Mining the Software Change Repository of a Legacy Telephony System

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

Abstract

Ability to predict whether a change in one file may require a change in another can be extremely helpful to a software maintainer. Software change repositories store historic changes applied to a software system. They therefore inherently contain a wealth of information regarding (hidden) interactions between different components of the system, including the files that have changed together in the past. Data mining techniques can be employed to learn from this software change experience. We will report on our research into mining the software change repository of a legacy system to learn a relation that maps file pairs to a value indicating whether changing one may require a change in the other.

1. Introduction[1]

In large software systems there are many unknown or undocumented relationships and interactions between different components of the system. Such undocumented relationships are major sources of complexity and cost for maintaining the system.

Source code management systems, along with error tracking and change repositories maintain a comprehensive change history of a system. They inherently store a wealth of information regarding many of the interactions and relationships among different components of the system. Data mining methods convert data containing past experience with a given process into the knowledge about this process. Therefore, source code management systems are a fertile area of application for data mining. The idea here is to learn relationships among software entities from the historic change records.

In Section 2 of this paper we present the notion of Relevance Relations among entities in a system. Section 3 shows how the problem of learning a relevance relation can be mapped to a classification learning problem, while Section 4 describes the measures used to evaluate the quality of the learned classifiers.

As a proof of concept we will learn a relevance relation between file pairs in a legacy system. This relation maps a pair of files to a value indicating whether changing one may require a change in the other. Such a relation can be very helpful to a software maintainer.

To learn this relation we will mine the change repository of our subject legacy system. Sections 5 to 7 provide the details of this process as well as some of the results. The conclusion and future work is presented in section 8.

2. Relevance Among Software Entities

In this section we provide the definitions of Relevance Relation and other concepts closely related to it. We will also discuss a specific example of such relations in the context of software maintenance.

2.1 Relevance Relations

Definition: A *Software Entity* is any semantically or syntactically valid construct in a software system that can be seen as a unit with a well defined purpose[2]..

Examples of software entities include documents, source files, routines, modules, variables etc.

Definition: A *Predictor* is a relation that maps tuples of one or more software entities to corresponding values reflecting a prediction made about the elements of the tuples.

Definition: A *Relevance Relation* (RR) is a predictor that maps tuples of two or more software entities to a value *r* quantifying how *Relevant* i.e. connected or related, the entities are to each other. In other words *r* shows the strength of *relevance* among the entities. Therefore, "Relevant" here is embedded in, and dependent on, the definition of the relation.

For instance, in section 2.2 we will discuss a relevance relation called co-update that maps file pair tuples to one of two relevance values Relevant or Not-Relevant.

The relevance value *r* can be a number between 0 (lack of any relevance) and 1 (100% relevant), in which case the

[1] Due to the space limitation the discussion of related work has been kept to a minimum.

[2] Unless otherwise stated, in this paper an entity means a software entity.

relevance relation is *continuous*, or it can be one of a set of predefined values, in which case the relevance relation is *discrete*.

2.2 A real world example of Relevance Relations

When a maintenance programmer is looking at a piece of code, such as a file or a routine, one of the important questions that she needs to answer is:

"*which other files should I know about, i.e. what other files might be relevant to this piece of code ?*".

Knowing the answer to this question is essential in understanding and maintaining any software system regardless of the type of maintenance activity.

In this paper we will therefore focus on a special kind of relevance relation where the entities are files. We want to learn a special class of Maintenance Relevance Relations (MRR) called the *co update* relation:

co-update(f_i, f_j) $\rightarrow$ {Relevant, Not-Relevant} where,

$i \neq j$ and f_i and f_j are any two files in the system.

co-update(f_i, f_j) $\rightarrow$ Relevant means that a change in f_i may result in a change in f_j, and vice versa .

As it can be seen in this definition, the co-update relation is a discrete relevance relation mapping two entities (files in this case) to one of the two relevance values.

A relevance relation such as co-update could be used to assist a maintenance programmer in answering the above question about files.

