expert and as a proof of concept, we were motivated by an earlier promising work in creating a lightweight knowledge base about the same application domain that used a similar manual selection technique for the initial set of concepts [10]

The word extractor program also performs a limited text style analysis to detect words in uppercase in an otherwise mostly lowercase text. We found that in many cases, such words tend to be good candidates for technical abbreviations or acronyms and domain specific terms. The program also incorporated some domain specific knowledge to reduce the noise in the extracted set of words. For instance the program detects memory dumps or distinguishes a word followed by a period at the end of a sentence from a file name or an assembler instruction that includes a period. We believe the results presented in this paper would be further improved if we could benefit from the domain experts' knowledge in creating these word lists.

As part of the process of creating the set of acceptable words we also have created two other lists that we called the *transformation list* and the *collocation list*. Using the transformation list, our word extractor program performs actions such as lemmatization or conversion of plurals to singulars that result in the transformation of some words to acceptable words. We would also like to preserve sequences of interesting words that appear together even though the participating words may not be deemed acceptable on their own. This is accomplished by using the collocation list.

In Sections 2.2.3 and 2.2.4 we discuss how we can use the source comments and problem report words as features.

2.2.3 Source file comment features. Finding comments in a source file is a relatively trivial task. We used the three lists mentioned above to filter the comments in each source file and associate each file with a set of comment words. We then used the bag of words representation discussed above to create the file pair feature vectors.

2.2.4. Problem report features. Since each update that results in changing source files is in response to one or more problem reports, one can associate each problem report with one or more files changed to fix the problem.

Since description of a problem can be viewed as a document, a bag of words can be associated with each problem report. Lets assume that a file f_i is changed as a result of addressing problem reports p_1, p_2,…, p_n. Furthermore assume that these problem reports are represented by bags of words called W_{p_1}, W_{p_2},…, W_{p_n}. We create a bag of words for a file by finding the union of

problem report bags of words $\bigcup_1^n W_{p_i}$. By doing so we associate with each file a bag of all the words that appear in some problem report that caused the file to change. The use of a union of bags of words allows us to account for different reasons resulting in a file being changed. Once again this new bag of words is filtered using the three lists discussed in section 2.2.2. We remind the reader that a word in principle can appear in any problem report. Two files do not need to share a problem report in order to share a word. In other words, there is a 1 to n relation between problem report words and source files. The process described above will not generate attributes that uniquely identify examples.

Once a file is associated with a bag of problem report words it can be represented as a feature vector, and a file pair feature vector can be created using these individual file feature vectors.

In Section 3 we describe the results of experiments using different sets of feature vectors.

3. Experiments

To properly evaluate and compare the effects of using different techniques and feature sets, first we need to choose a performance measure or visualization method that best suits the task at hand. As we discussed earlier, the data in Table 1 shows the large imbalance that exists among the Relevant and Not Relevant classes. It is well known that in settings such as this *accuracy*, which is the number of cases correctly classified over total number of cases classified, is not the best performance measure. The reason for this is that a simple classifier that always chooses the majority class will have a very high accuracy, however such a classifier has no practical value.

It is also the case that most learning algorithms are designed to maximize accuracy, and therefore faced with a training set with a large imbalance they generate a classifier that always suggests the majority class. Therefore to learn the co-update relation, we created less-skewed data sets from the training repository and tested the generated classifiers using the complete testing set. Each training set included all the Relevant examples in the training repository and K times as many Not-Relevant examples from the training Repository. We used the following 18 values for K:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50

In the experiments using the syntactic attributes, the Not-Relevant examples in these less skewed training sets were formed by selecting a stratified sample of the Not-Relevant examples in the training repository. In other words in the training sets we tried to maintain, as much as possible, the same proportion of Not-Relevant examples with a certain attribute value vector $a_1, a_2, \ldots, a_n,$ `class` as was present in the skewed training repository.

Precision and recall are standard metrics from information retrieval, and are also used in machine learning. *Precision* 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.

While precision and the recall of the Relevant class are appropriate measures for this application, they are not always convenient to use to compare classifiers performance. We frequently found that, when comparing classifiers generated by different methods of creating training sets, while the precision plot for one method was more dominant, the recall plot for the other method tend to be more dominant.

We would like to use a visualization method which depicts measures which are equally important as precision and recall but on one single plot. ROC plots [4] provide us with such a tool. In an ROC plot the horizontal axis represents the *false positive rate* and the vertical axis represent the *true positive rate*. For our application the Relevant class is the same as the positive class. We use the following 'confusion matrix' to define these measures.

