# Mining the Maintenance History of a Legacy Software System

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

*A considerable amount of system maintenance experience can be found in bug tracking and source code configuration management systems. Data mining and machine learning techniques allow one to extract models from past experience that can be used in future predictions. By mining the software change record, one can therefore generate models that can be used in future maintenance activities. In this paper we present an example of such a model that represents a relation between pairs of files and show how it can be extracted from the software update records of a real world legacy system. We show how different sources of data can be used to extract sets of features useful in describing this model, as well as how results are affected by these different feature sets and their combinations. Our best results were obtained from text-based features, i.e. those extracted from words in the problem reports as opposed to syntactic structures in the source code.*

## 1. Introduction

In this paper we investigate the application of machine learning (inductive) methods to the software maintenance process.

Maintaining a software system is an activity typically performed in an environment where knowledge about how to proceed is scarce. This scarcity of knowledge is due to a combination of factors: the sheer size and complexity of the systems, high staff turnover, poor documentation and the long periods of time these systems must be maintained. This is certainly the case for legacy software systems where frequently there are non-trivial relations between different components of the system, that are not known.

This lack of knowledge is often the source of delays and other complications such as introduction of new errors during the 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 that they may better understand software systems.

Certainly discovering and understanding non-trivial relations plays an important role in overall understanding of that system. It has been suggested by other researchers [2, 6] that using Artificial Intelligence (AI) techniques such as employing knowledge bases can further benefit research in reverse engineering and program comprehension. Knowledge Based Software Engineering (KBSE) is the classic application of AI into software engineering. The common feature of these systems is that they frequently employ knowledge bases or expert systems. In a previous paper [12] we mentioned some of the issues with large knowledge base system.

The focus of the current paper is 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.

Despite their enormous potential and their success in other application domains, inductive methods are little used in software engineering, particularly in source code level software maintenance. This is in sharp contrast to knowledge bases and deduction which have been actively researched as part of KBSE since early 1980's.

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?*"

One way to assist a software engineer to answer this question is to find relations that exist among files that, when present, tend to indicate that the files will both need to change together. We refer to this particular relation as the *co-update* relation. The co-update relation is an instance of a relation learned in the context of software maintenance activity. We refer to this and similar relations in general as Maintenance Relevance Relations, since we are interested in the relevance of files to each other during software maintenance. The measure or the degree of relevance is the result of classification e.g. whether two files are relevant or not. The nature of relevance is defined by the particular relation of interest. For instance, two files may be relevant to each other because they are part of the same subsystem. In our case the relevance of two files to each other is in terms of them being updated together.

One place to look for such relations is problem tracking systems where software engineers keep track of reported problems and changes applied to the system to fix them. We can view this software update data as past maintenance experience, and use machine learning to extract the above-mentioned relations.

We used a large legacy telephone switch system, i.e. a PBX, developed by Mitel Networks as a case study. 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. Formulating the problem as classification

The machine learning techniques used in our experiments fall under the category of *classification learning*. As shown in Figure 1, in classification learning we apply an *induction algorithm* to a set of pre-classified *training examples* of the *concept* we want to learn e.g. a relation between fields. The output of the induction algorithm is referred to as a *classifier*, which is also known as a *model* or a *concept description*. Each training example is a *known* or *solved* 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.

In Figure 1 an induction algorithm reads $k$ pre-classified examples $e_1, \ldots, e_k$ of some concept we want to learn. Each example $e_i$ is described by $n$ attributes $a_{ij}$ $j$=1, …, n, and is assigned a class $c_i$ which is one of $m$ predefined possible classes. The induction algorithm generates a classifier that can be used to classified an unseen (or unclassified) case.

The relation we are interested in learning is between pairs of files. We would like to predict, given two files, whether when one changes, the other may need to be changed (i.e. that they may change together). We can formulate this as a classification problem where we use $m$ distinct classes to indicate how likely it is that two files change together. Therefore in a classification formulation of this learning problem the concept that we are learning is the co-update relation that holds between files. The generated classifier or model represents the learned relation.

