Applying Machine Learning to Software Clustering

Susan Elliott Sim
901270630
CSC 2541F
Prof. G. Karakoulas

1 Introduction

Several branches of software engineering, such as reverse engineering, and software architecture, need to recover a design level view of a software system using the program code as the primary information source. There are many ways of representing such a decomposition pictorially, but typically a high-level view of a system usually consists of boxes representing components and edges depicting relations between these components. A component can be a source file, or a set of source files. Figure 1 is the decomposition of a small regular expression utility. The system itself is represented by the level-0 node, "regex". There are two boxes at level-1, each corresponding to a subsystem. The leaf nodes correspond to the files in the system. Note that in general, the leaf nodes do not all occur at the same level.

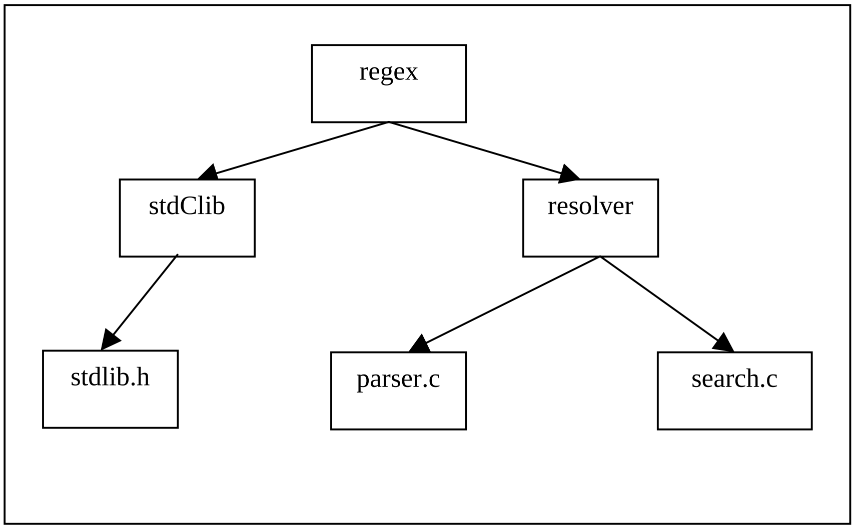

Table 1: System Decomposition

Obtaining a high-level view automatically is more important in large systems, such as those with hundreds of thousands or millions of lines of source code. In many cases, the software is has been around for a long time, has changed significantly since it was first implemented, what little documentation that exists is out of date, and the original designers are no longer available for consultation. As a result, the program code itself is the only reliable and complete source of information about the system. Furthermore, it is extremely time consuming for these decompositions to be created manually. To recover these high-level views, the source code is analyzed using *fact extractors*. These facts are normally variable use and function call relations between files. Facts are in turn analyzed to *cluster* the source files into subsystems. This clustering represents a hierarchical decomposition of the software system into components.[1]

A great deal of research has been done on developing automatic software clustering algorithms. This work has drawn on fields such as graph theory, mathematical concept analysis, data mining, and microchip design and layout. These algorithms tend to rely on structural information found in the code itself. For example, variable uses and function calls are treated as relations between different files within a software system. Mnemonics, code and data binding can also be used in clustering. Algorithms use measures of cohesion, coupling, complexity, and interface size to construct and evaluate decompositions. Wiggerts [10] and Tzerpos [9] provide good surveys of clustering techniques.

[1] See http://www.cs.toronto.edu/~vtzer/papers/hybrid.ps and http://plg.uwaterloo.ca:80/~itbowman/papers/linuxcase.html for more information on the process used by our the Holt research group to recover the design of a system.

Machine learning algorithms are among those that have been applied to the problem of creating clusters. Merlo, McAdam, and De Mori [5] have used artificial neural networks (ANNs) to recognize clusters. By using comments and identifier names in source code as inputs, the problem become very similar to attributing articles to newsgroups. Greater success was met by Schwanke and Hanson [8] who trained ANNs to group together similar (according to distance measure) files into clusters. Mancoridis et al. [4] used traditional hill-climbing and genetic algorithms to create clusters with promising results.