		Classified As	
		Not Relevant	Relevant
True Class	Not Relevant	a	b
	Relevant	c	d

Figure 2. A confusion matrix for a two class classification problem

The true and false positive rates for the Relevant class are defined as

$$\mathrm{TP}_R = \frac{\text{Relevant cases correctly classified}}{\text{Number of Relevant cases}} = \frac{d}{c+d}$$

$$FP_R = \frac{\text{Not - Relevant cases incorrectly classified}}{\text{Number of Not Relevant cases}} = \frac{b}{a+b}$$

Note that TP_R is the same as the precision of the Relevant class.

In an ROC curve the following holds:

- Point (0,1) corresponds to *perfect classification*, where every positive case is classified correctly, and no negative case is classified as positive.
- Point (1,0) is a classifier that misclassifies every case
- Point (1,1) is a classifier that classifies every case as positive
- Point (0,0) is a classifier that classifies every case as negative

Better classifiers have (FP, TP) values closer to point (0, 1). A classifier is said to dominate another classifier if it is more 'north west' of the inferior classifier.

Both true and false positive measures are intuitively applicable to our application domain. A high true positive value means that the classifier correctly classifies all the existing Relevant examples in the testing set. A low false positive value means that the classifier does not classify many Not-Relevant examples as Relevant. Once superiority of a classifier over another classifier is determined by using the ROC plot one can further investigate the quality of the better classifier by other means such as precision and recall plots.

3.1 Syntactic versus text based attributes

Figure 3 shows the ROC plots generated for the 18 ratios of skewness between Not-Relevant and Relevant examples using syntactic and text based features.

We used exactly the same set of Relevant and Not-Relevant pairs for all experiments done with the same skewness ratio. In each plot the classifier with imbalance ratio = 1 corresponds to the rightmost point on the plot. As the ratio of Not-Relevant examples to the Relevant examples in the training set increases, the true and false positive ratios decrease. The lower leftmost point on each plot corresponds to an imbalance ratio 50 classifier. While increase in the training set skewness has the undesirable effect of decreasing the true positive rate, the amount of change in the case of problem report feature based classifiers is much less than the classifiers using other feature sets.

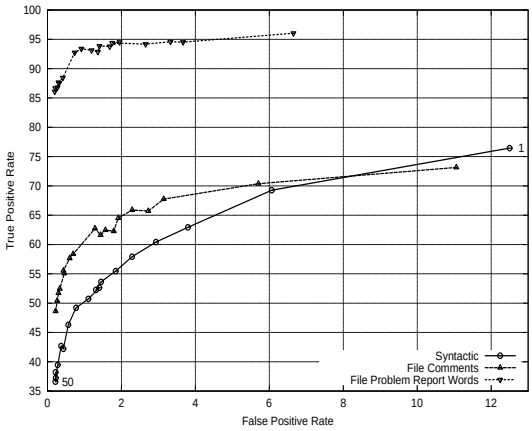

Figure 3. Comparison of syntactic and text based feature sets

As Figure 3 shows, the classifiers learned from syntactic attributes generate interesting true and false positive values, but their performance is not sufficient to use them in the field. The figure shows that classifiers generated from the problem report feature set clearly dominate the classifiers generated from source file comment and syntactic feature sets. The figure also shows that although the comment feature set does not perform as well as the problem report feature set, the classifiers generated from this feature set still dominate the classifiers generated form syntactic features.

In Figures 4 and 5 we have shown the precision and recall plots corresponding to the 18 skewness ratios discussed above. As can be seen here, the increase in skewness results in an increase in the precision of the Relevant class and decrease in its recall. However in the case of problem report feature based classifiers the degradation of recall values occurs in a much slower rate. For the imbalance ratio of 50, the problem report feature based classifier can achieve a 62% precision and 86% recall. In other words out of every 100 file pairs that the classifier predicts may change together, it is correct in 62 cases. Also the classifier can correctly identify 82 out of every 100 file pairs that actually change together. The importance of such performance values becomes more apparent when we take into account that a maintenance programmer may be faced with thousands of potential source files in the system.

