Automated Concept Location Using Independent Component Analysis

Scott Grant, James R. Cordy, David Skillicorn
School of Computing, Queen's University
Kingston, Ontario, Canada
(scott, cordy, skill)@cs.queensu.ca

Abstract

Concept location techniques are designed to help isolate sections of source code that relate to specific concepts. Blind Signal Separation techniques like Singular Value Decomposition and Latent Semantic Indexing can be used as a way to identify related sections of source code. This paper explores a related technique called Independent Component Analysis that has the added benefit of identifying statistically independent signals in text, as opposed to ones that are just decorrelated. We describe a tool that we have developed to explore how ICA performs when analysing source code, and show how the technique can be used to perform unsupervised concept location.

1. Introduction

As software packages continue to grow in size and complexity, it becomes more difficult to understand what individual sections of the code are responsible for. With a greater amount of code available to read through, it is not always clear whether certain portions of code affect others, or if blocks of code are similar enough to warrant refactoring. Automated techniques can provide ways to help visualize the relationships between blocks of source code, and to ease the task of those working with the code base.

Concept location techniques are designed to help isolate sections of source code that relate to specific concepts [9, 8], and is also referred to as the concept assignment problem [1]. Often in these techniques, it is the user who poses the question and provides the desired concept to locate. For example, if a programmer is interested in identifying which section of the code handles the timer, or covers most of the user input, they may attempt to use a concept location technique to pull out the methods or files that are responsible.

This paper will demonstrate a novel technique for applying Independent Component Analysis to the concept location problem.

Independent Component Analysis (ICA) [3, 5] is a blind signal separation technique that separates a set of input signals into statistically independent components. It operates in a similar way to Singular Value Decomposition, which is often used in Latent Semantic Indexing and has previously been explored as a way to extract information about concepts from source code [7, 11]. The primary difference is that instead of focusing on signals that are simply decorrelated, ICA extracts signals that are mutually independent of one another. This is a stronger bound, and when used in a domain like program comprehension, can ensure a stronger difference between the extracted signals.

Independent Component Analysis involves the factorization of a source matrix comprised of a set of mixed data signals into two new matrices. One of the matrices describes a number of independent components, representing the individual extracted voices from the party described above. The other matrix is a mixing matrix, and holds information about how the independent components themselves were combined to produce the original set of mixed signals [10].

The original example of ICA as a technique is the idea of a set of microphones hung over a crowded room, wherein a number of people are engaged in conversations. If we examine the set of data obtained from the microphones, the focus on statistical independence as a bound instead of decorrelation allows ICA to isolate the original source signals, and individual voice data for each of the attendees can be recovered.

2 Idea

Independent Component Analysis is regularly described by the equation $x = As$. It factors an original data matrix x into a transformation, or mixing matrix, referred to as A, and a source signal matrix s, where the extracted independent signals are stored. If x is an mxn matrix, and we are interested in k independent signals, A will be an mxk matrix, and s will be kxn.

In this particular application, the A and s matrices listed above can be used to identify both the concepts and the relevance of documents to those concepts. The independent components extracted from the word frequency matrix show which tokens best describe a given signal. The highest and lowest points represent the relevance of tokens at the given position in the signal. For example, if the *i*th position in a single independent component is the highest value overall in that component, it can be considered the strongest contributor to the signal. This signal view is not just looking at the presence or absence of tokens, as single tokens do not necessarily indicate whether a method is or is not relevant. It is the presence or absence of the entire set of tokens that matters.

The mixing matrix offers a glimpse into how relevant a given signal is to each method. If there are *k* signals extracted using ICA, each method's row will have *k* columns corresponding to the significance of each signal. The highest value can be interpreted as indicating the strongest presence of that signal, or the most relevant topic to that method.

3 Implementation

In order to generate our ICA signals, we developed a custom application in Python. The tool runs as a series of discrete steps, piping data from one phase to the next, and an overview is shown in Figure 1.

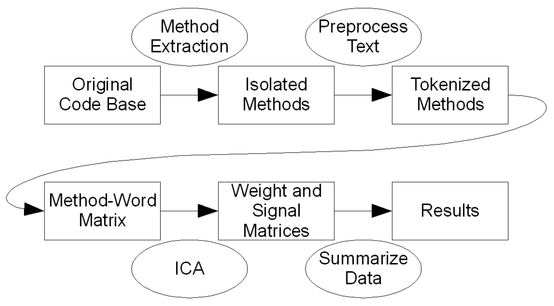

Figure 1. Implementation Method.

