Identifier Splitting: A Study of Two Techniques *

Henry Feild
Loyola College
Baltimore MD
21210-2669, USA
hfeild@cs.loyola.edu

David Binkley
Loyola College
Baltimore MD
21210-2669, USA
binkley@cs.loyola.edu

Dawn Lawrie
Loyola College
Baltimore MD
21210-2669, USA
lawrie@cs.loyola.edu

April 15, 2006

Abstract

Any time source code is analyzed, whether for maintenance or general code comprehension, identifiers play a key role. Their makeup, therefore, is very important. Each identifier can be thought of as constructed from individual words, where each word may be found in a natural language dictionary or might be an abbreviation, a programming language key word, a program-specific word, or just plain gibberish. A summary of the types of words found in the code can indicate, at a glance, how comprehensible the code will be.

The problem of determining the words that make up an identifier in the absence of explicit word divisions such as underscores and camel-casing is addressed. One approach considered is a greedy algorithm. However, greedy algorithms can be time and space inefficient, making them less desirable. One interesting alternative is an artificial neural network. Neural networks are fast and have been used for speech and vision recognition, as well as a host of other patter-recognition tasks. This paper describes both a greedy algorithm and an a neural network (using the C-implementation of the Fast Artificial Neural Network) that are used to split non-well separated identifiers.

*Proceedings of MASPLAS'06 Mid-Atlantic Student Workshop on Programming Languages and Systems Rutgers University, April 29, 2006

1 Introduction

Identifiers, which represent the defined concepts in a program, account for, by some measures, almost three quarters of source code [5]. The makeup of identifiers plays a key role in how well they communicate these defined concepts.

Motivation for the analysis of identifiers comes from several previous studies. For example, Deißenböck and Pizka observe that "Research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process" [5]. Thus, they conclude that the importance of identifier names is crucial to program comprehension [5]. A second motivation comes from the work of Caprile and Tonella, who observe that "Identifier names are one of the most important sources of information about program entities" [4].

Furthermore, Rilling and Klemola observe that "In computer programs, identifiers represent defined concepts [where] identifier density corresponds to comprehension cost" [12]. Knuth noted that descriptive identifiers are a clear indicator of code quality and comprehensibility [8]. As a measure of how much of a program is devoted to identifiers, Deißenböck and Pizka report that in the source for eclipse (about 2 MLoC) 33% of the tokens and 72% of characters are devoted to identifiers [5].

Finally, Antoniol et al., observe that most of the application-domain knowledge that programmers possess when writing code is captured by identifier mnemonics [2]. Thus, how readily the semantics inherent to identifiers can be extracted is of key importance. They write, "Programmers tend to process application-domain knowledge in a consistent way when writing code: program item names of different code regions related to a given text document are likely to be, if not the same, at least very similar." Thus, an underlying premise of their work is that programmers use organized methodical meaningful (high quality) identifiers for code items.

Following Takang et al., Brook's theory of program comprehension underpins the theoretical framework of this analysis [15]. Brooks argues that programming involves the construction of mappings from a problem domain via intermediate domains into a programming domain – represented by program text. He contends further that the process of program comprehension is one of reconstructing knowledge about these domains and the relationships between them.

Thus, to comprehend a program, an engineer must map the identifiers to the concepts they represent. Jones notes that a variety of different kinds of character sequences are used in source code identifiers [6]. Some are complete words or phrases, some abbreviated forms of words or phrases, while others have no obvious association with any known language. Studies have found that people's performance in processing character sequences can vary between different kinds of sequences. For instance, frequently used character sequences (*i.e.*, dictionary words) are recognized faster and are more readily recalled than rare ones [6]. Thus, the more meaningful the identifiers of a program, the easier it is to map them to appropriate concepts. As an example, compare, pqins() with priority_queue_insert().

Anquetil and Lethbridge (among others) have observed that there is some controversy over the value of general identifier names [1]. For example, Sneed finds that "in many legacy systems, procedures and data are named arbitrarily $\cdots$ programmers often choose to name procedures after their girlfriends or favorite sportsmen" [14]. A similar pattern was observed by one of the authors at a previous industrial position in the code of a colleague who was fond of Star Wars.

This paper describes two methods of breaking an identifier down into its most basic parts: a greedy algorithm and a neural network. This paper considers the source code used in the study in Section 2. It then presents the two techniques and empirical studie them in Sections 3 and 4. The final three sections consider related work, future work and conclusions.