3. Relevance Relations and classification learning

While in most cases one can easily specify the behavior of a relevance relation in terms of its domain and range, the actual definition of the relation is unknown. It is also the case that one could provide instances of the relation of interest, without knowing an exact definition for it. For instance by looking at a software change repository we can find past instances of the co-update relation, without knowing the definition of the relation itself. If we knew the definition, we could use it to predict whether for any pair of files in the system a change in one may require a change in another, even if we do not have a record of them being changed together in the past. Therefore, it is natural for one to try to learn from these instances.

The problem of learning a relevance relation can be directly mapped to a classification learning problem. Here the classifier represents the *learned* relevance relation. In classification learning terminology we are trying to learn a *concept* e.g. the co-update relation between files. Figure 1 shows the relation between a relevance relation and a classifier modeling it.

To learn a concept such as the co-update relation, an *induction algorithm* must be provided with pre-labeled (pre-classified) *examples* or *cases* of that concept. The example consists of the description of the concept and an assigned *class* or *label*. An example is described by calculating the value of a list of predefined *attributes* or *features*. The features of an example must describe the entities in the corresponding tuple which is mapped by the relevance relation. For instance an attribute could be the file type of the first file in a co-update tuple with possible values "source" and "header". Another attribute could be the number of routines called by both files in the tuple. In the first case the attribute is based on one of the entities in the tuple, while in the latter case the attribute is based on two entities.

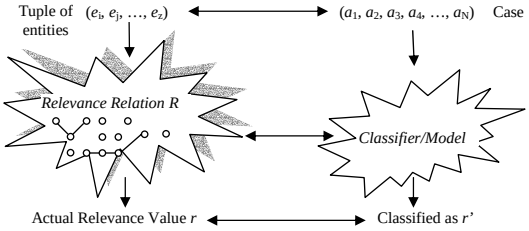

Figure 1. The Mapping between a Relevance Relation and a Classifier

The output of the induction algorithm is a classifier which will model the relevance relation of interest. As Figure 1 suggests, once a model is learned a previously unseen tuple of entities (e_i, e_i,…, , e_z) can be translated to a case with feature or attribute values (a_1, a_2,…, , a_N), and input to the learned model. The output of the model *r'* is an approximation of the actual relevance value *r* [3]. A classifier that always correctly classifies the given cases, accurately represents the corresponding relevance relation. However, this is hardly ever the case in a real world setting. In Section 4 we will discuss how we can measure the performance of generated classifiers or models.

Due to software engineers' time restrictions it is not always practical to ask them to provide instances of the co-update relation, therefore we used heuristics instead.

We extract from the software change repository all the updates applied to the system within the time period for which the data will be mined. The *Co-update heuristic* suggests that files changed together by each update be paired and label as Relevant.

The *Not-Relevant heuristic* labels a pair of files as Not-Relevant if these files are not changed by any updates within the time period used for mining

If $T=[T_1,T_2]$ is the period of time to which the Co-update heuristic was applied, $T'=[T_3,T_2]$ the period of time to which the Not Relevant heuristic is applied includes T i.e., $T_3 \leq T_1$

Our methodology allows the set of Not-Relevant tuples

[3] Here we emphasize that any relevance relation can be modeled by a classifier learned from the instances of that relation. For the special case of suggesting software entities that tend to change together one could also employ other techniques such as association rule learning as discussed in [5].

be further refined, if there is additional information, e.g. expert feedback, suggesting a tuple should not be labeled as Not-Relevant.

4. Measuring classifier performance

Figure 2 shows a *confusion matrix*, where the counts of predicted versus actual class of examples used for evaluating a model are tabulated. The following measures can be derived from this matrix.

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

Figure 2. A confusion matrix

$$TP_R = \frac{\text{Relevant cases correctly classified}}{\text{Number of Relevant cases}} = \frac{a}{a+b}$$

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

$$E_c = \frac{C_{PN} * b + C_{NP} * c}{a + C_{PN} * b + C_{NP} * c + d}$$

In our research we consider the Relevant class to be the positive class and the Not-Relevant class the negative class. TP_R and FP_R are the *True Positive* and the *False Positive* rates for the Relevant class. Readers familiar with the *Recall* measure will recognize that it is the same as true positive rate. E_c is the *Cost Sensitive Error Rate*. It generalizes the formula for *Error Rate* by allowing arbitrary cost factors to be assigned to each of the two possible misclassifications. C_{NP} is the cost factor for misclassifying negative examples e.g. Not-Relevant as positive. C_{PN} is the cost factor for misclassifying positive examples e.g. Relevant, as negative. When the costs for both kinds of errors are set to one, this formula simplifies to the formula for error rate.

