AN EMPIRICAL COMPARISON OF TECHNIQUES FOR EXTRACTING CONCEPT ABBREVIATIONS FROM IDENTIFIERS

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

Abstract

When a programmer is faced with the task of modifying code written by others, he or she must first gain an understanding of the concepts and entities used by the program. Comments and identifiers are the two main sources of such knowledge. In the case of identifiers, the meaning can be hidden in abbreviations that make comprehension more difficult. A tool that can automatically replace abbreviations with their full word meanings would improve the comprehension ability (especially of less experienced programmers) to understand and work with the code. Such a tool first needs to isolate abbreviations within the identifiers. When identifiers are separated by division markers such as underscores or camel-casing, this isolation task is trivial. However, many identifiers lack these division markers. Therefore, the first task of automatic expansion is separation of identifiers into their constituent parts. Presented here is a comparison of three techniques that accomplish this task: a random algorithm (used as a straw man), a greedy algorithm, and a neural network based algorithm. The greedy algorithm's performance ranges from 75 to 81 percent correct, while the neural network's performance ranges from 71 to 95 percent correct.

Keywords: neural network, software maintenance, code comprehension, software engineering

1 Introduction

Identifiers, which represent the defined concepts in a program, account for, by some measures, almost three quarters of source code [4]. The makeup of identifiers plays a key role in how well they communicate these defined concepts. Thus, they aid in the transfer of understanding from a current programmer to subsequent programmers. In addition, tools that attempt to assist engineers when performing source code analysis (*e.g.*, for program comprehension, maintenance, or evolution) will be more successful if the tool can map program identifiers to domain level concepts (for example those appearing in the documentation).

Motivation for the importance of identifiers in transferring knowledge 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" [4]. Thus, they conclude that the importance of identifier names is crucial to program comprehension [4]. 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" [3].

Furthermore, Rilling and Klemola observe that "In computer programs, identifiers represent defined concepts [where] identifier density corresponds to comprehension cost" [14]. 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 [4].

To understand a program, an engineer must map the program's 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 [10]. 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()`.

When performing maintenance on code written by others, transfer of the concepts referred to by the identifiers is of great importance. Those identifiers comprised completely of natural language words and well known abbreviations lead to faster comprehension than those using (unknown) abbreviations or those that are completely nonsensical [6, 10]. The latter increase comprehension cost.

Given the presence of abbreviations (in particular those whose concepts are unknown) the long term goal of this work is to use machine translation techniques [11] to map these abbreviations to domain level concepts. This should enable all readers of the code to understand the defined concepts in the program. Several steps are required to attain this goal: first, the identifier must be divided into its constituent "words." Then any abbreviations must be identified, and finally the concepts must be extracted (typically from the internal and external documentation).

The work presented in this paper focuses on the first step. Although many identifiers include easily identifiable clues in the form of camel-casing or underscores, many do not. This work discusses two different techniques for extracting the constituent words of the later class. The first is a greedy algorithm that searches for longest substrings. The second implements a neural network to predict splitting points. A random algorithm is also presented and used to generate base line output. An empirical comparison of the three techniques is reported.

The remainder of this paper consists of a description of the three techniques in Section 2. An empirical study is then presented in Sections 3 and 4. The final two sections discuss related work and a conclusion.

2 The Techniques

This section first introduces some terminology regarding identifiers before describing the three splitting techniques and discussing some example output. Identifiers come in a variety of forms. Some consist of a single concept (*e.g.,* hash); others may contain several concepts separated by some *division marker* (*e.g.,* hash_table or hashTable); many, however, consist of multiple concepts with no division markers (*e.g.,* hashtable) or some mixture thereof (*e.g.,* hashtable_entry).

The following terminology is used in this paper: word breaks (*e.g.,* underscores and camel-casing) are referred to as *division markers*, while the string of characters between division markers and the endpoints of an identifier are referred to as *hard words*. Some hard words directly identify a concept (*e.g.,* those that exist in the dictionary). Others include multiple parts (and sometimes multiple concepts). These parts are referred to as *soft words*. Thus, a hard word consists of one or more soft words. In the later case, the absence of division markers makes the individual soft words (concepts) difficult to extract.

Take, for example, the identifier hashtable_entry. This identifier consists of one division marker (an underscore) and, thus, two hard words, hashtable and entry. The hard word hashtable is composed of two soft words – hash and table, while the hard word entry is composed of a single soft word.

