Vector Space Analysis of Software Clones

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

Abstract

In this paper, we introduce a technique for applying Independent Component Analysis to vector space representations of software code fragments such as methods or blocks. The distance between these points can be determined, and used as a measure of the similarity between the original source code fragments they represent. It can be reasoned that if the initial matrix representation contains enough information about the syntactic structure of the source code, the vector space representation will be sufficient to predict the similarity of fragments to one another, and can provide the likelihood that the code is a clone.

1. Introduction

Reuse of software code fragments by copy/paste/edit is a common software development practice that leads to a large number of similar code segments, or *code clones*, in software systems [1, 15]. Code clones can cause problems for software maintenance and evolution [9, 12], making them a popular topic in software comprehension [5].

In the following paper, we introduce a method for using existing Information Retrieval methods such as Independent Component Analysis to analyse vector representations of software methods. The vector representations of a software package can be examined for their proximity to each other, where the distance between any two vectors can be considered a measure of their similarity. It can be reasoned that if the initial matrix representation contains enough information about the syntactic structure of the methods, the vector representation will be sufficient to predict the similarity of methods to one another.

Independent Component Analysis (ICA) [3, 8] is a blind signal separation technique that separates a set of input signals into statistically independent components. The primary difference between ICA and Latent Semantic Indexing (LSI) 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, and a correspondingly stronger similarity between fragments with similar signal profiles. ICA 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, and the other is a mixing matrix that holds information about how the independent components themselves were combined to produce the original set of mixed signals.

2. Background

Latent Semantic Indexing (or Latent Semantic Analysis, as it is also commonly referred to) was introduced in 1990, and was described as a way to take advantage of higher-order structure in the association of terms with documents in order to improve the detection of relevant documents on the basis of terms found in queries [4]. These term-associations can be interpreted as the semantic structure of the document set. By assuming that some latent semantic structure exists in a set of documents, the problem of term association can be treated as a statistical problem. While LSI has been tremendously useful, it operates under a slightly weaker statistical bound than another IR technique, Independent Component Analysis (ICA).

ICA is a blind source separation method designed to extract the statistically independent components of a non-Gaussian source signal. It is described by the equation $x = As$, and 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 $m \times n$ matrix, and we are interested in k independent signals, A will be an $m \times k$ matrix, and s will be $k \times n$ [17].

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 rather than decorrelation allows ICA to isolate the original source signals, and individual voice data for each of the attendees can be recovered.

Our previous work in this area [6] has shown that individual concepts found in software can be isolated using

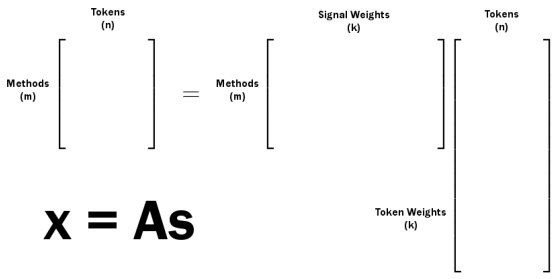

Figure 1. ICA Matrices

this technique. As ICA identifies the signals that are statistically independent from one another, we can be confident that each axis in the vector derived from our original representation is a measure of some significantly different attribute. We believe this results in a clearer conceptual division between methods, and better results when determining syntactic and semantic similarity. With this information, we demonstrate how ICA can be applied to the problem of locating clones in software, using a method-token matrix to represent the set of code blocks in our input source code. The resultant signals can be used to transform the original representation of the code in vector space into a new set of data based on the magnitude of the statistically independent concepts in each code block.

We use the following definitions throughout, which are clarified here for convenience.

- **Method-token matrix**: The matrix generated from the input source code, where each row corresponds to a single function, method, or code block, and each column corresponds to the presence of a token in that code. For example, we expect to see a 1 at position M_{ij} if the ith method in our source contains the jth token in an ordered list of tokens that span the corpus. This is also known as a document-term matrix.
- **Vector space**: An n-dimensional space in which representations of the code blocks we analyse are stored. Initially, n is set to the number of non-unique tokens spanning the input corpus, as each axis in the vector corresponds to the presence of a token. We run ICA on a matrix with reduced dimensions, and for ease of visualization, we plot the final matrix points in a three-dimensional space.
- **Nearest neighbour score**: The distance between a point and its nearest neighbour. For our purposes, this is a metric that can be used as the likelihood that a method has a clone. If two points are close, it is likely that those methods are very syntactically or semantically similar. Conversely, if two points have a very large nearest neighbour score, they do not share similar features, and are probably not clones of one another.