All three measures above are normalized. Ideally we would like to have a classifier with a true positive rate of 1, and the false positive and error rates of 0.

5. Attributes used in our experiments

To learn the co-update relation, we have experimented with different sets of attributes. These attribute sets can be divided into two groups:

- Syntactic attributes
- Text based attributes

Syntactic attributes are based on syntactic constructs in the source code such as function calls, variable definitions or type definitions. These attributes are extracted by static analysis of the source code[4]. They also include attributes based on names given to files. Interested readers can find the complete list of these attributes in [4].

Text-based attributes allow us to exploit another source of knowledge about the files modified together: the text of comments and problem reports. Each file is represented by a vector of features that correspond to the words found in the collection of all comments or all problem reports. Such a "bag of words" is a commonly used representation method for documents in information retrieval and machine learning. The Boolean bag of words representation sets a feature to *true* if the corresponding word exists in the document and *false* otherwise.

We have adapted this representation to accommodate file pair tuples as is the case for the co-update relation. After assigning a bag of word feature vector to each file, we create an example for tuple (f_i, f_j, *r*) by creating a new bag of words feature vector which is the intersection (or logical AND) of the feature vectors corresponding to f_i and f_j. Therefore, in the new file pair feature vector a feature is set to true if the corresponding word appeared in the sets of words assigned to both f_i and f_i, otherwise the feature is set to false. The idea here is to find similarities between the two files. Of course the example will be labeled as *r* e.g. Relevant or Not-Relevant.

We have created bag of word feature vectors for files using:

- Source code comments
- Problem reports

In the first case a program file is seen as a document, i.e. assigned a set of words, consisting of the words in its comments.

In the case of problem reports the words in problem reports must somehow be associated with program files. This is achieved by creating a set of words for each file consisting of the words in all problem reports that caused it to change.

6. Experimental setup

The subject for our experiments was a large telephone switching software system (a PBX) developed by Mitel Networks corporation. This software was originally created in 1983 and is still a major source of revenue for our industrial partner. Approximately 1.9 millions lines of high level language (HLL) and Assembler code were distributed in about 4700 source files. The high level language source files which are the subject of this report constitute about 75% of these files.

In our research we used the source code and error tracking and update data maintained in a system called SMS. Using SMS one can view problem reports submitted against the system and updates applied to fix them. By applying the Co-update heuristic we extracted a set of file pairs that were changed together by updates submitted in the 1995 to 1999 time period, i.e. the set of Relevant file

[4]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.

pairs. Using the Not-Relevant heuristic mentioned earlier, we found a set of file pairs that were not changed together during this time period, i.e. the set of Not-Relevant file pairs[5].

The *group size* of an update is the number of files changed by it. Experiments reported in this paper limit the Relevant file pairs to the ones changed by updates where the group size is at most 20. These updates constitute 93% of updates with a group size larger than 1. Our experiments have shown that limiting group size generate better results than not doing so. We generate a new Relevant tuple (and example) for each individual update that change a pair of files together. Table 1 shows the distribution of Relevant and Not Relevant examples used in our experiments. We split these examples to 1/3 *Training Repository* and 2/3 *Testing Repository*.

Table 1. Class distributions

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

As discussed above for each instance of the co-update relation we generate an example by calculating the value for a set of predefined attributes. We use the relevance value of the instance as example's label e.g., Relevant.

Table 1 shows that the number of Not-Relevant examples is about 270 times the number of Relevant examples. This imbalance creates difficulties for most learning algorithms as they are designed to select models with higher accuracy. In this skewed scenario a classifier that classifies every example as Not-Relevant will have a very high accuracy, yet it will be completely useless. To compensate for this, we train our classifiers on far less skewed data sets. These training sets include all the Relevant examples and a sampled subset of Not-Relevant examples in the training repository so that thy will have the following Not-Relevant/Relevant ratios:

1-10,15,20,25,30,35,40,45,50

In other words we learn from 18 *training sets* with the above skewness ratios and test on the complete testing repository that has the original skewness among classes. In the reminder of this paper any reference to a *"ratio N classifier"* means a classifier that is trained on a training set with a skewness ratio *N*.

