DIVISION OF COMPUTER SCIENCE

Clone Detection in Telecommunications Software Systems: A Neural Net Approach

S. Carter
R. J. Frank
D. S. W. Tansley

Technical Report No.208

December 1994

Clone Detection in Telecommunications Software Systems: A Neural Net Approach

S. Carter, R.J. Frank*, D.S.W. Tansley

BNR Europe Limited
London Road, Harlow, Essex, UK, CM17 9NA
`sc@bnr.co.uk, dswt@bnr.co.uk`
* *University of Hertfordshire*
College Lane, Hatfield, Herts, UK, AL10 9AB
`comqrjf@herts.ac.uk`

Abstract

We report on the development of a tool which uses neural nets to help in the detection of 'clone' software in large telecommunication software systems. Cloning is a primitive form of software re-use, whereby existing blocks of source code are copied and adapted, for use elsewhere in the system or to implement new functionality. A number of representation schemes for presenting the source code to Self-Organising Maps (SOMs) have been tried. The more favourable of these have been applied in a hybrid, modular tool architecture. The current result is a demonstration system in trial.

1 Background

1.1 Work context

The authors were given the opportunity to start to evaluate the costs and benefits of the much-hyped neural network technology in late 1992, following years of low-level background monitoring of the field by the Software and Systems Engineering group within the Advanced Technology Centre at BNR Europe Limited, Harlow, UK. Our aim was to design a programme of work to be carried out in 1993 to demonstrate the capabilities of the technology in a useful way to the company. It was decided that this should be done by developing a software source code clone detector.

1.2 Clones and cloning

The clone 'problem[1]' arises from a simple, frequently-used technique to increase or amend the functionality of telecommunication systems, notably those large, older systems from a time when software development techniques were relatively primitive compared to those we have today. It involves copying existing sections of source code and changing the copy as necessary in order to implement the new functionality. Whilst this should result in copying known-to-be-working code, it also copies as-yet-unknown bugs, and may often include large amounts of redundant statements. Software cloning is a major contributor to the steadily increasing size, complexity, and increased error rates of such systems.

The choice for telecommunications companies is stark – either build new systems from scratch using modern technology, or re-engineer the existing systems. If re-engineering is the choice, a major part of the problem will be solved if the clone software present in the system can be managed – either eradicated or tracked. Similarly, new clones should either be prevented or logged to aid future bug-fixing. To do this practically, an automatic tool is required – a clone detector. Since clone detection inevitably involves pattern matching, a neural net approach would seem to offer some promise.

There are a number of different types of clone to be found in telecommunications software. By different types, we mean degrees of how far away from some original piece of software a cloned piece of code is. Thus, one can describe clones using the following typology:

- Type I – An exactly identical source code clone, i.e. no changes at all.
- Type II – An exactly identical source code clone, but with indentation, comments, or identifier (name) changes.
- Type III – A clone with very similar source code, but with small changes made to the code to tailor it to some new function.
- Type IV – A functionally identical clone, possibly with the originator unaware that there is a function already available that accomplishes essentially the same function.

[1]There are actually both positive and negative aspects of software cloning.

Whilst a neural net approach could be used to address any of these types, currently we are addressing the problem of types II and III, which probably form by far the majority of clones present in existing systems. Future work may address type IV. (Type I is trivial to detect.)

Finally, in terms of level of scale, at the current stage of our work, we are looking for clones at the 'procedure' or 'PROC[2]' level. (This is analogous to the level of 'functions' in the C programming language.)

2 A Neural Network Solution

The problem of detecting clones is to do with taking a large data set and identifying groups of closely-related sections of that data, i.e. a classification problem. The data set is some range of the total set of source code procedures for a system that we are concerned with. By 'closely related', we mean 'closely related in terms of being candidate clones'.

A solution emerged in the form of the well-known 'Self-Organising Map' (SOM) [1] [2] . This performs a classification using an unsupervised training phase. It had the additional advantage of preserving any topological relationships between the input vectors. This would be useful in our problem domain as it might be used to measure some degree of 'cloniness' between any specific classes of clones identified by the network.

3 The Input Vector

The mapping of the raw source code into a form suitable for inputting to the network is a major area for our investigation. Preliminary discussions generated the following possibilities for abstracting the raw source code into its specific features: call graphs, parse trees, tokens (e.g. names, identifiers, etc.), visual patterns (e.g. indentation, characters represented as single bits, etc.), data structures, and comments. The input to the net would be some coded form of these. Combinations of the above are also possibilities.

The precise form of the input would be dependent upon the specific type(s) of network used. Our early experiments concentrated on two features of the source code in particular: physical layout patterns, keyword and operator frequency distribution, (and also combinations of these two). Although first attempts, these have actually proved successful enough to not warrant the trying of other features at this stage.

Figure 1 illustrates how a typical input vector is constructed.

[2]A statement in BNR's proprietary programming language – but note that our work is *completely* language-independent.