In order to test the three algorithms, it is necessary to have a test oracle. In essence, this data is a list of pairs that each include the original input hard word and the properly divided output. For example, the test data includes the pair

hashtable hash-table

where a hyphen is used to indicate where division markers should occur. Construction of the oracle data used in the experiments is described in Section 3.

The remainder of this section describes the three techniques:a *random algorithm*, a *greedy algorithm*, and an algorithm based on an *artificial neural network*. The goal of each is to identify all of the soft words within an identifier.

2.1 The Random Algorithm

The first technique is the simplest of the three. It randomly divides a hard word into soft words with the same frequency as in the oracle data. For example, consider a data set consisting of a single ten character hard word identifier where the oracle inserted a single division marker. This identifier has nine inter-character locations into which a division marker can be inserted. Given a single division marking in the oracle data, the random algorithm would have a one in nine chance of inserting a division marker between each pair of characters.

The random algorithm is used as a kind of straw man and allows for the comparison of relative performance of the other algorithms. To understand this use of the random algorithm, consider the performance of algorithm *A* on two data sets *D1* and *D2*. If *A* correctly divides 76% of *D1* and 80% of *D2*, then using an absolute scale *A* does better on *D2*. Now consider the situation assuming the random algorithm correctly divides 50% of *D1* and 70% of *D2*. One conclusion form this is that *D2* is easier. Using a scale relative to the random algorithm's performance, *A* does comparatively better on *D1*. Thus, the random algorithm forms an alternative base-line useful for measuring algorithm performance when comparing results on different data sets.

2.2 The Greedy Algorithm

The second technique implements a greedy algorithm that employs three lists: *dictionary* words, *known abbreviations*, and finally, borrowing an idea from Information Retrieval (IR), a *stop list*. The implementation uses the publicly available dictionaries that accompany the Linux spell checker ispell (Version 3.1.20) (45,292 entries). It also uses a list of common abbreviations extracted from several programs and the authors experience.

Abbreviations include domain abbreviations (*e.g.,* alt for altitude) and programming abbreviations (*e.g.,* txt for text) (261 entries). At present this list is small as it was conservatively constructed to include only truly common abbreviations. One area of future work is to look at the ability of IR classification techniques to automatically extract program specific abbreviations (and tie them to concepts in the documentation) [9].

Finally, a stop-list is used to omit words not thought to bring useful information. These include keywords (*e.g.*, `while`), predefined identifiers (*e.g.*, `NULL`), library function and variable names (*e.g.*, `strcpy` and `errno`), and all identifiers that consist of a single character (4,573 entries). This list is large because it includes all function names from several common libraries (*e.g.*, `libc`).

The greedy algorithm begins by performing a lookup on each hard word. Those on one of the lists are returned as a single soft word. The remaining hard words are assumed to be composed of multiple soft words. The search for these soft words is a recursive procedure that begins by looking for the longest prefix and the longest suffix that is on a list. The two are symmetric. In the case of the longest suffix, successive characters are removed from the beginning of the hard word until a string on one of the lists is encountered (or the string becomes empty).

If the string does not become empty, both searches are then recursively called on the remaining characters. Upon return the result from the call (prefix or suffix) returning the higher ratio of soft words found on one of the lists to total soft words is returned. At each return, a division marker is inserted.

2.3 The Neural Network

The third approach uses an artificial neural network to split hard words into soft words. Traditionally, there are three layers of "neurons" or *nodes* – the input, hidden, and output layer. The input and output of each node (and thus the network) is numeric. 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 is weighted. This weight is multiplied by the value of the sending node and then summed with every other node connected to the receiver node. This sum is then passed into an activation function which will cause the receiving node to "fire" or "not fire", similar to a human neuron. The number of input and output nodes is determined by the problem being solved. Experimentation found that the optimal number of nodes in the hidden layer was about two-thirds the number of nodes in the input and output layers combined.

The experiments made use of the Fast Artificial Neural Network (FANN) library [12]. The resulting networks are *symmetric*, meaning the value of each node in the network falls between -1.0 and 1.0. To "learn" weight values, a FANN network is "trained" using one of several available feedback algorithms from within the FANN library. The networks used in this study implemented the *improved resilient back propagation algorithm*, which performs best for many problems [12]. The training process makes multiple passes over the training data to tune the weights. Each pass is referred to as an *epoch*. The networks presented here are trained over the course of 1100 such epochs.