The learning system used in our experiments is C5.0[6], an advanced version of the C4.5 decision tree learner [3]. Decision tree learning is one of most widely used and well researched areas of machine learning. A decision tree is an explainable model that can be studied and reasoned about by domain experts.

7. Comparing models learned from syntactic and text based features

We have learned models of the co-update relevance relation by conducting three sets of experiments using the 17 syntactic feature set, the source file comment feature set, and the problem report feature set. Each set of experiments generated 18 classifiers corresponding to the above training skewness ratios. For each set of experiments we plotted TP_R of each classifier against its FP_R. This plot is known as the ROC plot. Figure 3 shows the ROC plot generated from these experiments[7]. In an ROC plot the ideal point is (1, 0) where the true positive rate is at its maximum and the false positive rate is at its minimum. A classifier that is on the north-west side of another classifier on the plot, i.e. closer to point (1, 0), is said to be the dominant one.. In the ROC plots shown in this paper the rightmost point corresponds to a classifier learned from a balanced training set, while the leftmost point corresponds to ratio 50 classifier.

As Figure 3 shows the text based attributes generate models that dominate the classifiers generated from syntactic features. The problem report based features generated the best models. Increasing the number of Not-Relevant examples in the training set causes a drop in TP_R and FP_R. The drop in TP_R, which is the undesirable effect, is far less in the case of classifiers learned from problem report based features. A closer look at the ratio 50 problem report based classifier revealed that it achieves a precision of 62% and a recall of 86% for the Relevant class. We believe performance values such as these makes a classifier a good candidate for field deployment.

We also combined text based features used in the classifiers of Figure 3 with syntactic features and repeated our experiments using these combined feature sets. As can be seen in Figures 4 and 5 the classifiers generated from the combined feature sets in most cases dominate the original text based classifiers. The effects are more prominent in the case of source file comment based classifiers, however this should not be very surprising as the problem report based classifiers already show fairly high quality.

Finally, Figure 6 shows the cost sensitive error rate plots for the ratio 50 text based classifiers and two random classifiers used as comparison baselines. In a two class classification problem, a common baseline is a random classifier with a probability of 50% for each class. However since the distribution of our classes is skewed we have also used a classifier with the same skewness as the testing repository. Examples in the testing repository

[5] To further restrict the size of this set the first file in a Not-Relevant pair must also appear as the first file in a Relevant file pair.

[6] We have also experimented with a simple learning algorithm called 1R [1] and Set Covering Machines (SCM) [2].The unsatisfactory, 1R results shows the complexity of the data. Results obtained for SCM were not significantly better than the ones obtained with C5.0.

[7] The axes in these plots are scaled between 0 and 100.

are randomly classified, with the desired probabilities, by these classifiers. To better account for the variation in the randomness, we create one accumulative confusion matrix for each classifier by repeating this 10 times. Figure 6 shows that increasing cost factors C_{NP}.and C_{PN} both increase the cost sensitive error rate, however text based classifiers perform better than both base random classifiers. The problem report based classifier generated the best cost sensitive error rates.