3. Method

In order to identify the most similar code blocks in our source data, we take the following steps:

- Construct a method-token matrix using the non-unique tokens found in our source code.
- Reduce the dimensionality of our matrix using SVD.
- Apply ICA to our reduced matrix, and save the results.
- Generate a new matrix based on ICA's valuation of the token relevance in order to identify the points in the new vector space that correspond to our input data.
- Calculate the nearest neighbour scores of each method using the previous matrix.

The source package to be analysed is segmented by method (or any other code fragment unit), and a list of the non-unique tokens used across the application is generated. We define non-unique tokens as those tokens that appear more than once, and therefore can contribute to some correlation between methods. A method-token matrix is generated from the application methods and the list of non-unique tokens. The presence of a token is represented by the value 1, and the absence of a token by the value -1.

To provide an example of the method-token matrix we use as input, consider the following example.

$s_1 =$ My dog has fleas.
$s_2 =$ That dog has fleas.
$s_3 =$ My ukelele has fleas.
$s_4 =$ My team won the football game.
$s_5 =$ That dog ate all the turkey.

We have five input documents (or methods, when parsing source code), named s_1 through s_5. In total, there are six tokens used in the input, of which five are non-unique. Since the token 'a' is only used in s_2, it is only relevant for s_2 and can be omitted, as it is outside the scope of tokens we are interested in using in our comparison.

$tokens = \{$all, ate, dog, fleas, football, game, has, my, team, that, the, turkey, ukelele, won$\}$
$non-unique = \{$dog, fleas, has, my, that, the$\}$

Our input matrix, here referred to as x_{flea}, will necessarily be a 5×6 matrix, with the five rows representing the input documents s_1 through s_5, and the six columns representing the non-unique tokens above in the order they are encountered when parsing the input documents.

$$x_{flea} = \begin{bmatrix} 1 & 1 & 1 & 1 & -1 & -1 \\ 1 & 1 & 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & 1 & -1 & -1 \\ -1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & -1 & 1 & 1 \end{bmatrix}$$

The representation of the original source code as a method-token matrix now allows us to use a technique like

	s_1	s_2	s_3	s_4	s_5
s_1	X	5.81	5.52	7.93	9.71
s_2	5.81	X	8.64	11.79	9.62
s_3	5.52	8.64	X	9.62	13.98
s_4	7.93	11.79	9.62	X	8.64
s_5	9.71	9.62	13.98	8.64	X

Figure 2. DV_{flea} nearest neighbours

ICA to pull out some interesting results. Initially, we use SVD as a dimensionality reduction method in order to reduce the memory footprint needed, and to force ICA to focus on only the most dominant signals. After the size of the vector space has been reduced, we use ICA to identify the statistically independent components of the input matrix.

It may help to think of the application of SVD and ICA as a way of remapping the original matrix into a new vector space, where each axis corresponds to some mathematically significant concept or feature found in the original matrix. It is important to note that the use of ICA over a technique like LSI results in statistically independent components, and should result in axes that are more unique than those generated from components that are merely decorrelated. Another interesting way to consider the results is to map to three dimensions and to plot the results; we do this to identify clusters visually that may be clone groups, along with outliers that are unlikely to have similar blocks of code.

In this application, the most useful application of this new data is the similarity between the documents in our original matrix. By using ICA to map our source matrix into a reduced vector space, with axes that correspond to some mathematically derived and independent feature of the data, we can then use the results to see how close each method is to each other one. LSI itself uses SVD transform the original document-term matrix into a decomposition of matrices used to identify relationships between the source data. We take a similar approach here.

ICA is defined above as $x = As$. If the rows and columns of x are documents and tokens, and the rows and columns of s are signals and tokens, we can generate a new document value matrix DV using the following equation:

$$DV = xs^T$$

The logic for this comes from the fact that ICA has done the work of figuring out which terms are semantically close. By taking the product of our source matrix with raw token availability and our derived signal-token matrix, the relationship between methods becomes apparent.