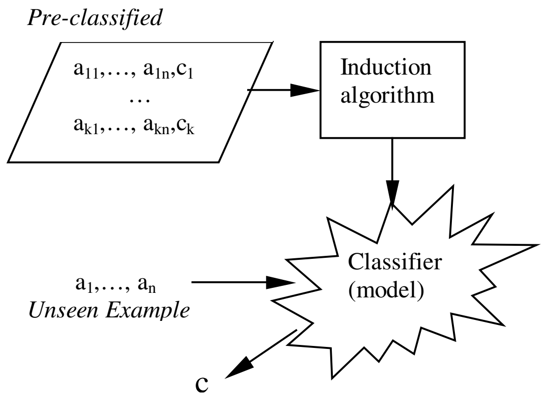

**Figure 1. Classification learning**

Lets assume that we learn the co-update relation in the form of a classifier. Such a classifier, embedded in an application with a proper user interface, could be used at least in one of the following scenarios:

- An expert software engineer who suspects a change in file $f_i$ may result in a change in file $f_j$ can provide a pair of files ($f_i$, $f_j$) to the application and receive a measure of their relevance.
- Probably a more frequent usage would be the case that a software engineer wants to know what other files may be affected by a change in a file $f_i$. In this case he or she submits the file name $f_i$ to the application and receives a list $f_j, \ldots, f_z$ of such files in return. In certain circumstances where the classifier suggests too many files, a ranking between the returned results may further assist the software engineer in the maintenance task. However, one would expect that a good classifier typically would not generate too many unrelated answers.

Furthermore the classifier can also be used in other applications such as subsystem clustering that can benefit from the knowledge of existing relations between files.

In this paper we will present a discrete version of co-update relation with two classes ($m$=2) 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]. A decision tree has a condition at each node. The condition is a test of the value of one of the predefined attributes used to describe the example. At each leaf, there is a prediction for the class 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 and the corresponding outcome if all the tests on the path result in a true value.

Decision tree learning 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. This is in contrast to other approaches such as most statistical methods. We intend to provide further structural analysis of the generated models in future publications.

## 2.1. Creating training and testing repositories

The first step in the learning process is obtaining a set of training examples – in our case this means we need a set of pairs of files where, for each pair, we know whether they are Relevant or not. Once we generate such a set we can create the corresponding examples by calculating the value of attributes used to describe an example. In the following paragraphs, we describe how we extract such data from maintenance records.

The typical process of fixing a problem in a software system starts with filing a *problem report*. The problem is then assigned to a software engineer. To fix the problem the software engineer may end up changing the source code by submitting an *update*. The updated system then goes through further testing and verification. If it passes this process, the update is considered as final or *closed.*

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.

We use the data stored in SMS for a certain periods of time to heuristically find the following two kinds of examples of the co-update relation:

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

To apply the above heuristics first we select a time period T=[$T_1$-$T_2$] 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'=[$T_3$-$T_2$] where $T_3 \leq T_1$, are labeled as *Not Relevant*[2]. Figure 2 shows the relation between T and T'. 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*.

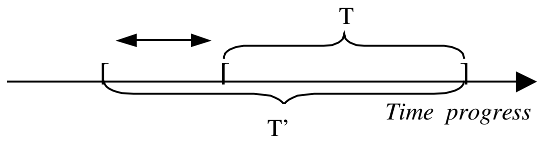

**Figure 2. The relation between time periods the co-update and Not-Relevant heuristics are applied**

Notice that, based on the closed world assumption T' must cover at least the same time period as T. It is also the case that a Relevant file pair ($f_i$, $f_j$) is generated based on existing facts; i.e. $f_i$ and $f_j$ are changed by an update in time period T. However, a Not-Relevant pair ($f_i$, $f_k$) is generated because within the time period considered there was no update changing $f_i$, and $f_k$ together, or in other words due to the lack of a fact to the contrary. This means that, in general, the Not-Relevant heuristic can introduce some labeling or classification noise due to lack of knowledge. In principle increasing the time covered by T' provides 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. However we can not choose a very large time period T',

[1] We have also experimented with a simple learning algorithm called 1R [5] and Set Covering Machines (SCM) [8]. 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.