To pass hard words into the network, they must first be encoded as numeric values. Similarly, the numeric output from the output layer must be interpreted. In more detail, the input hard word is first converted to lower case. The ASCII values for each character is then scaled to the range -1.0 to 0.5 and all non-alphabetic characters are mapped to 1.0. The gap between the character '`z`' and the non-alphabetic characters helps to identify separators (such as digits) within hard words. Finally, each output node in the networks represents the position between the input characters. For instance, the first output node corresponds to the position between the first and second characters of the input. The value for each node in the output layer is the probability that the position it represents should contain a split. To interpret these probabilities, a *threshold* is used. If the output value is greater than the threshold, then a division marker is inserted. The threshold that produces the best results is different for each network and is determined using the success criteria discussed in Section 3.

Initially, a single network was created to split all hard words (up to 25 characters in length). However, following the work of Pomerleau, multiple specialized neural networks based on hard word length were created [13]. Thus, the algorithm for splitting identifiers is essentially a case statement where each case employs a neural network specialized to a given hard-word length. The original network (that accepts arbitrary length inputs up to 25 characters) is used as a default when one of the length-specific networks is unavailable. This occurs, for example, when insufficient training data of a particular length exists. Although each "network" represents a collection of networks (one for each input length) for presentation simplicity each is referred to as a "network".

The training data for a neural network is typically divided into three sets: a *training set*, *cross-validation set*, and *test set* [5]. The first two are used during the training of the networks, whereas the last set is used at the very end of the training process to test the network and should only be used once to report the success of the final, trained network. For each neural network studied in Section 4, the training data was divided into a training corpus and cross-validation corpus using the standard 90% / 10% split [7]. The test phase made use of the test oracle creation process as described in Section 4.

2.4 Examples

A general understanding of the differences between the output from the neural networks and the greedy algorithm is useful in gaining some intuition for the two and for interpreting the results of the next two sections. Figure 1 shows the result of splitting four different identifiers using both techniques.

In general, when the greedy algorithm performs badly it is due to limitations in the abbreviation list. For example, the greedy algorithm would have correctly split `trustdom` had `dom` (*domain*) been on the list of abbreviations. The neural network's splits depend heavily on the training set and the settings used during training. The exact

	Split Identifier	
Un-split Identifier	**Neural Network**	**Greedy Algorithm**
Both Acceptable		
rcvptr	rcv_ptr	rcv_ptr
Only Network Acceptable		
trustdom	trust_dom	trust_do_m
Only Greedy Algorithm Acceptable		
listlength	listlength	list_length
Neither Acceptable		
sockaddr	sockaddr	sock_add_r

Figure 1. Split identifiers produced by the greedy algorithm and neural network techniques.

Data Set on Network	Algorithm Performance Network {na,n2,n1}	Greedy	Random
da on na	**78.12%**	75.69%	54.59%
da on n2	70.96%	**75.69%**	54.59%
da on n1	71.56%	**75.69%**	54.59%
d2 on na	**82.04%**	81.31%	61.22%
d2 on n2	**88.12%**	81.31%	61.22%
d2 on n1	77.04%	**81.31%**	61.22%
d1 on na	**89.99%**	77.96%	78.45%
d1 on n2	**86.09%**	77.96%	78.45%
d1 on n1	**95.14%**	77.96%	78.45%

Figure 2. Correctness rates for three techniques using selected datasets.

causes for unacceptable splits are hard to pinpoint – more training data and a longer training time may have prevented these misplaced and absent splits, but there is no guarantee.

3 Experimental Setup

This section describes the source code used in the study. In particular it describes the extraction of the identifiers from the code and the generation of a test *oracle* used both to train the neural network and to evaluate the three approaches.

The source base used in the experiments includes 186 programs containing 26MLoC of C, 15MLoC of C++, 7MLoC of Java, and 21KLoC of Fortran. To produce the evaluation oracle and the neural network training data, 4,000 identifiers were randomly chosen from the 746,345, and division markers were added by four programmers.

The collection of identifiers was then divided into hard words using the original division markers. Duplicates and hard words with conflicting splits were removed. In the sequel, three sets of hard words are experimented with: da, the set of all hard words, and d1 and d2, two of the four programmer-split sets. After the creation of the oracle data, five separate neural networks were trained, one for each data set. The resulting networks discussed here are referred to as na, n1, and n2, trained on da, d1, and d2 respectively.

The output of the three techniques was compared with the expected output generating a statistic used to quantify success. It counts the number of exact matches between the actual and expected output.