It is from this document value matrix that other LSI-related activities like querying can be performed. We have demonstrated in previous research [6, 7] how each axis corresponds to some independent concept, and can be used individually to show which sections of code correspond to

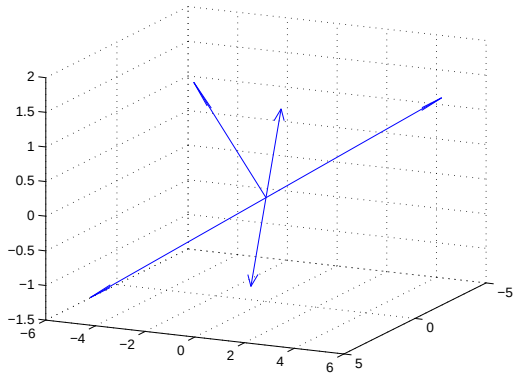

Figure 3. DV_{flea} vector plot

each concept. When the axes are used together, we have a better overall view of the full similarity of each block.

After the matrix x_{flea} above has been processed using ICA, we generate the document value matrix, here called DV_{flea}, which looks like this:

$$DV_{flea} = x_{flea}s_{flea}^T = \begin{bmatrix} 2.47 & -1.82 & 2.03 \\ 5.34 & 2.38 & -0.78 \\ 3.28 & -5.98 & -1.51 \\ -5.34 & -2.38 & 0.78 \\ -3.28 & 5.98 & 1.51 \end{bmatrix}$$

If we treat each row as a distinct point in three-dimensional space, the distance between each of these points gives us the similarity of each document to the others. In this case, we can show each source document, and each of the nearest neighbours with their scores. In this way we get an ordered list of related documents spanning the entire range of the document set.

The strings and their nearest neighbours can be seen in Figure 2. Each score is the Euclidean distance between the points in three-dimensional space. The dimensionality of the space is somewhat subjective, and we have chosen to use three dimensions here to better display the meaning of the results visually. As an example, Figure 3 shows how the points map when plotted as vectors. By plotting in three dimensions, we get an immediate sense of the placement of the points relative to each other. As ICA enforces a strong statistical bound on the axes, we expect to see points that are quite distinct from one another. This is demonstrated by the significantly different orientations of the vectors.

The meaning of these results is as follows. By applying ICA to the original method-token matrix generated from our input source code, we can derive a matrix *DV* that represents the strength of each document in a new vector space. The rows of *DV* can be plotted as points in this vector space, and the Euclidean distance between any two points can be interpreted as a measure of their similarity, since each axis in this new space corresponds to the strength of some statistically independent concept.

```
static unsigned long source_load (int cpu, int type)
    struct rq *rq = cpu_rq (cpu);
    unsigned long total = weighted_cpuload (cpu);
    if (type == 0) return total;
    return min (rq->cpu_load[type - 1], total);


static unsigned long target_load (int cpu, int type)
    struct rq *rq = cpu_rq (cpu);
    unsigned long total = weighted_cpuload (cpu);
    if (type == 0) return total;
    return max (rq->cpu_load[type - 1], total);
```

Figure 4. First Percentile Nearest Neighbour

4. Results

To demonstrate the effectiveness of our approach, we look at how well it identifies clones in the Linux *kernel* directory (hereafter referred to as *kernel*). This source code is developed in C, and spans nearly 40,000 SLOC. The source is segmented into 2731 individual methods, and was preprocessed to only include tokens greater than three characters in length. 9327 tokens were extracted, of which 6828 appeared more than once. The time needed to perform ICA and to generate our results for *kernel* is approximately one minute on a standard desktop machine.

If two methods are found to be very similar, the points in three-dimensional space derived from applying ICA to our input matrix will be extremely close. Since we are treating each signal in the matrix derived from ICA as some conceptual meaning, it means that for whatever those concepts are determined to be, the two methods share that similarity. In our tests, this has frequently (but not necessarily) demonstrated itself by correlating to some subset of the code functionality. For example, it may be the case that one signal has high values for methods that handle memory management. These conceptual axes are determined automatically when ICA is applied, and are not seeded or predetermined by the person analysing the matrices.

While it is not true that in any example, a score below a certain constant threshhold is a definite identifier of a clone, there is a very clear ordering in structure that can be observed when looking at the matches. A nearest neighbour score for a point in the first percentile means that the code fragment represented by that point is within the top one percent of potential clone candidates.

A sample clone, as indicated by its nearest neighbour score, is the method *source_load* seen in Figure 4. When we have calculated the distances between the *source_load* point and every other point corresponding to methods in our source code, it can be seen that the point for the *target_load* method is extremely close. Visually, these methods share a great deal of similarity, and the only differences are in the method name and in the usage of *min* or *max*.

