

Source code informal information analysis using connectionist models.

Ettore Merlo, Ian McAdam and Renato De Mori

School of Computer Science, McGill University

3480 University Street

Montreal, PQ, Canada

H3A 2A7

Centre de Recherche Informatique de Montreal

3744 rue Jean-Brillant, Bureau 500

Montreal, PQ, Canada

H3T 1P1

Abstract

Design recovery, a part of the reverse engineering process of a program, must supply a programmer with all the information they need to fully understand a program or a system. In this paper, a connectionist method is proposed for analyzing the informal information (comments and mnemonics) in a program that can be used for design recovery in conjunction with more traditional approaches. A connectionist model was chosen because of its properties of being robust (capable of tolerating noisy inputs), its associative memory ability (capable of retrieving a concept given only the context of the input word that originally fired the concept), and its generalization ability (learn conceptually relevant micro-features of the domain). This approach uses a combination of top down domain analysis (the creation of a concept hierarchy by a domain expert, to be used in the formation of the training set) and a bottom up approach (the analysis of the informal information using the network).

1. Introduction

As the software industry matures, companies have built up a sizeable number of legacy systems. A legacy system is a large old system that is difficult to maintain but too valuable to abandon. Many of these programs were designed before the time of structured programming. The code of these programs has often been modified by several people over many years, and the documentation may be incomplete or may not reflect what is in the code anymore. These systems need to be modified to keep them current. These modifications may consist of fixing bugs, adding new functionality or replacing certain components.

For this purpose, tools should be developed that can reconstruct the software's design, allow one to modify the program at the design level, then generate the new source code

from this modified design. This design recovery may be rather difficult since source code does not usually contain much of the original design information, and the maintainers are often not the same people who originally developed the system. Many sources of information may need to be used during the recovery process; source code, design documentation, personal experience and general knowledge about problem and application domains [Biggerstaff, 1989].

Design recovery is not limited to maintenance [Chikofsky and Cross, 1990]. The development of new systems may require a developer to understand comparable existing systems. This understanding requires design recovery, as do many other parts of the software lifecycle. Another area where design recovery could play an important role is software reuse. Deciding how to encapsulate code fragments using data cohesion and control cohesion criteria requires a certain amount of design recovery.

Many of the processes in design recovery are a mixture of bottom up structured analysis and top down domain analysis [DeMori et al., 1992]. The bottom up approach of structural and behavioral analysis can yield rigorous, formal results, that lend themselves to formal methods. Structural and behavioral analysis work on the actual source code minus the comments and mnemonics of the identifier names.

A bottom up analysis of the comments and the mnemonics of the identifier names along with a top down domain analysis approach would be a useful complement to the structural bottom up approach. It could be used to focus attention on the relevant portion of the code for further formal analysis. It could also be used in an interactive fashion to resolve ambiguities. Another use of this informal information is to provide a method of browsing through the source code using related concepts.

The results of an analysis of the informal information in the source code will be inherently unstructured. Experience has shown that it is very difficult to achieve a high degree of consistency within documentation or naming conventions. The necessary formalism to allow such an analysis appears to be no more than a guide to other techniques, or as an automatic

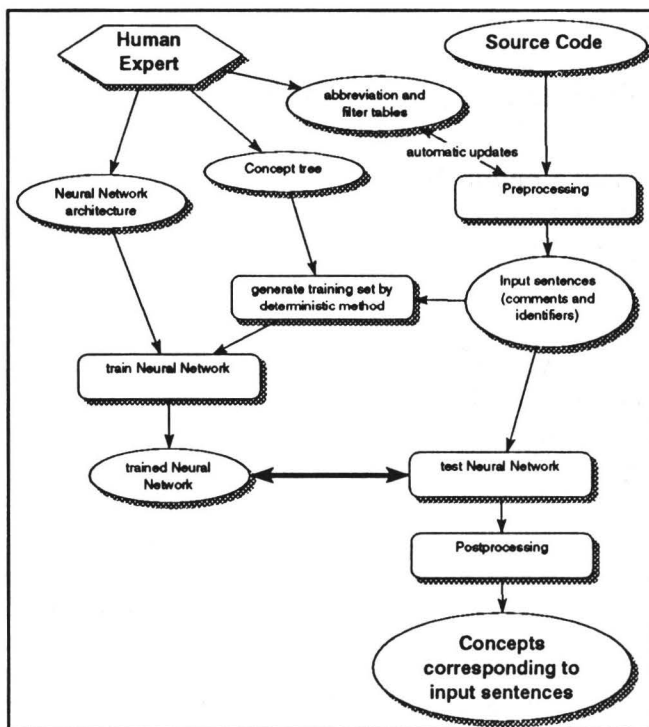


Figure 1 Overview of the process of informal information analysis

aid to manual source browsing. One possibility is to create hypertext like links to the structures created by structural and behavioral analysis. The nodes of this hypertext like system would consist of concepts extracted by the informal information analysis. For many applications of informal information analysis, one would not need to keep the results in such a structured manner (eg. source browsing).