2 Source Code

This section describes the source code from which the test set of identifiers were chosen and the extraction process. The source code used in the experiments comes from 186 programs, and includes 26MLoC of C, 15MLoC of C++, 7MLoC of Java, and 21KLoC of Fortran. The total of almost 50MLoC includes almost 3 million unique identifiers and over 55 million identifier instances; thus, the typical identifier appears about 19 times. Not all of this was used, however. To make the training and testing more manageable, a sample of 4,000 randomly selected identifiers were selected. The large size of available source code spreading multiple languages will hopefully help the application of the results to programs written in these languages outside of our available source.

3 Techniques

Past studies of identifier quality typically begin by dividing identifiers into their constituent parts [3, 5, 4, 12, 2, 6]. Herein, these parts are called "*words*" – sequences of characters with which some meaning may be associated. Words are atomic; thus, a word is never subdivided. Two kinds of words are considered: *hard words* and *soft words*. Hard words are denoted by the use of word markers (*e.g.*, the use of CamelCase or underscores). Following Caprile and Tonella [3], an identifier, I, is *well-separated* if, for any (ordered) pair of adjacent words in I, one of the following conditions holds: (1) one of the words in the pair has lexical type equal to special-string (*e.g.*, '_'); (2) the word that comes second in the pair is capitalized. For example, the identifiers sponge_bob and spongeBob both contains the well separated hard words sponge and bob.

For many identifiers, the division into hard words is sufficient. This occurs when all the hard words are mem-

bers of one of three lists: the list of *dictionary* words, the list of known *abbreviations*, or finally, borrowing an idea from Information Retrieval, the *stop list*. When a hard (or soft) word is on one of these three lists, then it is referred to as "on a list".

The study uses the publicly available dictionaries that accompany the Linux spell checker ispell (Version 3.1.20). The list of abbreviations includes domain abbreviation (*e.g.*, alt for altitude) and programming abbreviation (*e.g.*, txt for text and msg for message). Finally, a stop-list is used to omit words that are not thought to bring useful information. In English, words such as *"the"* are typically eliminated. For identifiers, the stop list contains three type of words: predefined type names (*e.g.*, int), language or environment global variables (*e.g.*, errno), and standard library names (*e.g.*, printf).

3.1 *Greedy Algorithm*

The greedy algorithm looks for the longest prefix and the longest suffix that are "on a list" (*i.e.*, in the dictionary, on the list of abbreviations, or on the stop list). Starting with each hard word that is not on a list, the algorithm conducts two searches and keeps the better result. The first search finds the longest prefix of the current word that is on a list. It then recursively calls itself on the remaining portion of the original hard word (this recursive call considers both prefixes and suffixes). The second search is the same as the first except it searches for the longest suffix. The results of the two searches are compared and the one producing the higher ratio of "*on list*" soft words to *total* soft words is chosen. If neither the prefix nor the suffix search generates any "on list" words, the process is repeated with the first character removed from the identifier. Removed characters are gathered together and prepended to the result. If the first soft word in the split identifier is a not on a list, then these characters are prepended to this first soft word. Otherwise, they are added as a new first soft word in the split identifier. The final output is the split identifier.

3.2 *Artificial Neural Network*

The second technique uses an artificial neural network, which mimics the parallel structure of neurons in the human brain. Traditionally, there are three layers in a neural network – an input, hidden, and output layer – each consisting of numerous "neurons" or "nodes". While the hidden layer can actually contain several sub-layers, it is common that only one is used. Every node in the input layer is connected to every node in the hidden layer, each of which is connected to every node in the output layer. Every connection consists of a weight. This weight is multiplied by the value of the sending node and then summed with every other node connecting to the receiver node. This sum is then passed into an activation function which will cause the receiving node to "fire" or "not fire", just like a human neuron.

In order to make a neural network usable, it must be trained. To train it, a set of data containing inputs and expected outputs is compiled. This must then be translated into a format that the neural network can understand. The program that performs the training will then upload the data set and start to run the data through the network, checking if the actual output of the network matches the expected output. This generates an error rate which, depending on the type of network, will update weights on the connections. When and how the weights are changed depends on the *training algorithm*. Some training algorithms update after every individual input, others after every input from the data set have passed through (called an *epoch*). The more popular training algorithms are updated by means of back-propagation (the mean-squared error is fed backwards through the network). The network is trained until it either achieves a state that produces a mean-squared-error that is less than or equal to what the trainer desires or it reaches its maximum number of epochs. When finished, the trained network is output to a file where it can be loaded by a network run program. This program acts like a black box – a user can now pass input through it and attain output, but the contents of the network is hidden.