The input source data is separated into individual text files, which for the purposes of this experiment consist of a text file corresponding to each method in the software package we analyse. From here, a list of the unique tokens used across the application source code is generated and saved to disk, and a large method-word matrix is created. Rows of this matrix correspond to each of the application methods, and columns are binary values that correspond to the presence or absence of a token in the given method.

Next, ICA is applied to the method-word matrix, generating two new matrices. These matrices, representing the weight and signal values, are saved to disk. With the ICA matrices available, we can use the list of unique tokens generated earlier to tie the meaning of the *i*th column to the value of the *i*th token in the list. By pulling the maximal values across each signal row, for example, we are able to identify the most meaningful tokens for that given signal. In the case of method analysis, we often find these to be related namespaces, variable names, or method groups (time-related methods, for example).

Our tool provides the ability to perform dimensionality reduction using Singular Value Decomposition, in order to explicitly define the number of signals we expect to see. The primary reasons for this are to reduce the memory footprint needed, and to force ICA to focus on only the most dominant signals. For presentation purposes, it helps to clarify the results by reducing the signal space to some relatively small value. We tended to use ten to twenty-five signals, which provided us with a good idea of the primary concepts that would be detected.

If the dimensionality reduction is not specified, the tool will attempt to determine the number of signals using Principal Component Analysis (PCA) immediately before performing the ICA step. Often, the number of signals determined by PCA will be significantly higher than our default value, but the increase in signals allows for the identification of concepts that are much ”quieter”. By this, we refer to concepts that may only span over a small handful of methods, as opposed to entire files or packages. Determining the optimal number of signals is clearly subjective, however, and choosing more or fewer depends on the scope of the desired solution. In our testing, the number of signals chosen did not impact the quality of the results, only the granularity.

4 Results

4.1 Overview

To demonstrate the output generated by our tool, we looked at how it performed when analysing *cook*, a package developed in C, and spanning approximately 80K SLOC. The program was broken up into 1362 individual methods, and preprocessed to focus only on tokens greater than three characters in length. 3990 tokens were extracted, of which 3081 appeared more than once. We chose to focus on 10 signals. The time needed to perform ICA and to generate our results for the *cook* project is approximately one minute on a standard desktop machine. We use *cook* as an example because it has been used previously at the International Workshop on Detection of Software Clones, and has a clear structure from which we can check the accuracy of our results.

4.2 Signal Tokens

The token list corresponding to the 10 extracted signals can be seen in Figure 5. These values represent the tokens that in some way can be interpreted as the most relevant to the given concept. They are often, but not always, indicative of tokens that are used in methods that best fit this concept. As an example of this, Signal 3 contains tokens like *string_ty*, *str_text*, *str_free*, and *char*. When the method list that best matches this concept is examined, the top method names include *get_prefix_list*, *builtin_expr_parse_lex*, and *os_pathname*, which all spend a great deal of time manipulating strings.

While it is true that the top scoring tokens in a signal can often help identify the concept that has been discovered, it is not always the case. Signal 1, which is discussed in more detail below, is not immediately recognizable as a concept from the token list given in Figure 5. There are a few tokens that might give a hint as to what the concept is, but in general, that set of tokens appears quite general. It is important to note that the combination of tokens is what identifies a method as fitting with a given concept, and not merely the presence of the high scoring tokens.

Signal 3: Highest Scoring Methods		
Method Definition	File	Score
static int match_interpret (result, args, pp, ocp)	cook/builtin/match.c	227.80
static int fromto_interpret (result, args, pp, ocp)	cook/builtin/match.c	227.52
static int interpret (result, args, pp, ocp)	cook/builtin/execute.c	223.57
static int interpret (result, args, pp, ocp)	cook/builtin/filter_out.c	222.45
static int match_mask_interpret (result, args, pp, ocp)	cook/builtin/match.c	221.87
static int script (result, args, pp, ocp)	cook/builtin/write.c	213.39
static int prepost_interpret (result, args, pp, ocp)	cook/builtin/text.c	212.23
static int interpret (result, args, pp, ocp)	cook/builtin/options.c	211.06
static int in_interpret (result, args, pp, ocp)	cook/builtin/boolean.c	210.70
static int or_interpret (result, args, pp, ocp)	cook/builtin/boolean.c	210.06

Figure 2. Signal 1: Highest Scoring Methods

Figure 3 shows the token value distribution for Signal 1, which discovered a set of *interpret* methods that shared many similarities above. There are a few clearly defined peaks that stand out, and correspond to tokens that are often good indicators that a given method is related to the concept represented by this signal. Many of the tokens actually lie around zero, and do not provide much explanatory value about the concept.