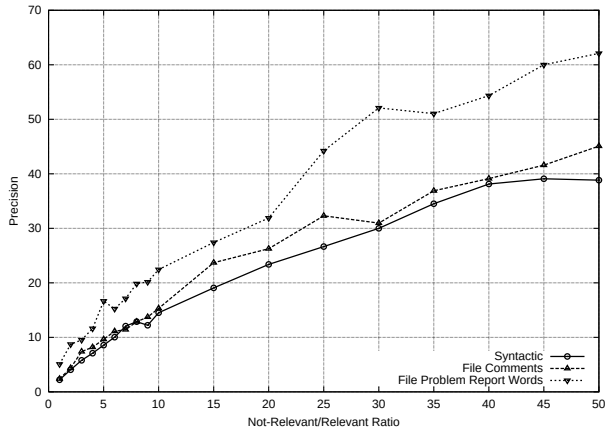

Figure 4. Comparison of the precision of classifiers generated from syntactic and text based feature sets

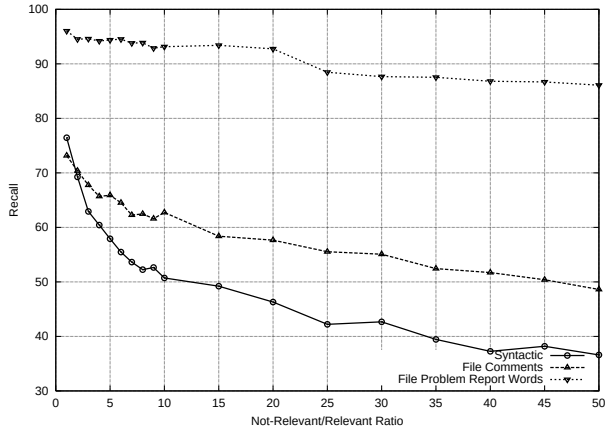

Figure 5. Comparison of the recall of classifiers generated from syntactic and text based feature sets

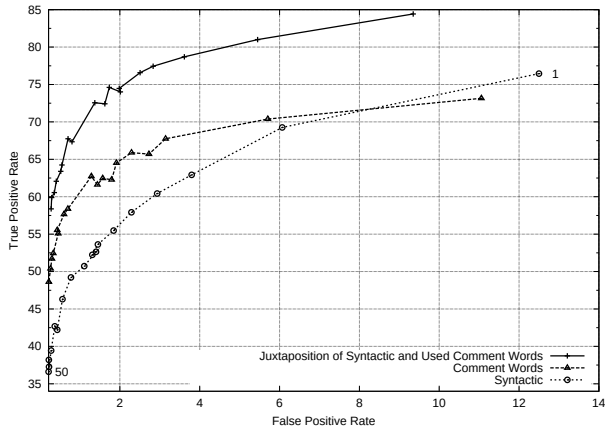

Figure 6. Combining syntactic and source file comment features

3.2 Combining feature sets

We also investigated the effects of combining different feature sets. Feature sets can be combined by simply juxtaposing i.e. concatenating, features in each set. This is the obvious choice when combining syntactic and text based features because they are different in their nature.

In the case of text based features, such as problem report word features and the source file comment word features, we have at least two other alternatives. A word can appear in both a problem report and file comment and therefore be a feature in both feature sets. We can create a combined feature set by finding the intersection of the two sets of features, or we can find the union of the sets. In the first case, the new feature set consists of words appearing both in the comments of some source file and problem report words associated with a source files. In the second case, a word need only to appear in the comments of some source file or in a problem report associated with a source file.

We have performed extensive experiments using the concatenation and the union method of combining problem report word and source file comment feature sets. Due to space constraints, the experiments using the union method will be discussed elsewhere. Here, we would like to report that the union representation did not improve the results in most cases, including the more interesting results such as the ones obtained for ratio 50 problem report word features classifier discussed in the previous section.

In Figure 6 we have shown the ROC plots for classifiers that are generated from examples using the concatenation of syntactic and source file comment features.

Instead of using all the comment words features in the combined feature set, we only used features that appeared in the decision trees generated in the comment word feature set experiments presented in the previous section. This in effect is performing a limited feature selection on source file comment feature set. Feature selection is an active research area in machine learning. The idea is to select a subset of available features to describe the example without the loss of quality of the results obtained. A smaller feature set has the obvious benefit that creating an example or case requires making less observations or calculation to find the values for the features. In some cases, appropriately selected smaller feature sets may also improve the quality of the results obtained.

This figure shows that the concatenation of these two feature sets generated classifiers that considerably dominate both syntactic and source file comment feature sets.