An attractive approach for automatically learning how to obtain a concept categorization from informal information is based on Artificial Neural Networks (ANN). The advantage of using an ANN is that it can generalize from a deterministic method, used to generate a training set, by taking advantage of the textual context of the input sentence. Furthermore, a system based on ANNs is robust in extracting information from noisy data and capable of finding appropriate concepts that are only implicitly connected to the input sentence, using the associative memory capacity of the ANN [Miikkulainen, 1990].

2. System Architecture

The informal information analyzed consists of comments and identifier names. This method of extracting concepts is used to complement the traditional method of structural reverse engineering.

2.1 Overview of informal information analysis

An overview of the process of informal information analysis

is shown in Figure 1. Here rectangles represent processes while ellipses represent data structures produced by processes or activities of human experts. In particular, human experts provide the taxonomy of concepts (a concept tree) which have to be recognized in the comments of the source code. A human expert also provides abbreviation and system tables used for preprocessing the source code in order to extract the information to be sent at the ANN input. Finally, a human expert selects the ANN architecture, choosing topology (including feed-back links) and sharing of weights. The training set for the ANN is generated from the comments and identifiers filtered out by preprocessing. As training is supervised, the ANN outputs for each input pattern are concepts in the concept tree.

A neural network uses the information presented to it for either training or testing, and in the case of testing, produces an ASCII binary output. A postprocessing component takes the neural network output and transforms it into human readable form.

2.2 ANN structure

ANNs are mathematical models introduced for various applications including data compression, near-optimal solutions to combinatorial optimization problems, pattern matching, system modelling and function approximation [Simpson, 1990; Rumelhart et al., 1986].

Connectionist models typically consist of many simple, neuron-like processing elements called "units" that interact using weighted connections. Each unit has an activity level that is determined by the input received from the other units in the network. In feed-forward networks, the total input of unit j is $x_j = -q_j + \sum_i (y_i w_{ji})$, where y_i is the state of the i th unit, and q_j is the threshold of the j th unit and w_{ji} is the weight of the connection from the i th to the j th unit. The state of a unit is typically defined to be a non-linear function of its total input, such as the logistic function ($y_j = 1 / (1 + e^{-x_j})$). Learning consists of changing the weights or adding or removing connections.

Learning can be a supervised, reinforcement or unsupervised procedure. The approach described in this paper follows the supervised models of learning.

Back-propagation is a multilayer least squares learning procedure which performs gradient descent on an error surface. The error is a measure of how poorly the network is performing with its current set of weights. For each combination of weights, the network will have a certain error in performance which can be represented by the height of a point in weight hyperspace. These points form a surface called the error surface. In the case of back propagation, the error used is the least squares error - the squared difference between the computed output value and the desired output value for each unit in the output layer across all patterns in the data set.

The back-propagation technique is useful for *static* hetero-associative and auto-associative pattern mapping, in other words, for networks classifying patterns that do not vary with

time. For classifying time-varying patterns (sequences), a more powerful model is required. One possibility is to use recurrent connections within the network. This allows the network to develop a dynamic context that is updated with each new pattern presented during the sequence. This context is embodied in the set of states across the hidden units of the network and act as a short term memory. After the whole sequence has been presented, the state of the output units is interpreted as the classification of the sequence.

There are several possible ways of training recurrent ANNs [Pineda, 1987]. The most common method is to perform back-propagation through time. It consists of back-propagation of gradients to previous times by unfolding the network into an equivalent network with a different layer for each time slice.

Two types of recurrent networks have been considered, one having only local feed-back on the hidden layer. This network is trained with a method previously developed in our group [Bengio et al., 1989]. This type of network was previously used with success for Automatic Speech Recognition.

3. Informal Information Analysis

3.1 Processing methods

Feed-forward and recursive ANNs are used to extract knowledge from informal information in a program. The knowledge that is extracted consists of sets of concepts that correspond to a part of the source code.