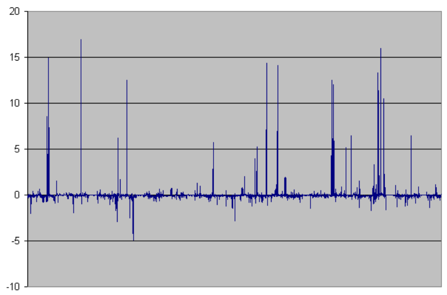

Figure 3. Signal 1 Token Value Distribution

4.3 Signal Methods

Figure 2 provides the top ten scoring methods for Signal 1. The methods score fairly closely to each other, and all methods deal with the interpreter in some way. One particularly interesting feature about the methods that are discovered is that they are not local to a single file, but rather span across a specific subset of the codebase. Each of the methods that score highest is found within the *builtin* folder, and they share a great deal of similarity, from number and type of parameters, to their actual purpose. The methods share a similar concept, and can be extracted as a group.

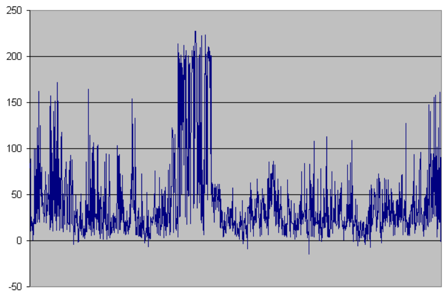

Figure 4. Signal 1 Document Scores

In Figure 4, the overall scores given to the methods for Signal 1 are shown. Our method extraction worked in an orderly depth-first way, so data about methods that are close in the directory tree structure will be placed in rows that are close together in the matrix. The large bump in scores around the 490-550 range correlates to the *builtin* folder, referenced above. The primary methods are very closely located in the code, which makes sense conceptually.

What is clear from the method list given in Figure 2 is that the functions given are conceptually related in some way. In fact, by looking at the scores between the methods, we often get a rough gauge on the similarity of the methods as well. The only real differences are a few single word or token changes, and a small difference in the assignment to the variable *s*. In the cases we've examined so far, methods that receive the same score are often Type 1 clones, and methods that score very closely are often Type 2 or Type 3 clones. This is an unintended, but very fortuitous side-effect of the technique.

Signal 10 contained a wider range of values, and didn't appear to narrow down to one single concept. However, when investigating the results, many small groups of related methods can be observed. One possible explanation for this lies in a fundamental assumption made when using ICA. In an ICA, at most one of the resulting signals can be Gaussian. Such a signal often describes keyword distributions that resemble noise. For example, tokens like *else*, *sizeof*, or *strlen* would be expected to be quite common, and not necessarily indicative of any one given feature or concept.

5 Analysis of Results

The *cook* software package gave us a good opportunity to determine whether or not ICA was a viable method to identify concepts in source code. By limiting the number of signals, we were able to demonstrate that we could identify a number of key concepts used in the package. We intend to continue analysis in this vein, including measuring the effects of limiting the number of signals. While it makes it much clearer to identify major concepts used in the code, it is difficult to determine what the right granularity is. It can be tricky to find the optimal number of signals to extract from the input matrix. Often, if the granularity is too small, concepts will get mashed together in one signal, and may appear to clutter up the data.

When we analysed the signals that were extracted from the *cook* source code, each signal (except for the noise signal identified above) gave a representation of a concept, a set of methods related in some way. Additionally, extremely similar method scores in a signal can act as a way to identify potential clones.

6 Related Work

ICA has been used successfully in natural language topic detection, by considering each extracted signal as a topic [6, 2]. We have also used the technique successfully to segment large document sets in the same way, identifying the major topics that are used [4]. LSI has seen more use in extracting information from source code [9, 8], and shares many similarities. However, due to the difference in assumptions about the nature of the signals, ICA may be better suited to concept detection, as it expects statistically independent signals. This stronger bound should force results that are significantly more distinct than techniques that assume decorrelated signals.

Aspect Mining attempts to identify candidate aspects in existing systems, allowing them to be extracted and isolated [12]. The strong bound of statistical independence may also allow ICA to perform well in this domain, where it processes across the entire code-base to find strongly-related sections.

7 Future Work

Although the technique is powerful, and also generalizable, we would like to gain a better awareness of what the signals themselves describe. For example, what does it really mean when methods receive low scores in a signal? In many cases, the inverse of the signal holds a great deal of information, along with methods that are themselves related.

We are currently using only the token values found in the code, such as the method and variable names. There are many other function metrics that could be leveraged in order to get a more accurate description of the method as a row in our matrix. Performing tests to determine which metrics are most valuable would move our method-token matrix to a method-feature matrix, and would ideally provide better results.