We also combined syntactic features with features that were used in problem report word feature set experiments discussed in Section 3.1. The results are shown in Figure 7. In this figure once again we observed that the combination of syntactic and text based features improved the existing results for most of the ratios including the more interesting classifiers generated from more skewed training sets. The improvements in the results are not as important as the ones seen in the case of combining file comment and syntactic features. This is not very surprising considering the fact that problem report word features on their own generate high quality results

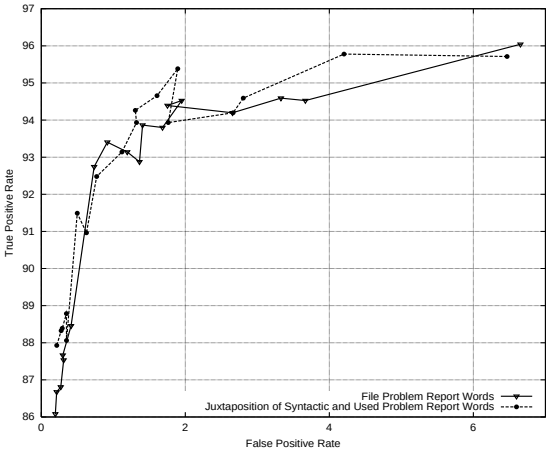

Figure 7. Combining syntactic and problem report word features

4. A closer look into models learned

One benefit of an explainable model such as a decision tree is the ability to find out why the model classifies an example as being of a certain class. The other benefit is that the model may reveal interesting relations that are not necessarily known. The extent of what can be learned from a model depends on factors such as the features themselves and the complexity of the model.

As mentioned above, our best models were obtained when we used problem report based features. We could improve on these results by combining them with syntactic attributes. We further analyzed the decision trees generated by these combined features to see what are the most influential features in the decision making as far as the C5.0 induction algorithm is concerned. These are words that appear higher up in the generated decision trees.

In Table 3 we have shown the features appearing at the top two levels of eighteen decision trees generated from the combined

syntactic and used problem report feature sets. Foe each feature we only have shown the highest level it has appeared in among the eighteen decision trees. For instance the word "hogger" has appeared nine out of eighteen times at the highest level in the decision trees i.e. level 0. In remaining nine decision trees, this feature may have appeared at a lower level including level 1, however we only have shown the highest contribution of this word into the decision making process.

Table 3. Top attributes in combined syntactic and problem report feature set

Level	Frequency	Feature
0	9	hogger
	3	rad
	2	play
	2	trunk
	1	fd
	1	ver
1	10	Same File Name
	4	acd
	4	ce..ss
	4	e&m
	2	group
	1	appear_show
	1	call
	1	display
	1	extension
	1	internal
	1	mitel
	1	ms2008
	1	per
	1	set
	1	single
	1	ss430
	1	tapi

As can be seen in this table only one of the features shown, namely "Same File Name", belongs to the syntactic features. Upon further analysis, we found the following ranking among remaining syntactic attributes used, where the number in parenthesis shows the highest level the feature appeared in a decision tree.

(2) Common Prefix Length
(3) Directly Include You
(3) Number of Shared Files Included
(3) Same Extension
(4) Number of Shared Direct and Indirectly Referred Routines
(4) Number of Routines Directly and Indirectly Referred in ISU and Defined in OSU
(5) Number of Shared Directly Referred Types
(6) Number of Shared Directly Referred non Type Data Items
(6) Number of Shared Routines Directly Referred
(13) Number of Routines Defined in ISU and Directly and Indirectly Referred by OSU
(15) Number of Shared Routines Among All Routines Referred in the Units
(78) OSU file Extension

These results show that the file name based attributes and file inclusion appear before other features based on routine calls and data and type definitions and reference. This suggests the importance of file name conventions and file inclusion mechanism as a way to group files in the system and share functionality.

Table 4. Most accurate paths in covering training Relevant examples for imbalance ratio 50

Path	# Examples
hogger & ce..ss	935
~hogger & Same File Name & fd & ~fac	153/2
~hogger & ~Same File Name & alt & corrupt	103/2
~hogger & ~Same File Name ~alt & zone & command	99/3
~hogger & ~Same File Name ~alt & ~zone & msx & debug	76/1

We also computed the paths in the decision tree generated from this combined feature set for imbalance ratio 50 training set. To provide a better sense as to how the tests on these features results in a classification of a file pair, in Table 4 we show the top 5 paths to a leaf classified as Relevant. In the second column we show the number of training examples labeled as Relevant which satisfy the conditions in the path. If there is a second number in the column, it shows the number of Not Relevant examples that satisfy the conditions in the path and therefore are misclassified as Relevant. A tilde (~) is used to show that the test is negative which means the word is not shared by problem reports or the syntactic attribute evaluated to a false value e.g. two files do not have the same file name.