There are different methods of processing a piece of source code: Extract identifier names, break them up into separate words and process these words as the input to the ANN. In this case, the ANN will find a set of concepts that correspond to each identifier name in the program. One can extract comments or pieces of comments and process this set of words as the input to the ANN. Extract both the identifiers and comments, and use the resulting set of words as the input to the ANN. This method is used for training in the experiments described in this paper. Extract identifiers and comments from a user selected piece of source code, break them up into separate words and process these words as the input to the ANN. This is used for an interactive source code browsing tool also developed with the research described in this paper.

3.2 Comment and identifier extraction

Comment and identifier extraction is performed by a preprocessing activity. The first step is to extract the identifiers and comments from the source code. To allow for flexibility in the types of identifiers extracted, a compiler was modified for this purpose. The preprocessor was modified to extract all the comments.

For the training set, the comments and identifiers are mixed together in their original order. For the test sets, one of

two types of files are generated, depending on the type of information one wants to analyze: a file containing identifiers and a file containing comments.

The second step of the preprocessor categorizes comments. For training, the ANN input is segmented as follows: each identifier is considered as an input sentence; comments are broken up according to the standardized fields that exist within certain comments, and in certain cases where the sentences may get very long, according to the location of newlines in the comment. It would be preferable to break up the comments by sentence boundaries, but this is unachievable since the coders left out a lot of the punctuation in the comments. Using punctuation in comments may well be a recommendation for future programmers, since it would make automatic analysis easier. When the fields are used as input sentence (segment) boundaries, an extra word containing the name of the field is concatenated to the input sentence. Free comments interspersed in the code are considered as input sentences.

For testing, the segmentation is similar, but the identifiers and comments are kept in separate files. For the identifiers, all duplicates are eliminated so as to speed up the operation of the ANN. A special table is kept of the number of instances of each identifier, so that statistics can be compiled after the ANN has run.

3.3 Sentence analysis and abbreviation expansion

The third step of the preprocessing performs Part of Speech (POS) tagging. For the comments, there are many types of words that are considered irrelevant for the purpose of the analysis. Articles, prepositions, even adverbs have very little conceptual content. Parts of speech (adjective, adverb, verb, noun etc...) are processed by a Unix utility program called 'style', that analyzes surface characteristics of a document. This program is also useful for breaking up the identifiers (most identifier names actually consist of several abbreviated words joined together by underscores), converting upper case at the beginning of a sentence to lower case and filtering out non-textual symbols.

The fourth step performs POS filtering. The selection of the filtering criteria is very important. By keeping the number of conceptually relevant words to a minimum, the ANN complexity is reduced. This also simplifies the task of the expert who has to create the list of words that will be used as the basis of the concept hierarchy.

The last preprocessing step is to convert the abbreviations into their long form. As of now, there are 334 abbreviation pairs in this table. There is an automatic method of updating the table with new abbreviations when it encounters new files.

There is also a table that filters out noisy words. This table should be used for noise words that are cluttering the input.

4. Training and test set construction

The concept hierarchy is generated with minimal effort by a

domain expert. A list of the meaningful words encountered in the comments and identifiers is automatically generated (adjectives, nouns, verbs), then submitted to an expert who is asked to eliminate the conceptually useless words, and form a hierarchy out of the remaining words.

The experiment described in this paper was performed on a system written in C for Automatic Speech Recognition (ASR) written by a team of 7 engineers and computer scientists. The system consists of 175 modules for a total of 88256 lines of code. Three modules were used for training and testing and another two modules were used for testing only. The ASR package is used for different applications ranging from connected digit recognition [Cardin et al., 1992] to understanding a database query. This package has to be maintained because various applications are built from it.

The concept tree for this system contains 106 nodes, 94 branches and has a maximum depth of 5 levels. The average fanout of the tree is 2.5. The training set consists of a file of input sentence / output sentence pairs. An example would be:

INPUT = {identifier : logarithm viterbi trellis pointer}
 OUTPUT = {logarithm, recognition-strategy, viterbi, trellis, list, pointer}

This example corresponds to an identifier indicated in the program as log_vit_trellis_ptr. The test sets is similar to the training set, though the output sentences are ignored.

The feedforward network requires a list of zeros and ones as input, one bit for each word. A file of all the possible conceptually relevant input words is used as a template. A corresponding file of all the possible concepts is used as a template to interpret the output.