8 Conclusion

Independent Component Analysis offers an interesting way to analyse source code, with a unique focus on distinct concepts. By focusing on the statistically independent concepts, it is possible to identify strongly-related threads of information used in the code. Additionally, the technique is language-independent, and offers a great deal of flexibility with respect to the granularity of the signal count. Our analysis has so far shown that the identification of concepts using ICA identifies related methods that are not limited to proximity in the code, and at the most basic level, requires very little preprocessing. Additionally, an added bonus of the technique is that in source code collections with type 1 through 3 clones, these clones share similar scores in signals, and can be detected by a quick parse through the scores of each signal.

Acknowledgment

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada and by the Ontario Graduate Scholarship Program.

References

[1] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the con-

cept assignment problem. *Commun. ACM*, 37(5):72–82, 1994.

[2] Ella Bingham, Ata Kabán, and Mark Girolami. Topic identification in dynamical text by complexity pursuit. *Neural Process. Lett.*, 17(1):69–83, 2003.

[3] P. Comon. Independent component analysis, a new concept? *Signal Processing*, 36(3):287–314, 1994.

[4] Scott Grant, David Skillicorn, and James R. Cordy. Topic detection using independent component analysis. In *LACTS 2008, Workshop on Link Analysis, Counterterrorism and Security*, pages 23–28, Atlanta, GA, USA, 2008.

[5] Aapo Hyvarinen, Juha Karhunen, and Erkki Oja. *Independent Component Analysis*. J. Wiley, New York, 2001.

[6] T. Kolenda, L. Hansen, and J. Larsen. Signal detection using ica: application to chat room topic spotting, 2001.

[7] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243, New York, NY, USA, 2007. ACM.

[8] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[10] David Skillicorn. *Understanding Complex Datasets: Data Mining with Matrix Decompositions*. CRC Press, 2007.

[11] Pieter van der Spek, Steven Klusener, and Pirre van de Laar. Towards recovering architectural concepts using latent semantic indexing. In *CSMR*, pages 253–257. IEEE, 2008.

[12] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring, 2003.

Signal Values with Scores			
Signal 1		**Signal 2**	
const	16.932740	source	27.408585
string_list_ty	15.999061	return	23.220881
atic	14.921768	char	15.549450
nstrings	14.374434	atic	9.927489
opcode_context_ty	14.078833	const	5.206306
string	13.272311	sizeof	4.965227
expr_position_ty	12.512504	this	3.868243
result	12.496964	unsigned	3.831395
return	11.980626	void	3.636333
struct	10.514301	errno	3.543781
Signal 3		**Signal 4**	
string_ty	29.923893	trace	31.085935
str_text	21.344953	long	25.820150
str_free	17.010397	08lx	21.071840
str_from_c	11.592439	this	7.334605
char	11.345406	source	5.721549
static	9.230515	method	5.525806
else	7.185923	str_text	5.127350
source	6.215664	assert	4.366374
str_length	5.996619	expr_position_ty	3.985932
str_copy	5.097120	opcode_ty	3.632044
Signal 5		**Signal 6**	
sub_context_ty	21.407264	size_t	21.914410
i18n	20.294338	source	18.906863
sub_context_new	19.942019	return	9.858660
sub_var_set	19.009613	sizeof	9.197594
sub_context_delete	15.161536	assert	8.443668
file_name	11.482302	filename	6.426897
filename	8.382733	length	5.874415
error_intl	8.056013	stmt_ty	5.812159
fatal_intl	7.351470	mem_change_size	5.653551
name	6.516519	list	5.317410
Signal 7		**Signal 8**	
break	19.388269	void	31.688888
case	18.895007	atic	28.247702
switch	18.895007	source	6.391675
default	13.725450	destructor	5.077392
char	9.370931	trace	5.053962
continue	8.848111	this	4.043551
else	6.970506	that	2.772687
goto	6.963573	fingerprint_ty	2.769293
while	5.350315	expr_position_destructor	2.275731
status	4.418778	symtab_ty	1.912480
Signal 9		**Signal 10**	
this	24.473886	text	7.528535
stmt_ty	11.062964	else	7.060219
opcode_ty	10.927282	sizeof	6.775935
status	8.536064	line	6.659413
opcode_status_ty	8.347982	strlen	6.407418
opcode_status_success	8.256018	stack	6.214723
opcode_context_string_list_pop	6.666987	strcpy	6.186951
string_list_delete	6.665052	state	6.130309
opcode_status_name	6.506309	strcat	6.092176
value	6.439897	count	6.025283

Figure 5. Signal Tokens