These tools are still experimental and have met with varying degrees of success. There is no general-purpose clustering algorithm that will work with any system. Each clustering algorithm works well with a particular programming paradigm or the architectural style of the system. These tools function best as an aid to a human constructing a clustering. In their current state, they are not be used to cluster a large system independently. As with many other knowledge-intensive tasks, there is no good substitute for a person with a great deal of knowledge of the software system.

Although, machine learning has been applied to clustering as an *unsupervised* learning problem, it has not yet been attempted as a *supervised* learning problem. With this reformulation, a learner would be asked to learn a classification, i.e. the decomposition, from a number of examples. The goal of in this task is to reduce the amount of human input required, rather than eliminate it entirely. The time required for such a task can be considerable, when a software system has on the order of 500 000 lines of code in 1000 files. It would save a great deal of effort if someone were only required to categorize a subset of these files.

There are a number of advantages of taking this approach. One, the problem takes on the structure of a classic machine learning problem: learning a concept inductively and generalizing this knowledge to other examples. Two, this approach fits with ones currently in use in reverse engineering. Human experts are usually consulted extensively when creating a clustering. With machine learning, the number of labels required can be reduced. Three, the programmers who use these decompositions are more likely to trust them if they were involved in the creating them.

This paper reports on some experiments undertaken using two learners, Naïve Bayes and Nearest Neighbour, on three software systems, the C488 compiler used in CSC 488 undergraduate course, the Linux Operating System Kernel, and an optimizing back-end from IBM. The best results were found with the Nearest Neighbour learner on the Linux kernel and IBM system, with accuracy rates of 96.0% and 93.9% respectively. Further experiments were conducted to evaluate the performance of this learners in greater detail.

2 Software Clustering as a Supervised Machine Learning Problem

As mentioned in the previous section, the goal of applying machine learning to the software clustering is to reduce the effort required from domain experts to recover the high-level design of software system. This can be accomplished by training a learner on a set of examples and have the learner to generalize that learning to other cases. The examples are the files in the system. The concept to be learned is the decomposition of the software system, which means the classification is the subsystem containing the example. So, given a file we would like the learner to tell us what subsystem it belongs in.

Clustering presents a number of unusual challenges as a supervised machine learning problem. These characteristics need to be taken into consideration when designing a learning system.

- The universe of cases is available and requires classification.

Unlike typical machine learning problems all of the instances that need to be recognized are available to the learner. As a result, probabilities and distributions can be calculated over true values, rather sample estimates. Furthermore, all of these instances are available at the start of learning.

- The cost of obtaining labels is high, relative to obtaining values and features.

Labels need to be generated by a domain expert. In contrast, adding features and values can be done by using analysis programs and scripts. Although, some features and values also need to be generated by hand, they may not be necessary for successful learning.

- The clustering is hierarchical.

As illustrated in Figure 1, the decomposition of a system is hierarchical. For example, the file parser.c belongs in both the resolver subsystem and the regex subsystem. Consequently, a completely learned concept would need to take this structure into consideration.

- Some classifications and features can be sparse.

At the lower levels of the decomposition tree, a subsystem may contain a small number of files. As a result, the number of examples for a given classification can be relatively small. Similarly, some of the nominal features will have a large number of distinct values, each occurring a relatively small number of times. The features to be used will be discussed in the next section.

- The cost of being wrong is greater than the cost of failing to provide a classification.

When the learner provides a classification, it should do so with a relatively high degree of certainty. It is more difficult to detect that a file is in the wrong cluster and correct it, than it is to provide a label for a file when prompted by the learner.

For the purposes of this project, a simplifying assumption will be made: the learner need only provide the level-1 subsystem as a classification for a file. This decision was made for two reasons. Once this top level categorization has been made, the lower level ones can be performed recursively. Furthermore, these subsystems have a large number of instances within them, thus improving the likelihood of a correct classification.