For the recurrent architecture, each input sentence has to be specified by a sequence. Each input word becomes one frame of the input sequence. The output is clamped at the end of the sequence presentation to the set of concepts corresponding to the sequence meaning.

5. Experimental Results

5.1 Feed-forward architecture

The initial architecture was a simple feedforward architecture, using error back propagation for training. This consisted of 296 units in the input layer and 42 units in the output layer. The output consist of the top three layers of the concept hierarchy (the concept hierarchy was pruned so as to keep the size of the network reasonable). The number of units required in the hidden layer was determined experimentally. Architectures containing 30 hidden units, 15 hidden units and 8 hidden units were trained. Learning of these architectures converged. When an architecture of 5 hidden units was trained however, the network did not converge. There were no connections from the input directly to the output which, thus forcing all the input patterns to be mapped through the hidden layer. This suggests that between 5 and 8 micro-features are

needed to describe the 42 concepts to be identified in the hierarchy. This is quite remarkable, since it implies that it is possible to conceptually encode the 296 input units in 8 hidden units.

Each layer is fully interconnected with the succeeding layer. There is also a layer containing one bias unit, which is fully interconnected with the hidden layer and the output layer. No delays or decays are used. The hidden layer has a symmetric sigmoid transfer function and the output layer has a sigmoid transfer function. The learning rate was set to 0.1 and there was no momentum term. The stopping threshold was set to 5×10^{-3} absolute deviation. The initial weight values were set randomly. The number of weights varied between 10212 for the 30 hidden units to 2754 weights for the 8 hidden units. There were 4176 frames in the training set, guaranteeing that the network generalized using the micro-features.

5.2 Recurrent architecture

Two types of recurrent architecture were tried. The initial architecture used various feedback connections. The second architecture has only local connections, thus allowing it to use a more efficient learning algorithm. The results reported in this paper are for the recurrent ANN with local feedback only. This consisted of 296 units in the input layer and 42 units in the output layer. The output consist of the top three layers of the concept hierarchy. The number of units required in the hidden layer was determined experimentally to be 40. This suggests that 40 micro-features are needed to describe the 42 concepts as well as the considerable amount of sequential information. There were no connections from the input directly to the output thus forcing all the input patterns to be squeezed through the hidden layer.

Other network details are similar to the ones for the feed-forward architecture and are omitted for the sake of brevity. The number of weights was 15202 for the 40 hidden units. There were 25431 frames in the training set, guaranteeing that the network generalized using the micro-features.

5.3 Performances

Given a comment segment CS_i , the ANN associates a set CNi of concepts to it. A set of comment segments has been hand labeled to evaluate the classification performance of the ANN. The set contains correct concepts and concepts that the network has missed originally (deletion errors). Furthermore some concept were wrongly inserted by the ANN (insertion errors). The intersection between the concepts hypothesized by the ANN and the correct concepts CC_i gives the set NNC_i which represents the correct concepts identified by the ANN:

$$NNC_i = CN_i \cap CC_i \quad (1)$$

where $\cap$ is the set theoretic intersection.

Words representing concepts can be extracted from the text by a lexical matcher. A lexical match consists of the set LL_i of words that are in the input sentence after pre-processing. We assume that if a word in a comment is used in the taxonomy of

concepts, it has to be considered a correct representation of that concept. That is, we assume that the programmer chose lexically relevant words. The set of concepts represented by words in a comment segment is indicated as LLC_i and is produced by the intersection of words used to describe the items in the taxonomy with lexical matches for a given comment segment. Figure 2 shows a diagram representing all the sets involved in the concept recognition in comment segments. The Taxonomy set represents the concept hierarchy. Since the output units of the ANN correspond to the elements of the Taxonomy set, CC_i and CN_i are subsets of it. LL_i , on the other hand, represents a subset of the concepts in the English language and is not in general a subset of Taxonomy.

The performance of the ANN classifier can be evaluated by comparison with the results of a lexical matcher. For this purpose, the following measures are defined and computed for each comment segment CS_i :

card (CC_i) - size of correct conceptual matches,
card (NNC_i) - size of correct neural matches,
card (LLC_i) - size of correct lexical matches,
card ($CC_i - CN_i$) - size of ANN deletion error set,
card ($CN_i - CC_i$) - size of ANN insertion error set,
card ($CC_i - LL_i$) - size of lexical deletion error set,
card ($LL_i - CC_i$) - size of lexical insertion error set.