For the neural network used in this experiment, the Fast Artificial Neural Network (FANN) library was used [11]. The implementation is in C, which helps minimize the training times. The network uses a symmetric sigmoid activation function, meaning the activations are between -1 and 1, as opposed to the more typical 0 to 1. The steepness of both the hidden and output layers was set to 0.3, which preserves fractional output. Finally, resilient back-propagation (*rProp*) is used for the training algorithm. This approach is one of the best training algorithms currently known. Other algorithms were experi-

mented with, however resilient back-propagation yielded the best results.

The input and output layers both contain 25 nodes, which allows for an identifier of up to length 25 to be processed. Each of the input nodes is set to a real value between -1.0 and 1.0 depending on the ASCII value of the corresponding character. All characters are converted to lowercase and non-alphabetic characters are coded as 0, as are the remaining positions between the end of the identifier and the 25^{th} node.

In the network a single hidden layer is used (multiple layers were experimented with). The rule of thumb for the size of the hidden layer, obtained from a neural network newsgroup, is two-thirds the size of the input and output layers summed. For the network used, this meant a single layer of 33 nodes. Other sizes were tried, but 33 produced the best results. Finally, if the value of an output node is between -0.9 and 1.0, then the neural network is indicating that there should be a split between the corresponding character and the following character from the input layer.

The network was configured to test itself and report a summary every 25 epochs. This data, shown in Figure 2, showed that the network performed best around 1100 epochs. Thus, in the experiment 1100 epochs were used to train each network.

3.3 *Training the Network*

To train the network, a list of unique hard words needs to be obtained from a collection identifiers. The hard words are then manually split, with expected splits represented with underscores. This list is then divided into two files: one for training, the other for *cross-validation*: cross validation consists of taking a data set that is not used in the actual training of the network and then periodically running it through the network during training. The mean-squared-error returned by this process is then combined with the mean-squared-error of the network and incorporated in the back-propagation. By convention, 90% of the original list is used for training and the remaining 10% for cross validation. Both of these lists are then translated into input for the network.

When the network-training program runs, it loads both the training and cross-validation lists. It begins training and every 25 epochs, a call back function is used to perform the cross validation. By default, the FANN library will only return a mean-squared error when testing a network. To attain raw success and error rates, the output from the network in its current state in training was directly compared to to the respective expected output as given by the training data. This raw comparison yielded interesting information, such as the number of splits correctly identified by the network, the number of extra splits added by the network, and the number of identical matches between the expected and actual output. This data was output during training in a format that could be read by gnuplot and was used as a visual aid in determining which variables could be changed to improve training.

	Average Run Time	
Programmer	**own set**	**All 4**
programmer 1	0.175	0.521
programmer 2	0.177	0.453
programmer 3	0.171	0.448
programmer 4	0.171	0.453
all 4	0.171	0.453
greedy algorithm	1.211	2.516

Figure 1: Running times in seconds for five trial networks and the greedy algorithm. (The columns for "own set" runs each programmer on only their training set of 1000 words.)

4 Results

A sample of 4,000 identifiers were randomly selected from the set of program described in Section 2. To be used for training, these were manually split by inserting a dash ('-') a word division should appear. Four programmers were each asked to split one quarter of the training identifiers.

For testing, five neural networks were trained using five different data-sets (the same options were used when training each network). Four were trained on each of the data sets generated by the four programmers. The fifth network was trained on the combination of these four data sets. These tests should show how well a network does on a general program level as well as differences between the individuals who perform the manual splitting for the train-