2.1 Software Systems

The Holt research group has analyzed the architecture of a number of software systems. Three software systems were used in the experiments. We have had fair amount of experience with these systems and their decompositions are well understood. As a result, there is there are both structural and mnemonic facts can easily be converted into features for learners. These systems present a broad spectrum of structural and mnemonic characteristics, and will serve to test the performance of the learners under different conditions.

2.1.1 C488 Compiler

The C488 compiler is a toy program written in C and used in the CSC 488 undergraduate course in compilers. There are 37 files in the system and 11 level-1 subsystems. The distribution of these files are given in the table below.

Subsystem	Number of Files	Percentage of Files
stdCLib.ss	11	29.72
parser.ss	6	16.22
codegen.ss	4	10.81
semantics.ss	4	10.81
machine.ss	3	8.11
shared.ss	3	8.11
scanner.ss	3	8.11
symbol.ss	2	5.41
main.ss	1	2.70

Table 2 : Distribution of Files in C488

The Standard C Library subsystem has the largest number of files, meaning of the learner simply guessed this answer, it would be correct about 30% of the time. It should be noted that the files in this subsystem are easily recognizable both structurally and mnemonically. Structurally, library files are distinct in that they are used by many files, yet they do not use any files themselves. Mnemonically, they are distinct in that they are not found in the same directory as files in any of the other subsystems. Hence, the directory structure of the C488 compiler is significant, as well as some of the names of the files. This system also has a very flat containment hierarchy, all the classes are located in level-1 of the decomposition and all the files are on level-2.

Because the C488 system is small, it is useful for experimenting with new ideas. In contrast, the Linux Kernel and the IBM Back End are more typical of industrial software with significant functionality and a large user base.

2.1.2 Linux Operating System Kernel

The Linux Operating System Kernel was developed using the Open Source Software model. The Holt group and a software architecture graduate course at the University of Waterloo has spent a great deal of time studying this system. [2, 3] Its containment tree is quite full and complex. The Kernel is written primarily in C. The directory structure provides valuable information about the clustering, however, in some cases it can be misleading because all of the header files are placed in the same directory.

Subsystem	Number of Files	Percentage of Files
fs.ss	511	51.93
net.ss	247	25.10
lib.ss	102	10.37
sched.ss	50	5.08
mm.ss	30	3.05
ORPHAN.ss	22	2.24
ipc.ss	14	3.66
init.ss	8	0.81

Table 3 : Distribution of Files in the Linux Kernel

There are 984 files and 8 categories. The largest level-1 subsystem is fs.ss, the file system subsystem, as it contains all the code for actual and virtual device drivers and includes almost 52% of the files. The next largest is net.ss, which contains the source code network protocols and connections.

2.1.3 IBM Compiler Back End

The IBM Compiler Back End is an important component in many significant IBM compiler products. It is written in PL/IX, an IBM PL/I variant, and is maintained by a team at the Toronto Lab. There are 1016 files in 4 level-1 categories. In this system, all the files are kept under configuration management, but essentially in a single directory. The different subsystems are denoted by prefixes and suffixes in the filename.

Subsystem	Number of Files	Percentage of Files
optimizer.ss	381	37.50
assembler.ss	299	29.43
allocator.ss	268	26.38
assembler.ss	68	6.69

Table 4 : Distribution of Files in the IBM System

Within each of these level-1 subsystems, there are about a hundred smaller ones. The most frequently occurring subsystems in the Back End have about 5 files in them.

3 Learners and Representations

Based on previous experience and a survey of the clustering literature, both structural and mnemonic information were selected as inputs to the learners. The mnemonic information used were the directory path of the file, and substrings from the filename. Substrings were chosen because filenames themselves tend to be unique. Furthermore, substrings can capture the naming conventions used in the software system. Anquetil and Lethbridge found in their study that better clustering results were obtained when overlapping substrings of length 3 were used.[1] To minimize the amount of spurious information provided to the learner, the first three characters and the last three characters of the filename were selected as features.