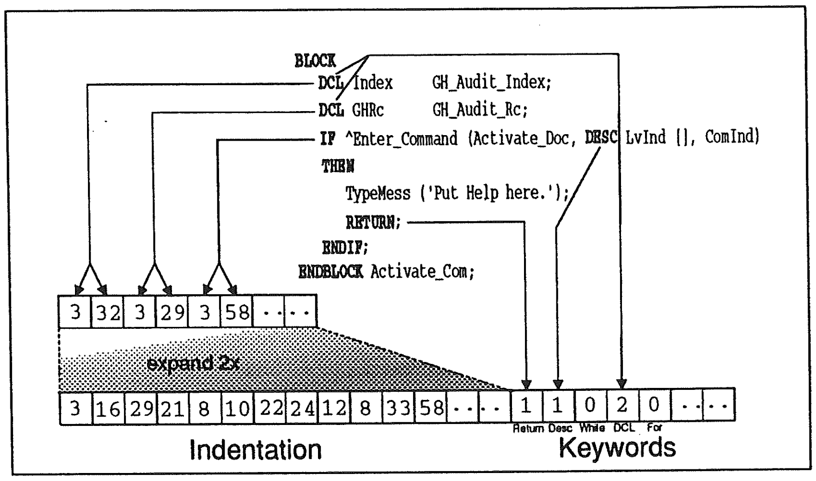

Figure 1: Schematic of input vector construction

The first type of source code feature we used was the 'physical' way the source code was layed out, i.e. how it was indented on the page, how many lines were present, how long the lines were, etc. There are various possible combinations of these elements to describe the feature of the way the code is laid out, depending on the level of detail required and the relative importance of each element. For example, we tried representing only the changes in indentation as a binary pattern vector and we tried representing the changes and the length of similarly indented blocks.

A significant problem with this approach was that the vector varied in length depending upon how many changes in indentation there were and/or how long the procedure was. We used padding and truncation to address these problems initially, in order to get a fixed input vector for the SOM, but later moved to a scaling algorithm to pre-process the data to always produce a fixed-length vector. The scaling algorithm worked by using the values in the original vector to define points on a graph with straight lines joining those points. The new vector was produced by sampling along this graph at the required interval to find the values for the new vector. This technique has the advantage that operations such as adding or removing a line of source code would result in similar vectors.

The left-hand side of figure 1 shows how a real vector (part) is constructed for indentation (and line length). For each line of code, the number of characters of indentation is one value of a data pair (e.g. 3); the other value is the length of the line (e.g. 32). In the example, a x2 expansion is required to scale the raw data vector to the fixed-length input vector.

The second type of source code feature we used was the frequency distribution of the programming language's keywords used in the procedures. Keywords were such things as 'RETURN', 'DESC', 'WHILE', 'IF', 'THEN', etc. There were 77 different keywords, although some of these could be removed due to rarity of use or redundancy (due to them always appearing in pairs, for example), leaving 62 meaningful ones. The keyword input vector (part) thus simply consisted of one fixed-length array per procedure, with each element of the array containing a number representing the count of occurrences of a particular keyword in the current procedure – see right-hand side of figure 1. Additionally, we added counts of the 19 different operators in the language, giving a total input vector size of 81 in this case.

4 The Output Vector

The output of a SOM is inherently coupled to its Kohonen layer – in fact, it is the pattern of activity in that layer itself. When a vector is applied to the trained network, one neuron becomes active in the Kohonen layer. Each neuron in the Kohonen layer represents a class, so the active neuron indicates the class that the input vector falls within. This raw output is not suitable in a clone detector tool. The sort of output required in such a tool is a list of 'candidate clones', perhaps with degrees of likelihood or 'cloniness' against each, compared to themselves or a 'reference procedure'. This output information can easily be obtained from the SOM with a little post-processing, however, and its topology-preserving characteristic helps to make a 'cloniness' measure more easy to calculate.

5 Experimental Results

These results were produced using input vectors from 1775 procedures extracted from a skim file of approximately 5Mbytes of arbitrarily-selected source code. A SOM was trained using these vectors. At the end of training the output from the SOM was further processed to populate a clone database. Different experiments were done using combinations of various sized SOMs and different input vectors (see Section 3).

The clone detection rate was determined by randomly choosing 100 test procedures from this data set and using the clone database to access the nearest procedure to it, subject to a cloniness threshold. The two procedures, test and candidate clone, were viewed side by side by a software engineer familiar with the type of source code who then decided whether the neighbouring procedure was or was not a clone of the test procedure. A numerical value was then arrived at for each test by examining the result for each procedure in the test and adding the cloniness value if the procedure was a clone and subtracting the cloniness value if it was not a clone. The final result was then scaled to be in the range 0 to 100. Note that this is therefore *not* a percentage measure of accuracy!

Table 1 summarises the key experiments performed with the different input vector schemes described in Section 3.