[2] In the experiments discussed in this paper we have use d a time period T'=T or in other words $T_3$ = $T_1$.

because the changes applied to the system in the distant past may no longer reflect the current state of the system.

To create the set of Relevant and Not Relevant file pairs we processed the updates for the 1995-1999 time period. We found 1401 closed updates in response to 1213 problem reports. An update can change one or more files. 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 of greater than or equal to two. Out of all closed updates in the above time period, 43% changed two or more files. However 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. Upon further analysis of updates we found that 40% of all the updates have a group size between 2 and 20 inclusively. These updates constitute 93% of updates with a group size larger than 1. We have used a group size limit of 20 for the experiments reported in this paper.

Our version of the co-update relation will only focus on the HLL (high level language) files. We found that the information created by the Assembler parsers available to us was not as accurate as we had wished. This is mostly due to the less structured nature of the 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 $\binom{N}{2}$=N(N-1)/2 pairs of files. This is because we are not pairing a file with itself and the co-update relation is symmetric in nature. 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); i.e. the closed world assumption. Using the above formula, the number of potential file pairs when all the source files in the system are used is close to eleven million pairs. By using HLL files, only this number is reduced to approximately six million file 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.

One way to reduce the number of potential file pairs is to increase the size of T' when applying the Not-Relevant heuristic. By increasing the size of T' we will find a larger number of Relevant file pairs and because of the closed world assumption this means that we will have a smaller number of Not-Relevant file pairs or examples. However the number of Relevant file pairs grows slowly with the increase of the size of the time window, which means the reduction in the number of Not-Relevant file pairs also grows slowly.

An alternative way to reduce the number of potential file pairs is to constrain the first file in all potential file pair to be the same as 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. 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 using other sources of information. For instance 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 of course includes the SR that was created from updates in the same time period but with a maximum group size of 20.

Conceptually 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.

For the purpose of evaluating our classifiers we used the *hold out* method which involves randomly splitting the set of file pairs into three parts. Two parts form a training repository and one part is used for testing. By doing so we make the testing set independent of the training repository, and maintain the original distribution of classes.

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. Attributes used in the experiments

Each example of the co-update relation, that holds between a pair of files labeled as Relevant or Not-Relevant, must be described in term of a set of features. The attributes or features can be based on a property of one of the files in the pair but more often are a function of both of these files. To learn the co-update relation we have experimented with different sets of attributes. In general, these attribute sets can be divided into two groups:

- syntactic attributes
- text based attributes

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

**2.2.1. Syntactic attributes.** 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 attributes are extracted by static analysis of the source code. They also include attributes based on names given to files. In Table 2 we have shown a set of such attributes used in the experiments reported in Section 3.

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 [9]. 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) Relevant to the first file in the pair (ISU), or what makes the second file Not Relevant to the first file. 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 attributes.** Problem reports describe, in English, a problem or potential bug in the software. They contain the description of the problem and may also include other information deemed to be helpful to the person who will eventually fix the problem. This additional information may include program traces, memory dumps, hardware status information etc. As we mentioned earlier, they are stored and maintained in SMS, independently from source files themselves. Therefore they need to be processed separately from the system source files.

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

**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* [10] 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. Therefore, for each source file, we first create a *file feature vector* from the set of words associated with the source files.

However, the concept that we want to learn i.e. the co-update relation, involves a pair of files. To create a *file pair feature vector* for a pair of source files ($f_i$, $f_j$) we find 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. The idea here is to find similarities between the two files. In the intersection vector, a feature is set to a true value 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 created a set of *acceptable words* by first using an acronym definition file which existed at Mitel Networks. These acronyms belong to the application domain, therefore they 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 i.e. words not to be included, proposed in [7]. Then through a few semi-automatic iterations we filter a set of problem reports against these acceptable words. We update the set of acceptable words by analyzing the rejected words using word frequencies, and 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 [11].