Many kinds of structural information, i.e. relations between files, can be extracted from a software system. The three most general ones are: variable uses, function calls, and implementation of declarations. A variable-use between two files occurs if file X uses a variable or variable type defined in file Y. A function-call between two files occurs if file X calls a function defined in file Y. Both of these relations can be established during pre-processing, compilation, or linking of a program. An implement-by relation occurs if file X implements a function declared in file Y. Normally, in C, file Y is a .h file and X is a .c file. The difficulty in representing this information to a learner is that a file can have an arbitrary number of relations to other files. Furthermore, it can have more than one type of relation to a particular file. How this difficulty is resolved will depend on the learner.

In summary, features in the cases consisted of: path name, substrings from the file name, variable-use relations, function-call relations, and implement-by relations.

3.1 Naïve Bayes

The Naïve Bayes learner essentially asks the question, “Given data seen during training, what is the most likely classification for this example?” This classification is called the *maximum a posteriori* hypothesis. It builds up this hypothesis by calculating the distribution of the values of features relative to their classifications. The NewsWeeder version of the Naïve Bayes classifier was selected because it allows variable length input examples.[6, section 6.10] As a result structural information can be easily represented to this learner.

A relation was represented by using a prefix to indicate its type, followed by the fully qualified name of the file. For example, if file X used a variable from file Y, it would be represented to the learner as “usevar-/path-of-Y/Y”. Including the path ensures that every distinct file has a unique string associated with it. A similar convention was used with the function call and implement relations.

Two different representations were used with the Naïve Bayes learner. The first one treated the path name a single string. The second separated the path name into subdirectories names.

3.2 Nearest Neighbour

Nearest Neighbour is an instance-based learner. Rather than build a model of the hypothesis space, it stores all the training examples and calculates the answer when presented with test example. The Nearest Neighbour learner uses a distance function to find the training example that is most similar to the test example, and extrapolates the classification. Example that are similar have small distances between them.

Two different distance functions were used: HEOM and HVDM.[11] Although both functions are able to handle both nominal (categories) and linear discrete (numeric) data types, the examples contained only the former. As a result these functions will only be discussed in terms of how they handle nominal data. The Heterogeneous Euclidean-Overlap Metric (HOEM) looks at whether the values of a feature *overlap*. It looks at each feature, if they are the same in both examples the metric gives a score of 0, otherwise it is given a score of 1. The Heterogeneous Value Distance Metric (HVDM) considers two values to be closer if they have a larger number of classifications in common. For each feature, it outputs a number between 0 and 1, representing the proportion of the time the values in the two examples have the same classification.

One important characteristic of Nearest Neighbour is both examples need to have the same number of features. This constraint was resolved by making each relation type a single feature. The values for these features was a list of fully qualified file names. The distance functions needed to be modified to handle a "list" data type. A distance metric is suggested by Wiggerts.[10] Every element in the two lists should be categorized according to the matrix below.

		List j	
		present	absent
List I	present	a	b
	absent	c	d

The Simple matching coefficient is: $\frac{a+d}{a+b+c+d}$, which gives a number between 0 and 1. The form of this coefficient is consistent with the other components of the heterogeneous distance functions and can simply be added to the distance calculation.

4 Results

Two series of experiments were conducted with the learners. In the first series, the four versions of the learners were evaluated using a cross-validation method on the three software systems. In the second series, considers the effect of reducing the size of the training sets on the accuracy of the Nearest Neighbour learner with the Linux Kernel and IBM software. In general, the learners performed surprisingly well.

4.1 Experiment 1: Performance Under Cross-Validation

A 10-fold cross-validation was used with Linux and the IBM Back End. A 5-fold cross validation was used with theC488 system because there is a much smaller number of examples available. These results are summarized Table 5.

With C488, there were 30 cases in the training set and 7 cases in the test set and the best accuracy rate was obtained with the Naïve Bayes learner using the second representation in which the path was separated into subdirectories. This learner was able to classify 91.4% of the test cases correctly, which is considerably better than guessing the largest subsystem, stdCLib.ss, which contains 29.72% of the cases.