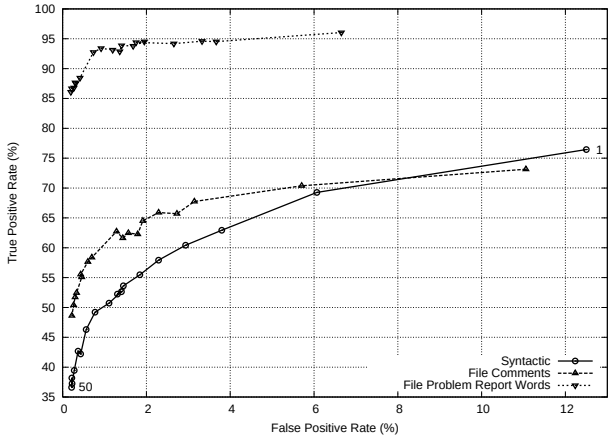

Figure 3. Comparing syntactic and text based features

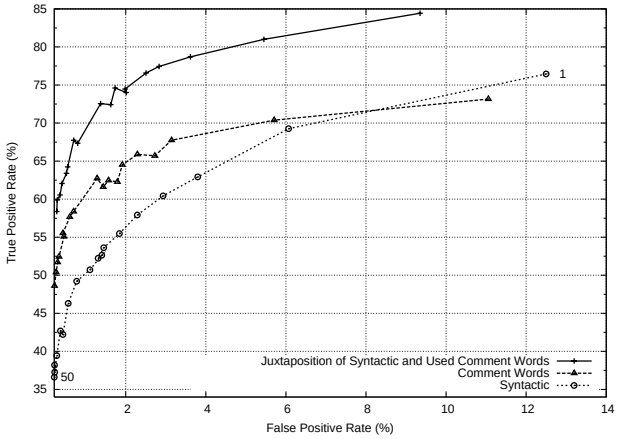

Figure 4. Combining syntactic and source file comment features

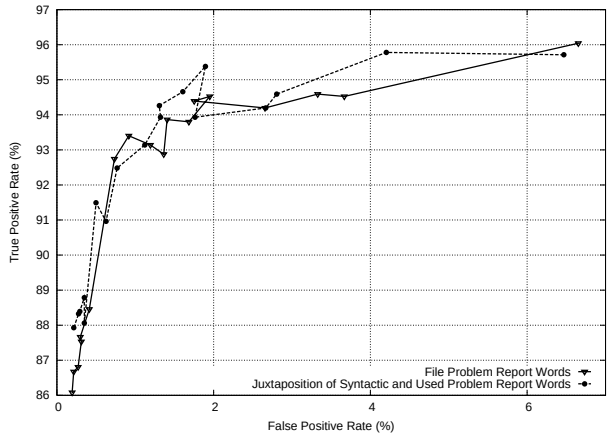

Figure 5. Combining syntactic and problem report features

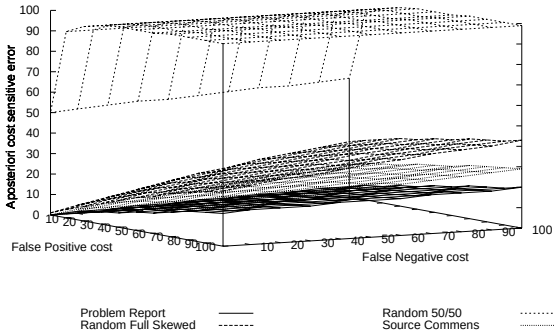

Figure 6. Cost sensitive error rate plots

8. Conclusion and future work

In this paper we presented the notion of Relevance Relation to represent relations among entities in a software system. We showed how classification learning can be used to model relevance relations. As a case study we set out to learn models for the co-update relevance relation between pairs of files in a large legacy system.

We presented results obtained from syntactic and text based feature sets, and their combinations. Our results show one can learn models with performance values that merit their practical use. We further analyzed these models under different misclassification cost assignments to evaluate their quality. The problem report based models generate some of the lowest cost sensitive error rates even in the presence of high misclassification costs. In the future we intend to experiment with other feature sets, and learn other relevance relations in software systems. We also plan to deploy the learned classifiers and evaluate their performance in the field.

References

[1] Holte R.C. 1993. Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. Machine Learning, Vol. 3 pp. 63 91

[2] Marchand M. and Shawe-Taylor J. 2002, The Set Covering Machine, JMLR, Vol. 3, pp. 723 745

[3].Quinlan J.R. 1993. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, Pat Langley, Series Editor

[4] Sayyad Shirabad J., Lethbridge T.C. and Matwin, S. 2003. Mining the Maintenance History of a Legacy Software System. Proceedings of the 19th ICSM, pp. 95-104.

[5] Zimmermann T., Weißgerber P., Diehl S., and Zeller A. 2004. Mining Version Histories to Guide Software Changes. Proceedings of 26th ICSE. (To appear).