Obviously there is a need for further analysis of these paths by an expert since some words or their combinations carry special meaning that may be missed by someone who is not as familiar with the system.

We also looked into the classification results of the Relevant examples in the testing set. We sorted the paths by their accuracy in terms of classifying testing Relevant examples. We found exactly the same five paths and in the same order shown above.

C5.0 is also capable of creating rule sets as opposed to decision trees. The rule sets have the benefit of sometimes being simpler than decision trees. However our experiments with problem report feature sets showed that the generated rule sets perform worse than the corresponding decision trees for all imbalance ratios, therefore we only present the decision tree analysis results here. However the fact remains that one could in principle find alternative and potentially simpler explanation than the ones obtained from decision trees.

Although we are constrained by space, it is not difficult to envision similar or even more sophisticated analysis of the decision trees or rule sets generated from other features sets. We intend to continue this analysis and report the results in future publications.

5. Conclusions

In this paper we presented an application of machine learning to source code level software maintenance activity. We showed how one can use software update records to learn a relation that can be used to predict whether change in one source file may require a change in another source file. We empirically compared three feature sets extracted from different information sources. Our experiments show that the problem report word features generate classifiers with precision and recall values appropriate for real life deployment of our method. We also showed that combining used text based features with syntactic attributes in most cases improves the results. In Section 4 we presented the examples of analysis that can be performed on the generated decision trees. Such analysis can be used to better understand the learned model (or the relation that it represents). Clearly one important future step for our research is the question of the field evaluation of these classifiers. We also perceive other research opportunities including:

- Investigating the effects of automatic feature selection on the results
- Investigating the effects of automatic feature selection on the results
- Experimenting with other SMS and non-SMS based feature sets.
- Using alternative learning methodologies such as Co-Training [3] to label file pairs. For instance, we could start with a set of Relevant examples labeled by the co-update heuristic, and a smaller set of Not-Relevant examples provided by software engineers. We could repeatedly generate unlabeled file pairs (and the corresponding examples) and then using classifiers based on problem report, source file comments, or syntactic features classify a desired number of unlabeled examples and add them to the pool of labeled examples. In each repetition we would create new classifiers from the new set of training examples available and evaluate them. This process is repeated as long as better results are obtained.
- Using the learned co-update relation in other applications, such as subsystem clustering, which is an important tool in better understanding large software systems.

Acknowledgements

The authors would like to thank the SX 2000 team for their continuous support and help. This work was sponsored by CSER and supported by Mitel and NSERC.

References

[1] AAAI 2000 Workshop on Learning from Imbalanced Data Set, Austin, Texas.

[2] B. Bellay B. and H. Gall. An Evaluation of Reverse Engineering Tool Capabilities. *Journal of Software Maintenance: Research and Practice*, v. 10 no. 5, pp. 305-331, 1998.

[3] A. Blum and T. Mitchell. Combining Labeled and Unlabeled Data with Co-Training. *Proceedings of the 11th Conference on Computational Learning Theory*, Madisson, WI, Morgan Kaufmann Publishers, pp. 92-100, 1998.

[4] J. Egan. Signal Detection Theory and ROC analysis. New York, Academic Press, 1975.

[5] 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, pp. 87-97, 1995.

[6] D. D Lewis. Representation and Learning in Information Retrieval. *Doctoral dissertation, University of Massachusetts*, 1992.

[7] M. Marchand & J. Shawe-Taylor. The Set Covering Machine, *Journal of Machine Learning Research*, v. 3, pp. 723-746, 2002.

[8] T.J. McCabe. A Complexity Measure. *IEEE transactions on Software Engineering*. v. 2 no. 4, pp. 308-320, 1976.

[9] D. Mladenic. Text-Learning and Related Intelligent Agents: A Survey. *IEEE Intelligent Systems*, v. 14 no. 4, pp. 44-54, 1999.

[10] J. Sayyad Shirabd, T.C. Lethbridge, and S. Lyon. A Little Knowledge Can Go a Long Way Towards Program Understanding. *Proceedings of the 5th International Workshop on Program Comprehension*. Dearborn, MI, pp. 111-117, 1997.

[11] J. Sayyad Shirabd, T.C. Lethbridge, and S. Matwin. Supporting Software Maintenance by Mining Software Update Records. *Proceedings of the 17th IEEE International Conference on Software Maintenance*, Florence, Italy, pp. 22-31, 2001