For Linux, there were 886 training cases and 98 test cases. The best learner with this system was Nearest Neighbour using HOEM, which had a success rate of 96%. The largest subsystem in Linux, accounts for almost 52% of the cases.

The best learner for the IBM Back End was also Nearest Neighbour using HVDM. There were 914 training cases and 102 test cases, which produced an accuracy rate of 93.9%. Even though there were fewer classes in this system than Linux, its largest subsystem only accounted for 37.5% of the files.

		Learner			
		Naïve Bayes 1	Naïve Bayes 2	NN – HEOM	NN – HVDM
SW System	C488	80.0%	**91.4%**	88.6%	88.6%
	Linux	87.9%	86.6%	**96.0%**	95.9%
	IBM	89.2%	89.0%	**93.9%**	93.8%

Table 5 : Summary of Accuracy Rates

For each software system, the confusion matrix for its best learner was examined. Each cell in the confusion matrix contains two numbers. The top one is the number of cases that fell into the group. The second one is the average value of the metric used by the learner to select the classification. For Naïve Bayes 2, the metric is the normalized probability of the guessed classification. For Nearest Neighbour, the metric is the distance to the most similar training case. These metrics were included to determine if any a threshold could be set for classification confidence.

The confusion matrix for C488 indicates the learner classified 3 files incorrectly. There does not appear to be any systematic errors. In two of the three cases, the learner guessed the semantics.ss subsystem. One of these misclassifications was the single file from the main.ss subsystem. This error is not surprising as the learner had not seen an example from that classification before. There also does not seem to be a reasonable threshold for the p-value that would not sacrifice performance. Many of the correct classifications are made with very small p-values and the p-values of misclassifications are not consistently smaller.

count p-value		Guess By Learner								
		codegen.ss	machine.ss	main.ss	parser.ss	scanner.ss	semantics.ss	shared.ss	stdClib.ss	symbol.ss
Actual Classification	codegen.ss	3 0.60	-	-	-	-	-	-	-	-
	machine.ss	-	2 0.75	-	-	1 0.85	-	-	-	-
	main.ss	-	-	-	-	-	1 0.38	-	-	-
	parser.ss	-	-	-	6 0.71	-	-	-	-	-
	scanner.ss	-	-	-	-	3 0.72	-	-	-	-
	semantics.ss	-	-	-	-	-	3 0.68	-	-	-
	shared.ss	-	-	-	-	-	1 0.25	2 0.42	-	-
	stdClib.ss	-	-	-	-	-	-	-	11 0.86	-
	symbol.ss	-	-	-	-	-	-	-	-	2 0.40

Table 6: Confusion Matrix for Naïve Bayes 2 on C488

In the confusion matrix for Linux, there are two patterns. Files from the lib.ss and fs.ss subsystems are easily mistaken for each other. These two cells, shaded in gray, have the largest counts of any of the misclassification cells. The other pattern is a difference between the average distance of correctly and incorrectly classified cases. The average distance of the correctly classified cells is no more than 3.24. Except for one cell, the average distance of the incorrectly classified cells is no less than 3.97.[2] Although this gap is suggestive, an examination of the raw scores indicates that there are still more correctly classified cases than incorrectly classified ones above this gap. Therefore, it would not be worthwhile to set the threshold here. On the other hand, the value 5 could be used an extremely conservative threshold as there are no correctly classified instances with a distance greater than this number.

count distance		Guess by Learner							
		ORPHANS.ss	fs.ss	init.ss	ipc.ss	lib.ss	mm.ss	net.ss	sched.ss
Correct Classification	ORPHANS.ss	17 3.23	1 4.97	-	-	1 4.00	-	-	-
	fs.ss	1 2.99	497 3.17	-	1 3.99	6 4.15	1 4.93	2 3.96	-
	init.ss	1 5.93	2 5.91	7 3	-	-	-	-	-
	ipc.ss	-	2 3.99	-	12 3.16	-	-	-	-
	lib.ss	-	5 3.97	1 3.99	-	91 3.09	-	2 3.99	1 4.00
	mm.ss	-	-	-	-	-	29 3.24	-	1 4.00
	net.ss	2 3.99	-	-	-	1 4.97	-	243 3.17	2 4.48
	sched.ss	1 3.99	1 3.97	-	1 3.98	2 3.98	-	-	46 3.15