4 Experimental Results

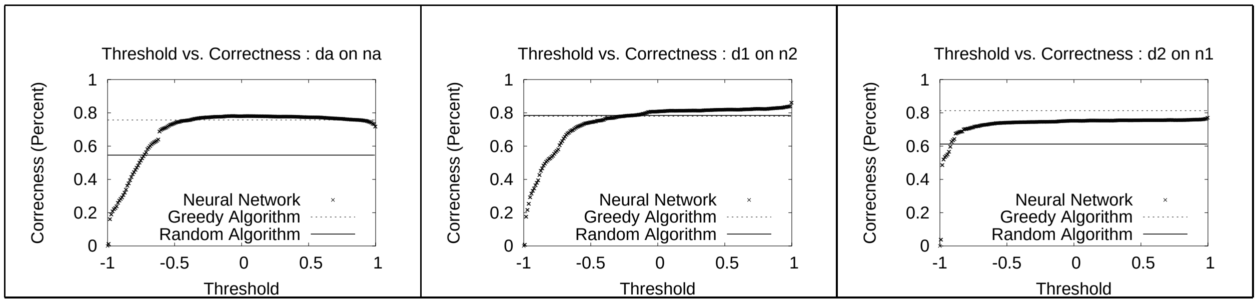
First, the three algorithms are compared in “head-to-head” in Figure 2. Because a neural network is dependent on the training data, three rows are included for each data set. For example, the first row in each grouping is the neural network trained on da (all the data). Each entry in the table is the percentage of correctly split identifiers. For each neural network, the best percentage over all thresholds is used (the influence of threshold is discussed later in this section).

From the table, using the random algorithm’s performance as a measure of data set difficulty, the data sets get easier to split further down the table. Though not by a large margin, the random algorithm actually does better then the greedy algorithm on data set d1. At the same time, the neural networks achieve the highest average correctness on this data set. An investigation into the cause of this behavior uncovered the following: data set d1 has the fewest splits. Furthermore, these splits were inserted following an intuitive understanding of the identifiers. It appears that the network n1 is doing a good job of capturing this intuitive understanding. On the other hand, the greedy algorithm does not have in its lists the abbreviations uncovered by the intuitive splitting and thus does not performs as well.

Comparing the neural networks and the greedy algorithm, the neural network out “scores” the greedy algorithm in six of the nine runs as can be seen in Figure 2. However, four of these are by less than 5%. In all cases, the neural network’s highest correctness comes from the data set on which it was trained.

The correctness scores of a neural network are dependent on a threshold (the value used to determine the binary value of the output nodes in the network). The values in Figure 2 represent the maximal correctness. To better understand the influence of threshold, Figure 3 shows the results for the full range of threshold values. Each graph plots correctness, the percentage of identifiers correctly split (according to the oracle), versus the neural network’s threshold value, which runs between -1 and 1, with a 0.005 step. The greedy and random algorithms are not affected by the threshold; thus, their correctness rates appear as horizontal lines.

The graphs for a neural network typically follow an inverted parabolic curve, which makes identification of the threshold that produces the maximal correctness possible. This pattern is shown in Figure 3a, where the maximum



(a) (b) (c)

Figure 3. Correctness rates for selected networks and the greedy and random algorithms.

value occurs toward the middle of the graph. Figure 3b shows an exception where the maximum appears on the far right. This is interesting because the far right corresponds to the *identity transformation* – a transformation that introduces *no* divisions.

The comparison from Figure 2 considers the maximal correctness for each neural network. For each dataset, the remainder of this section takes a closer look at the impact of threshold. Starting with data set da, as shown in Figure 2, na outperforms (in order) the greedy algorithm, followed by n1, n2, and finally the random algorithm. Looking at the graph in Figure 3a, na actually does best from threshold values from -0.385 to 0.880. It remains better than the random algorithm for thresholds above -0.725. Below -0.725 the low threshold means that the probably of inserting a division marker is so high that the correctness rate plummets. Being trained on da is doubt part of na's success. Recall that da is a combination of all four oracle data sets; thus, it contains artifacts and biases from four splitting methods. The na network, therefore, has been trained to pick up of these four techniques, making it a more generalized network. However, the other two networks have only been trained using data generated using a single method. Thus, when presented with one of the other data sets, these specialized networks do not perform as well.