Table 1: Input Vector Type vs Kohonen Layer Size Results

Input Vector Scheme	**Kohonen Layer Size**	**Results**
Keyword/Operators	10x10	35.2
Keyword/Operators	15x15	36.0
Keyword/Operators	20x20	37.0
Indentation	10x10	41.3
Indentation	15x15	51.4
Indentation	20x20	55.0
Keywords/Operators/Indentation	20x20	68.7

The results show that as the number of units increase in the Kohonen layer, the categorisation becomes more accurate.

The results also show that although the keyword and indentation techniques both produce reasonable results, the combination of the two produces a better result, as more information about the procedure is provided to the net and there is a smaller likelihood that two procedures that were not clones of each other would have similar input vectors.

6 The Neural Network Clone Detection Tool

Figure 2 shows a schematic view of the proposed tool architecture and mode of operation. This is described below.

The proposed tool will be required to operate with a much larger environment (100k+ procedures) compared to the development and test environment. We plan to use a two level hierarchical neural network based on the SOM paradigm. The first level SOM will be trained on all the procedures (in some suitable subset of the entire system) to classify them into a relatively small number, say n, classes.

A set of n separate second level SOMs will then be trained on the procedures in each class to discriminate these broad classes into smaller sub-

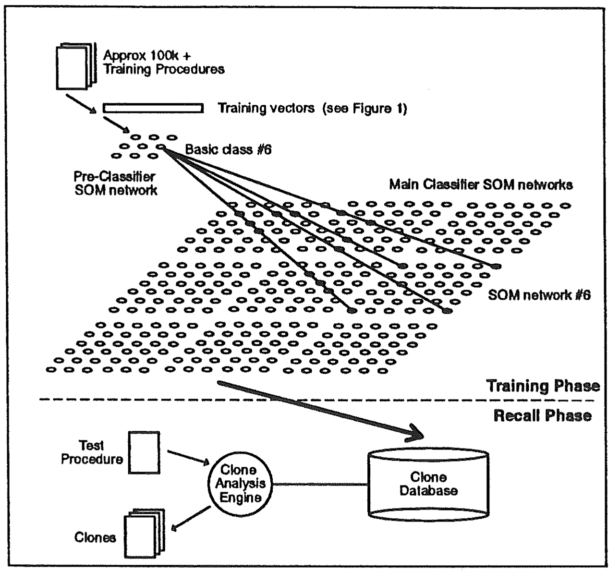

Figure 2: Schematic of clone detection tool

classes. The output of this neural network hierarchy would then be processed into a cloned procedure database. This would be accessed by the clone analysis component of the tool in a recall phase, once the training had populated the database.

The clone detection tool would find the clones of a given procedure by accessing the class and subclass of a procedure from the neural network output database and calculating the closeness of the given procedure to the other procedures in the sub-class and neighbouring sub-classes.

At this point the closeness measure used may be based upon the textual differences of the source code, procedure names, etc. The user can inspect any of the candidate clone procedures against the test procedure in text windows placed side by side on a workstation.

This mode of operation of the tool will be useful in debugging where the cloned procedures of a procedure with a known error can also be discovered, allowing an increase in efficiency of this activity.

Another possible mode of operation of the tool would be in providing a statistical analysis of large parts of the system. This would constitute a more time-consuming operation but would provide system development managers with an overall view of the state of the system from the point of view of code duplication, development effort, etc.

The tool could also be used as an aid in producing a reusable software library for use in later generations of the software.

7 Conclusions

A demonstration tool has been produced that will take the raw output from a trained SOM, together with the training vectors, in order to produce an ordered list of likely clones of any given procedure. It will then present windows on a screen which display the source code containing the likely clones in order for the user to inspect them manually.

The work as it currently stands shows that the best performing network is one that has the order of a 20x20 Kohonen layer, using an input vector that encodes a combination of source code features including indentation pattern, keyword and operator frequency, and the procedure length.

A clone detector product will use a hierarchy of SOMs which will enable us to manage a realistic amount of source code. This will result in an extremely powerful and uniquely flexible tool. The clone detector will be just the start of a possible portfolio of reverse engineering and design recovery tools that could use components based on neural networks.

Acknowledgments

We would like to thank Neil Davey from the Department of Computer Science at the University of Hertfordshire, UK, for his contributions to the work presented in the paper in his role as consultant.

We also acknowledge the financial assistance of the Department of Trade and Industry of the UK government, who have partly funded the placement of Ray Frank on a 'Senior Academics in Industry Scheme' (SAIS) at BNR Europe Limited, through the Teaching Company Directorate (TCD) organisation, under grant no. SAIS-003.

References

[1] T. Kohonen, "Self Organised formation of topologically correct feature maps," *Biol. Cyber.*, vol. 43, pp. 59–69, 1982.

[2] T. Kohonen, "A simple paradigm for the self-organized formation of structured feature maps," in Amari,S., and Arbib, M.[Eds], *Competition and Cooperation in Neural Nets,* Lecture Notes in Biomathemetics vol. 45, Springer Verlag, 1982.