Table 7: Confusion Matrix for NN-HEOM on Linux

[2] During previous clustering attempts, any files that could not be allocated confidently were placed in the ORPHANS.ss subsystem. Consequently, the file from this subsystem that was classified as belonging to the fs.ss subsystem with a distance value of 2.99 is likely a correct classification. It is gratifying to find that even these small experiments were able to improve an existing cluster.

For the IBM Back End, files from the optimizer.ss are misclassified more often than ones from the other three subsystems. These cells are shaded gray. A gap between the average distances of the correctly and incorrectly classified cells that is similar to the one found with Linux also occurs. Therefore it is also difficult to set a useful threshold for this system as well.

count distance		Guess by Learner			
		allocator.ss	assembler.ss	optimizer.ss	services.ss
Correct Classification	allocator.ss	247 3.13	-	6 3.82	8 3.86
	assembler.ss	-	66 3.06	1 3.98	2 3.49
	optimizer.ss	16 4.08	-	367 3.15	11 3.99
	services.ss	4 3.74	2 3.49	7 3.98	279 3.05

Table 8: Confusion Matrix for NN-HEOM on IBM Back End

Although these results are quite good, the rely on having are large proportion of the files already labelled. Requiring 90% of 1000 files to be classified before using a machine learner is not a big improvement over clustering by hand. In the next section, an experiment on the effect of reducing the size of the training sets is discussed.

4.2 Experiment 2: Effect of Reducing Training Set Sizes

This subsection reports on additional experiments were done with Nearest Neighbour – HEOM on the Linux and the IBM Back End. The Linux and IBM Back End were used because results with these software systems are more indicative of legacy systems. Consequently, the Nearest

Neighbour learner was also selected, Although the performance of the HOEM distance measure was only slightly better than HVDM, it runs much faster, so it was selected for further testing.

Four different train/test split sizes were used: 80/20, 60/40, 40/60, and 20/80. For each train/test split size, five sets were randomly generated. Although procedure results in test sets that are not independent, it would be difficult to do otherwise. The results are shown in Figure 2 below.