Total counts are computed by summing up over the total number of comments and identifiers. The **percent correct** and **accuracy** are computed as:

$$PCN = NNC / (NNC + DELN) \quad (2)$$

$$ACN = NNC / (NNC + DELN + INSN) \quad (3)$$

$$PCL = LLC / (LLC + DELL) \quad (4)$$

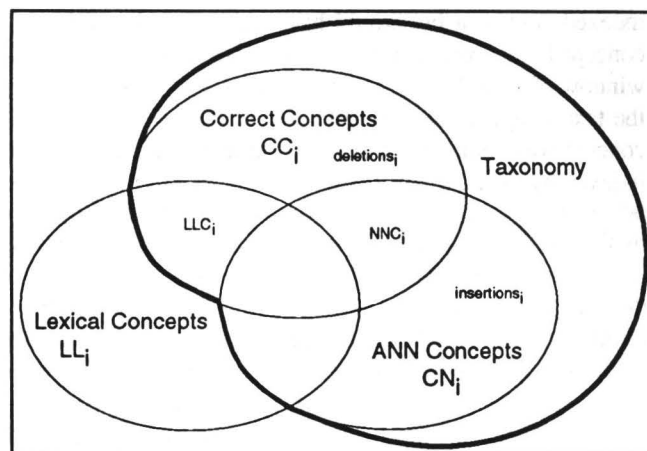


Figure 2 Diagram showing the sets of concepts that exist during classification

$$ACL = LLC / (LLC + DELL + INSL) \quad (5)$$

where

PCN = percent correct for the ANN classifier

ACN = accuracy for the ANN classifier

PCL = percent correct for the lexical matcher

ACL = accuracy for the lexical matcher

CC = Total number of correct conceptual matches,

NNC = Total number of correct neural matches,

LLC = Total number of correct lexical matches,

INSN = Total number of ANN insertion error set,

INSL = Total number of lexical insertion error set.

DELN = Total number of ANN deletion error set,

DELL = Total number of lexical deletion error set,

Table 1 shows the experimental results for comments and identifiers for both the training and completely disjoint test sets. The average accuracy and percent correct for the recursive ANN classifier and the lexical matcher is shown. Table 1 also shows the average sizes of CC_i , NNC_i , LLC_i , CN_i , and LL_i . Totals are also reported for insertion and deletions for the recursive ANN classifier and the lexical matcher. The total number of comments is also shown.

5.4 Analysis of results

An analysis of the results shows that ANN classifier was able to retrieve more code fragments than a lexical matcher based on concepts. The errors are mostly insertion errors, causing false alarms if the ANN is used to provide guidance to code analysis techniques which are expensive in terms of computation.

For the application of a concept acting as an index for retrieving comments, the minimum distance in lines between a comment indexed by a particular concept using the ANN, which is correct most of the time, and the closest comment indexed by the same concept using a lexical matcher was introduced. This metric could be used to measure the usefulness of the indexing, since a programmer can easily identify a concept which appears a few lines away from the

	comments		identifiers	
	Training	Testing	Training	Testing
average CC_i	3.56	3.79	2.97	2.41
average CN_i	3.77	3.82	4.02	3.86
average NNC_i	3.46	3.47	2.77	2.2
average LL_i	8.80	9.13	2.33	2.48
average LLC_i	0.92	1.10	0.73	.77
CC	733	785	859	829
NNC	712	719	800	759
LLC	189	227	211	266
INSN	64	71	361	570
DELN	21	66	59	70
INSL	1624	662	462	588
DELL	544	558	648	543
PCN	97.14%	91.59%	93.13%	91.56%
PCL	25.78%	28.92%	24.56%	32.09%
ACN	89.34%	84%	65.57%	54.25%
ACL	8.02%	9.28%	15.97%	19.04%
total number	206	207	289	344

Table1 Results for ANN and lexical classification

indexed comment but would have difficulty identifying the concept if the comment that was indexed was outside the window of visibility of the programmer. For each concept in the taxonomy, the set of comments that are indexed by this concept was computed. For the comments that are correctly indexed by the ANN the distance, measured in lines of code, between such a comment and the closest comment indexed by the same concept using a lexical matcher, was computed.