Next consider data set d1. Unsurprisingly, as seen in Figure 2, n1 performs best, followed by na, n2, the random algorithm, and finally the greedy algorithm. Visible in Figure 3b the identity transformation, with a correctness rate of 86%, actually outperforms the random algorithm, the greedy algorithm, and n2 for lower threshold values. However, both na (when threshold is more than -0.345) and n1 (when threshold is more than -0.785) outperform the identity transformation. Interestingly, for d1, which introduces the fewest division markers, both the identity transformation and random algorithm outperform the greedy algorithm. As seen in Figure 5, the greedy algorithm introduces the most divisions, which may account for its poor performance on d1. One implication of these observations is that d1 appears easier to split than the other data sets.

Data Set / Network	Threshold	Success
d1-on-n1	-0.085	95.14%
d2-on-n1	1.000	77.04%
da-on-n1	0.630	71.56%
d1-on-n2	1.000	86.09%
d2-on-n2	0.150	88.12%
da-on-n2	0.995	70.96%
d1-on-na	0.550	89.99%
d2-on-na	0.250	82.04%
da-on-na	-0.065	78.12%

Figure 4. Success rates for selected data sets and networks.

Finally, for data set d2, n2 performs best, followed by na, the greedy algorithm, n1, and finally the random algorithm. The patterns are similar to those of d1 except for the following differences: First, as can be seen in Figure 3c, the identity transformation does not outperform the greedy algorithm. In fact, n1 is the only neural network where the identity transformation is the better for d2. Second the cross over points are slightly lower. For example, na outperforms the identity transformation when the threshold is more than -0.495 compared to more than -0.345 for d1.

Finally, Figure 4 repeats the neural network data from Figure 2 together with the threshold at which the neural network maximum occurs. Notice that the specialized networks n1 and n2 tend to do well around a threshold closer to zero when run with their respective data sets. However, when a data set other than the one used for training is run on a network, the threshold producing the maximal correctness dramatically increases.

In summary, the data presented in this section shows that the neural networks' correctness is normally better than those of the greedy algorithm. In particular in cases where the data set being used is close to the training set for

Data Set	Number of Divisions Oracle	Network*	Greedy Algorithm	Random Algorithm
da	1515	754	2282	1505
d1	242	213	637	249
d2	431	267	654	437

Figure 5. Number of divisions made by the oracle and the three algorithms for selected data sets. (*The network trained on the specified data set is used in each instance)

the network. This is in part because the networks tend to add fewer splits than the greedy algorithm, causing many hard words to be left alone, which the greedy algorithm tends to be overzealous when inserting splits.

5 Related Work

There are several research areas where identifier splitting is based solely on the division markers already present in the identifier [2, 4]. At least one project has gone beyond simple division markers. Anquetil et al. [1] analyzed soft words in file names to explore the lexical, syntactical, and semantic structure of file names.

6 Conclusion

Splitting identifiers is important because the resulting soft words are better suited for analysis by both programmers and software tools. This paper has presented three algorithms for dividing identifier hard words into their constituent soft words. The random algorithm was used as a straw man against which to compare the other two. These included a greedy algorithm and several neural networks. The neural networks performed well given specialized data created by a single person; however, performance degrades when multiple people are involved in splitting the identifiers. The greedy algorithm is consistent across data sets but generally inserts more divisions than desired.

Having split hard words (identifiers) into their constituent soft words, the next phase of this work is to apply machine translation techniques in an attempt to associate consistent meaning with each soft word. One approach is to make use of probabilistic techniques to determine the most likely meaning of the soft words.

7 Acknowledgments

This work is supported by National Science Foundation grant CCR0305330.

References

[1] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. *Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference on*, pages 84–93, 1998.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

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

[4] 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.

[5] T.G. Dietterich and G. Bakiri. Error-correcting output codes: A general method for improving multiclass inductive learning programs. *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91), Anaheim, CA: AAAI Press*, 1991.

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

[7] D.T. Jones. Protein secondary structure prediction based on position-specific scoring matrices. *J. Mol. Biol*, 292(2):195–202, 1999.

[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] D. Lawrie. *Language Models for Hierarchical Summarization*. PhD thesis, University of Massachusetts Amherst, 2003.

[10] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, pages 3–12, 2006.

[11] C. Manning and H. Schutze. *Foundations of statistical natural language processing*. The MIT Press, 1999.

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

[13] D. Pomerleau. Neural network vision for robot driving. *Early visual learning*, pages 161–181, 1996.

[14] 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.