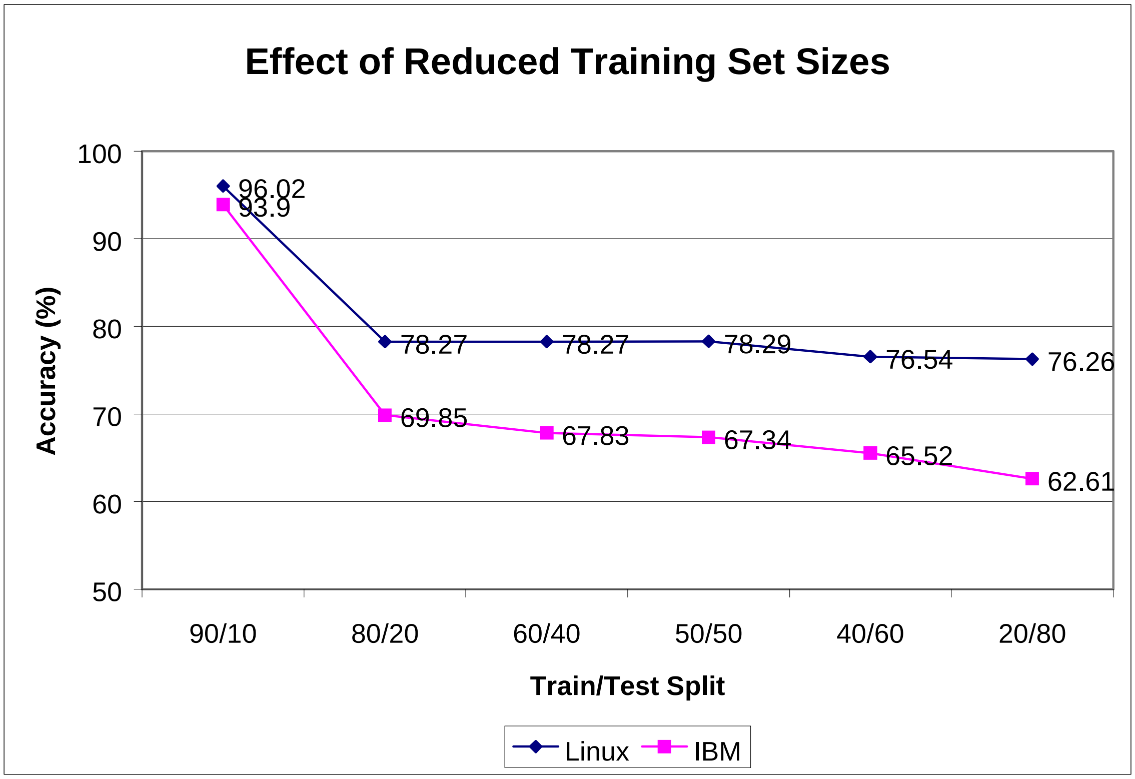

Figure 2: Accuracy Rates of NN-HEOM

The graph shows that there is a large drop in accuracy when the training size is from 90% to 80%. The accuracy with Linux decreased 17.75 percentage points and with IBM Back End it decreased 24.05 percentage points. However, the decrease in accuracy in the remaining reductions are relatively small. Reducing the training size from 80% to 20% only decreases the accuracy by 2.01 percentage points for Linux and by 7.24 percentage points for IBM Back End.

It is somewhat surprising that the IBM Back End is affected by the reduced training test sizes so much more than the Linux Kernel. This trend is counterintuitive because IBM Back End has fewer classifications. One possible explanation is that there are too few classifications. Each level-1 subsystem actually encompasses many disparate subsystems, which means each class groups together many files that are dissimilar. Perhaps it would be possible to obtain better results, if there were more classes with a sufficient number of files that are more similar to each other.

There is a variant of the Nearest Neighbour learner, k-Nearest Neighbour, which, as the name implies, uses the majority vote of the k nearest neighbours to determine the classification of a test case. The exact number of neighbours, k, is tuned during training using the leave-one-out method. This algorithm was not used because the experiments would have taken a prohibitively long time to complete. However, this learner may perform better with the reduced training sets. In the next section, other improvements and applications are discussed further.

5 Conclusion

The experiments in this study represent a promising start in considering software clustering as a supervised machine learning problem. The initial results using the Naïve Bayes and Nearest Neighbour learners are very encouraging. The decomposition of three software systems used, C488, Linux Kernel, and IBM Back End, were learned with accuracy rates of 91.4%, 96.0%, and

93.9%, respectively. When the size of the training set is reduced from 90% to 80%, there is a large drop in accuracy, but the decrease from 80% to 20% is very flat. The most significant contribution of this approach is that it is a general purpose clustering algorithm, something that has not yet been achieved in software engineering. Given these early successes there are many applications and improvements that are worth considering.

5.1 Applications

Given that the learner performs best when it has a large number of labelled examples, this machine learning approach may be useful in some problems related to constructing a clustering. One such problem is clustering new files that have been added to the system. This incremental clustering may occur as individual files are added, or it can be done in batches, such as when a new version of the software is released. The established clustering would be used as the training cases and the new files would be the test cases. The capability is particularly useful for keeping a the design level views of the software up-to-date and reducing the necessity of re-extracting the architecture.

“Kidnapping” is an exercise performed to test a clustering. It involves removing a file from a subsystem, or kidnapping it, and asking the clustering algorithm to place it in the most appropriate subsystem.[3] This test is usually performed with decompositions created by algorithms that treat the system as a whole, such as those that graph theoretic concepts.