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 attributes.** 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 attributes.** Each update is in response to one or more problem reports, thus each problem report can be associated with one or more files changed to fix the problem. Conversely a file is changed in response to one or more problem reports. Therefore we can associate each file with one or more problem reports.

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 $f_i$ by finding the union of problem report bags of words $\bigcup_{k=1}^{n} W_{p_k}$ for problem reports that resulted in a change in $f_i$. 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. Since there is a 1 to n relation between problem report words and source files, this process will not generate attributes that uniquely identify examples. Once again the new bag of words is filtered using the three lists discussed in section 2.2.2.

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 be able to compare and evaluate the effects of using different feature sets, we first need to choose a performance measure or method of comparison that is appropriate for the particular task at hand. Looking back at Table 1 we observe a large imbalance between the Relevant and the 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.

To learn the co-update relation, we created less-skewed data sets from the training repository and tested the generated classifiers using the complete and independent 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 3. A confusion matrix for a two class classification problem**

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

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

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

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

Figure 4 shows the ROC plots for two classifiers A and B.

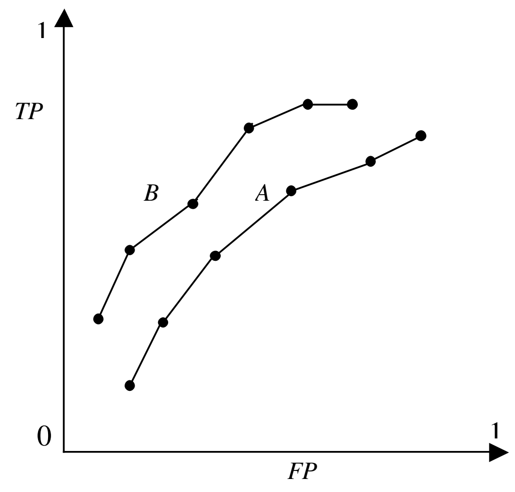

**Figure 4. An ROC curve**

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. In Figure 4, classifier B clearly dominates classifier A.

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. It should also be noted that $TP_R$ is the same as the recall of the Relevant class. 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 5 shows the ROC plots generated for the 18 ratios of skewness between Not-Relevant and Relevant examples using syntactic and text based features.

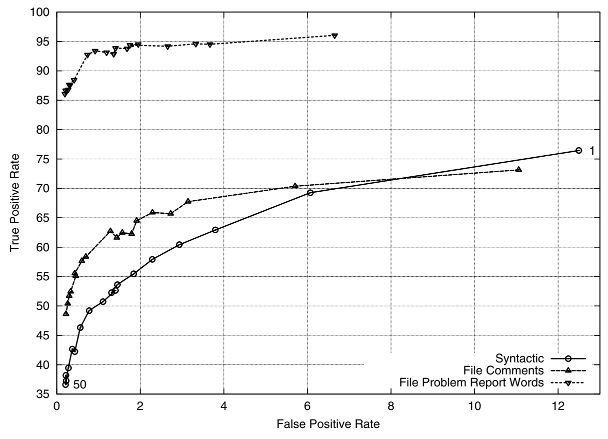

**Figure 5. Comparison of syntactic and text based feature sets**

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.

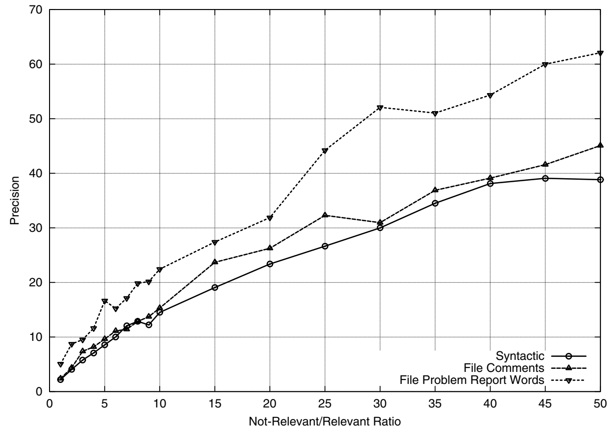

**Figure 6. Comparison of the precision of classifiers generated from syntactic and text based feature sets**

We further compared the precision and recall values generated by the best classifiers of each plot. As Figure 5 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 examples described by 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.