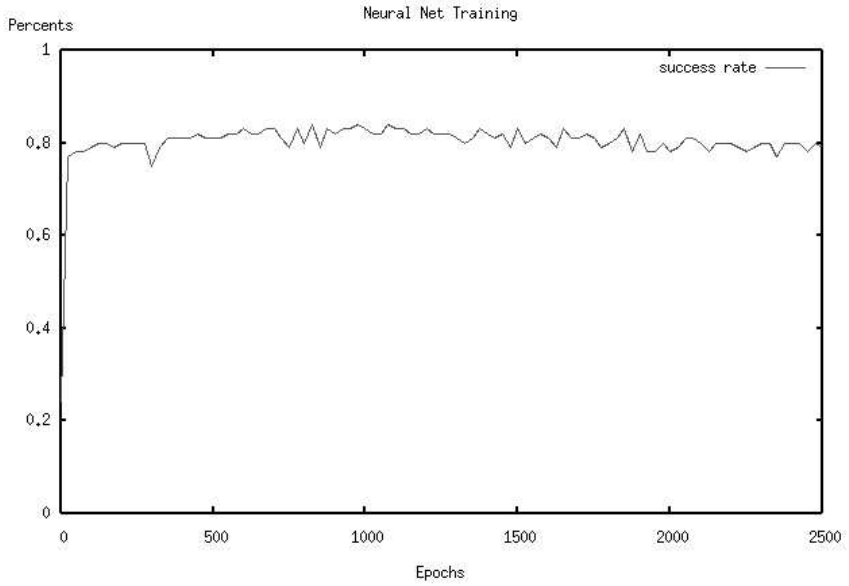

Figure 2: The success rate as reported every 25^{th} epoch by a network in training

ing data. Each data set was run through each network in addition to the greedy algorithm, yielding a total of thirty runs.

4.1 *Greedy Algorithm Results*

Because the greedy algorithm is independent of the data it is run on, it has the benefit of being consistent across all data sets. However, the simple rules it follows can differ from those intuitively applied by a programmer. This is shown in the success rates of the greedy algorithm, which range from 74% to 80%.

Figure 1 shows the run times for the network and the greedy algorithm when run on several inputs. The execution time of the greedy algorithm includes an initialization step that is dominated by the loading of several dictionaries into hash tables. The cost of this step is constant regardless of the number of identifiers analyzed by the program. The data indicates that this initialization time is not the only part weighing down the time. If it was, there would be a smaller increase in time between the running times for the four smaller data sets and the one large set. The greater increase in time, however, shows that the algorithm itself is relatively time-consuming.

4.2 *Network Results*

The neural network results vary from data set to data set. This is because the manually splitting of the training data is a rather subjective process. However, the results are still interesting. For example they show how different splitting methods effect training and success rates on unseen data (*e.g.*, the other data sets). One interesting observation is that the 'all' network performed worse than the others. This is because it must attempt to reconcile the many often conflicting ways of splitting used in the four data sets. The results is that the network does not learn how to correctly split as many identifiers.

4.3 *Conclusion*

The greedy algorithm's consistency is its best quality, however it requires an up-to-date dictionary and abbreviation list. The algorithm fails to meet the varying ex-

pectation of the authors. Finally, not once did the greedy algorithm climb above 80% accuracy on one of the four data sets. The running times were also notably longer.

The neural network demonstrated that it can learn to split identifiers. Here, consistent and accurate training data plays a key role. For example, consider the low success rate of the 'all' data set, which achieved an average rate of 66% on the various networks. These tests have also shown how fast a network really is compared to the greedy algorithm.

As seen in Figure 3, the data set 'all', which is a combination of the four other sets, performs badly compared to the others in all cases. The 'p1' data set was the most consistent in splitting, as can be seen by the 88% success rate when run on the network which was trained using it. As expected, all the data sets did second best on the 'all' network, since each set was a subset for the training of the 'all' network.

The greedy algorithm did better generally, however it failed to achieve a success rate for the 'p1' and 'p4' data sets that that competes with the 'p1' and 'p4' nets success rates for their respective data sets.

5 Related Work

This is the first use of neural networks for identifier splitting. However, much has been done in the similar field of pattern matching text, such as word meaning [13] and text-to-speech translation [7]. While these related works use different input and output translation methods, they are rich in information regarding the general structure of a network and its settings for text processing. Their results also give us an idea of how well the data presented herein stands on its own. For instance, Nakamura et alindicate that success rates for pattern recognition can be in the 80%-90% range. Their word category prediction network increased the success rate for prediction methods from 81% (set by a previous study) to 86.9% [10].

In an article on text classification, Dieter Merkl suggests that "There is general agreement that the application of artificial neural networks may be recommended in areas that are characterized by (i) noise, (ii) poorly understood intrinsic structure, and (iii) changing characteristics" [9]. These three characteristics are present in identifier splitting as there noise, such as embedded keywords (*e.g.*, "printf"), poorly understood structure, such as the disagreement witnessed first hand by the authors in the manual splitting, and changing characteristics, such as the use of new abbreviations.

6 Future Work

Future work revolves around establishing a network that can be used on arbitrary programs.