Figure 3 shows the histogram of the distribution of distance values computed as described above. A null distance means a colliding match. Distances greater than 144 lines (the window of visibility, roughly corresponding to two pages of code listing) were cumulated together. Figure 3 shows that the majority of lexical matches are outside the window of visibility, although there is a smaller number of lexical matches which are less than a few lines from the ANN match. For the test set, it was calculated that 8.02% of the matches were colliding matches, 14.3% were less than 6 lines away, 12.92 were between 7 and 143 lines away and 72.93% were 144 or more lines away.

Table 1 shows that the average percent correct and the average accuracy of the ANN classifier are high, and significantly higher than those of the lexical matcher. The difference is in general more than 60% bigger for percent correct. The ANN classifier performance is closer to the correct classification than the lexical matcher, as can be seen by comparing the values of average NNCi and average LLCi with the average CCI. As one can see, the lexical matcher finds an average of 1 or less concepts per comment or identifier.

A detailed error analysis reveals that certain comment segments and identifiers were not significant, some errors were due to an incomplete abbreviation table or incomplete taxonomy, and some classification errors were due to the choice of a concept name which can carry different meanings

in the taxonomy or in some comments.

Acknowledgments

The authors wish to thank Yoshua and Sami Bengio for providing support in the use of the McGill ANN simulator and Yves Normandin for providing the source code to be analyzed and for helping in creating the taxonomy of concepts. This research was partially supported by the Natural Science and Engineering Research Council of Canada.

Bibliography

- [Bengio et al., 1989] Bengio Y., Gori M., and DeMori R., "BPS: a learning algorithm for capturing the dynamic nature of speech", in the Proceedings of the international Joint Conference on Neural Networks, pp II-417 - II-424, 1989.
- [Biggerstaff, 1989] Biggerstaff, T. J. "Design recovery for maintenance and reuse." IEEE Computer 22 pp 36-49, 1989.
- [Cardin et al., 1992] Cardin R., Normandin Y. and DeMori R. "High Performance Connected Digit Recognition using Codebook Exponents", Proc 1992 Conference on Acoustic, Speech and Signal Processing. San Francisco, CA, pp I-505, I-509, March 1992
- [Chikofsky and Cross, 1990] Chikofsky, Elliot and Cross, James. "Reverse engineering and design recovery: a taxonomy", IEEE Software, Vol 7 No 1, pp 13-17, 1990.
- [DeMori et al., 1992] De Mori R., Merlo E., Kontogiannis K., Girard J. F., Levine J., "Program Understanding Through Structural and Behavioral Recognition, and User Interaction", AI and Automated Program Understanding Workshop, AAI-92, San Jose, California, July 1992
- [Miikkulainen, 1990] Miikkulainen Risto. "A neural network model of script processing and memory." Technical report UCLA-AI-90-03, Computer science department, UCLA. 1990.
- [Pineda, 1987] Pineda F. Generalization of back-propagation to recurrent neural networks. Physical Review Letters, 19, pp 2229 - 2232, 1987.
- [Rumelhart et al., 1986] Rumelhart D.E., Hinton G., McClelland. A general framework for parallel distributed processing, in Parallel Distributed Processing, Bradford books / MIT press, 1986.
- [Simpson, 1990] Simpson P., Artificial Neural Systems, Pergamon Press, 1990

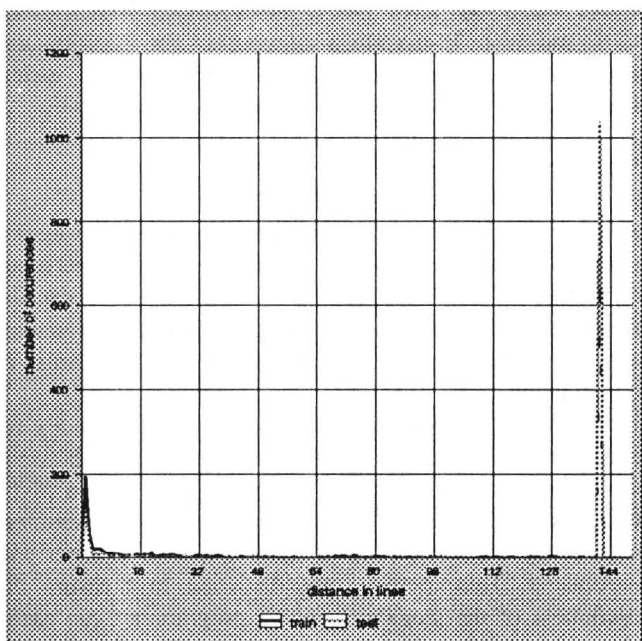


Figure 3 Distance in lines between lexical and ANN matches