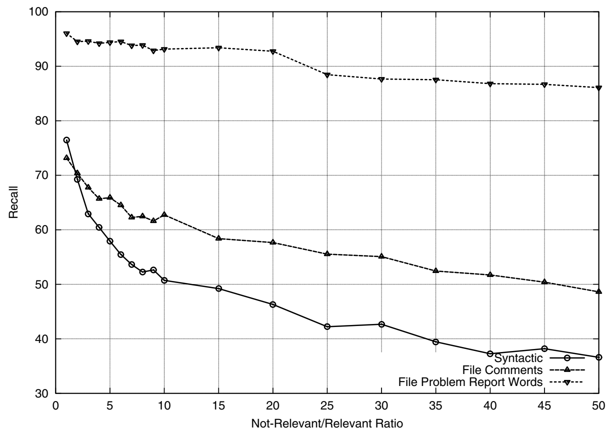

**Figure 7. Comparison of the recall of classifiers generated from syntactic and text based feature sets**

In Figures 6 and 7 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.

## 3.2 Combining feature sets

We also investigated the effects of combining different feature sets. Feature sets can be combined by simply 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 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.

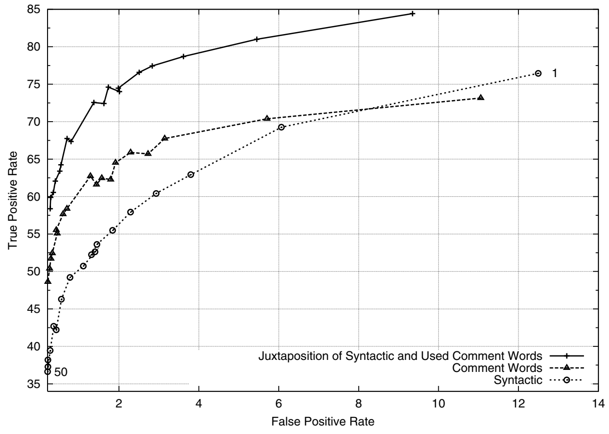

**Figure 8. Combining syntactic and source file comment features**

In Figure 8 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 tree 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 fewer observations or calculations 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 3.1.1. The results are shown in Figure 9. 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

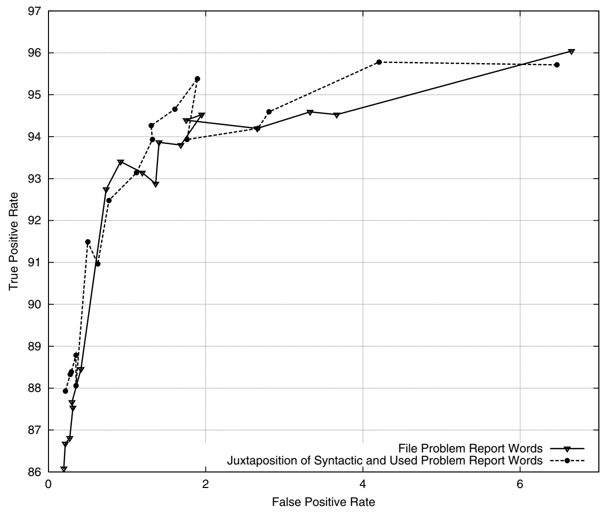

**Figure 9. Combining syntactic and problem report word features**

## 4. 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 a 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 features based on problem report words 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.

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:

- Experimenting with other SMS and non-SMS based feature sets.
- Investigating the effects of automatic feature selection on the results
- 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] R.C. Holte. Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Machine Learning*, vol. 3 pp. 63-, 1993.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.