To achieve this, there are several steps to be considered. One idea involves the identifier-to-network-input translation. There appears to be no relevant work that tries to translate input character-by-character as described in this paper. Therefore there is no model to compare to concerning the input translation.

Another approach involves having the network automatically pick the best network state during training. Currently, the network saves itself after the final epoch. However, the network tests itself every 25^{th} epoch, and therefore can check its state at that point, compare it to the previous "best-network", and then save over that if it is currently better. The challenge here is determining what a "better" network is. Both the success of the training set and the cross-validation set must be considered.

The most pivotal improvement will be to gather a large data set that contains consistent data. This would require a standard, which would have to be agreed on by the authors.

7 Summary

This paper described two methods which automate the process of identifier splitting. This is a useful task because an identifier with distinct work breaks is easier to comprehend. Of the two methods, the neural network has proved to be faster and more adaptable to the intuitive splitting methods of the authors. Future work will hopefully assist the network to attain better results and become a stable method of identifier splitting.

References

[1] N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software sys-

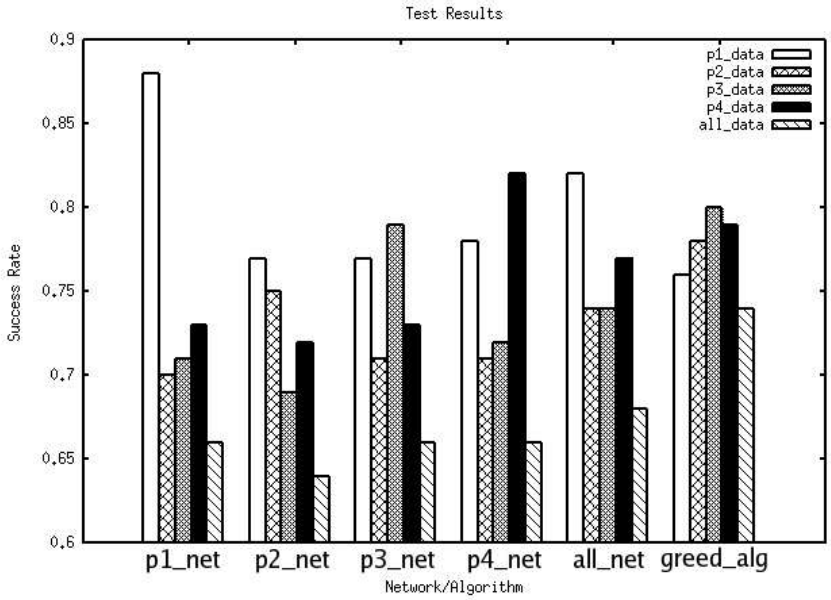

Figure 3: Comparison of the networks and greedy algorithm

tem. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, November 1998.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10), October 2002.

[3] B. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, pages 112–122, Altanta, Georgia, USA, October 1999.

[4] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, pages 97–107, 2000.

[5] F. Deißenböck and M. Pizka. Concise and consistent naming. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, MO, USA, May 2005. IEEE Computer Society.

[6] D. Jones. Memory for a short sequence of assignment statements. *C Vu*, 16(6):15–19, December 2004.

[7] Orhan Karaali, Gerald Corrigan, Ira Gerson, and Noel Massey. Text-to-speech conversion with neural networks: A recurrent tdnn approach, 1997.

[8] D. Knuth. *Selected papers on computer languages*. Stanford, California: Center for the Study of Language and Information (CSLI Lecture Notes, no. 139), 2003.

[9] Dieter Merkl. Text classification with self-organizing maps: SOM lessons learned. *Neurocomputing*, 21(1):61–77, 1998.

[10] M. Nakamura, K. Maruyama, T. Kawabata, and K. Shikano. Neural network approach to word category prediction for english texts, 1990.

[11] Steffen Nissen. Neural networks made simple, 2005.

[12] J. Rilling and T. Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the* 11^{th} *IEEE International Workshop on Program Comprehension*, Portland, Oregon, USA, May 2003.

[13] Miguel E. Ruiz and Padmini Srinivasan. Hierarchical text categorization using neural networks. *Information Retrieval*, 5(1):87–118, 2002.

[14] H. Sneed. Object-oriented cobol recycling. In *3rd Working Conference on Reverse Engineering*, pages 169–178. IEEE Computer Society., 1996.

[15] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Program Languages*, 4(3):143–167, 1996.