[3] In fact, the machine learning approach could be considered as a large kidnapping experiment in which a fraction of the files are removed, instead of just one.

A different, but related problem, would be suggesting a name for new file within a subsystem. Naming conventions may be complex, or they may be forgotten over time, or a new programmer may simply be unaware of them. Putting these rules into a program would be a form of self-documentation. This tool could be used just before a programmer checks in a new file into the configuration management system. It would look at the file name and its structural relations to determine whether the name follows the naming conventions of the system. If the name fails, a modification could be suggested.

5.2 Future Work

There are many ways that the learners could be improved. A fairly simple modification to Nearest Neighbour would be to add weights to the different features in the calculation of the distance function. Weights could even be added to the calculation of the Simple matching coefficient for lists. There is a variant, the Jaccard coefficient, which takes the Simple coefficient and gives d a weight of 0. This values of the weights would be determined using a combination of prior knowledge and trial-and-error, and will likely need to be set for each software system.

Another improvement would be to have the learner request classifications or labels for files which will provide the greatest amount of information. Recall from Section 2, that the learner has access to all the examples before training begins. The learner could use this information to calculate prior probabilities and distributions. Nigam and McCallum suggest an approach that uses Expectation Maximization (EM) and Query-by-Committee (QBC). [7] A 20% training set that is chosen by the learner with knowledge of its internal state would likely be more useful than a 20% training set chosen randomly or by a person

Currently the learners do not require any particular knowledge of the software system to learn the clusters. While this is a strength, their performance may be improved by adding domain knowledge. For example, if it is known that files that use a particular prefix in their name all belong to the same subsystem, it would be useful for the learner to have access to rules such as these. There are two ways for such rules to be used. They could be encoded into each case by adding features. Alternatively, a rule based learner could be used and combined with the ones used in this present study. Learners could be combined using stacked generalization. Since there are three types of information available, mnemonic, structural, and rule-based, a different learner could be trained on each type of information, and the results combined.

6 References

[1] N. Anquetil and T. Lethbridge, “File Clustering Using Naming Conventions for Legacy Systems,” presented at CASCON'97, Toronto, Canada, 1997.

[2] M. Armstrong and C. Trudeau, “Evaluating Architectural Extractors,” presented at Working Conference on Reverse Engineering, Honolulu, HI, 1998.

[3] I. T. Bowman and R. C. Holt, “Linux as a Case Study: Its Extracted Software Architecture,” presented at Twenty-first International Conference on Software Engineering, Los Angeles, CA, 1999.

[4] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, “Using Automatic Clustering to Produce High-Level System Organizations of Source Code,” presented at International Workshop on Program Comprehension, Ischia, Italy, 1998.

[5] E. Merlo, I. McAdam, and R. D. Mori, “Source Code Informal Information Analysis Using Connectionist Models,” presented at International Joint Conference on Artificial Intelligence, Chambery, France, 1993.

[6] T. M. Mitchell, *Machine Learning*. Boston, MA: WCB/McGraw-Hill, 1997.

[7] K. Nigam and A. McCallum, “Pool-Based Active Learning for Text Classification,” presented at Conference on Automated Learning and Discovery (CONALD), 1998.

[8] R. W. Schwanke and S. J. Hanson, “Using Neural Networks to Modularlize Software,” *Machine Learning*, vol. 15, pp. 137-168, 1994.

[9] V. Tzerpos, “Software Botryology: Automatic Clustering of Software Systems,” University of Toronto, Toronto, Depth paper March 20, 1998.

[10] T. A. Wiggerts, “Using Clustering Algorithms in Legacy System Remodularization,” presented at Working Conference on Reverse Engineering, Amsterdam, The Netherlands, 1997.

[11] D. R. Wilson and T. R. Martinez, “Improved Heterogeneous Distance Functions,” *Journal of Artificial Intelligence Research*, vol. 6, pp. 1-34, 1997.

Appendices: Source Code for Naïve Bayes and Nearest Neighbour