Figure 5 is an interesting example, as there is a great deal of structural similarity, but several key semantic differences. In both cases, the *entry* variable is declared and

```
static int __init kallsyms_init (void)
    struct proc_dir_entry *entry;
    entry = create_proc_entry
        ("kallsyms", 0444, NULL);
    if (entry) entry->proc_fops =
        &kallsyms_operations;
    return 0;


static int __init ioresources_init (void)
    struct proc_dir_entry *entry;
    entry = create_proc_entry
        ("ioports", 0, NULL);
    if (entry) entry->proc_fops =
        &proc_ioports_operations;
    entry = create_proc_entry
        ("iomem", 0, NULL);
    if (entry) entry->proc_fops =
        &proc_iomem_operations;
    return 0;
```

Figure 5. Tenth Percentile Nearest Neighbour

assigned using *create_proc_entry*, a comparison is made to *proc_fops*, and the value 0 is returned. However, a different string value and file system mode is given, different constants are used, and the *proc_fops* comparison is made twice in *ioresources_init*. These methods are probably not clones of each other, and we have no reason to assume that a nearest neighbour score in the tenth percentile would indicate a clone in *kernel*, but as a similar pair, they have been determined to be more alike than 90% of the other methods in the source code. It seems like a fair claim, when looking at the actual structure of the two methods in question.

We believe this is one of the interesting strengths of this method; it may not provide a boolean truth test for whether or not two blocks of code are clones of each other, but it can provide an estimate on the likelihood that the methods are clones relative to the rest of the source, with great certainty.

Although it is true that a number of methods have very low nearest neighbour scores, there is often a very marked difference between the nearest neighbour and the second nearest neighbour (or rather, between potential clones and the non-clone pairs). In fact, for the *kernel* example, scores beyond the nearest neighbour often jump up by factors of a thousand. If the distance between a first percentile match is 0.001, the second nearest neighbour will often have a score around 3.0 or 5.0. Again, these are relative scores, but the points that are not considered similar are considerably farther apart than those that are considered potential clones.

This phenomenon continues on throughout the analysis, and means that the relative scores need not act as a hindrance against determining which code fragments are clones. Although it is not possible to give a magic number that acts as a cutoff value when identifying clones by their nearest neighbour scores for all inputs, we can demonstrate that a meaningful gap appears between a source point and any non-clone points.

By analysing the nearest neighbour score for each method, we can also identify outliers that are far away

from other methods, and therefore bad candidates for clone matching. Although the nearest neighbour distance score is subjective between corpora, it is not unreasonable to say that the values that are farthest apart can be excluded outright. Common culprits for outliers that match very poorly with other methods are usually large singular functions that could arguably benefit from being refactored themselves. It is not the case that overuse of tokens in a method will result in high similarity scores to other methods; rather, the absence of tokens shared between two methods will adversely affect the distance score.

5. Related Work

ICA has been used successfully in natural language topic detection, by considering each extracted signal as a topic [2, 10]. In previous work we have used the technique successfully to segment large document sets in the same way, identifying major topics [7]. Recently we have experimented with using ICA for concept analysis in software systems [6], leading to the present work.

Clone detection is a popular area with a wide range of methods proposed [1, 11, 16]. The work closest to our is possibly that of Marcus and Maletic [13, 14] who have applied latent semantic indexing (LSI) to find similar code segments. However, their approach limits the tokens to identifiers and comments, ignoring keywords and structural symbols, whereas ours ignores comments and depends on keywords, yielding a more structure-oriented result.

The cleaned and pretty-printed function/method source documents used in our work are taken from a repository created by Roy and Cordy in their work on analysis of function clones in large scale open source systems [15].

6. Conclusion and Future Work

Using a technique like ICA appears to work well at identifying similar methods in source code, without any required built-in knowledge about program language or syntax. By mapping the methods to vectors using a method-token matrix and applying ICA to extract the statistically independent components that correspond to the original dataset, we can use a distance metric to determine how similar the original methods are to each other. Further, this gives us a way to estimate the possibility that these methods might be clones of one another.

The size of the matrices determined has been problematic, and running ICA on a matrix with several hundreds of thousands of methods can be prohibitive. We are looking at ways to reduce the size of the data in a meaningful way in order to tackle the problem of extremely large sets of data.

Additionally, in this work we have focused on a raw distance metric rather than cosine similarity; looking at the vector space using the cosine distance might help isolate similar features found in methods, as opposed to the overall similarity of each document. We have used a similar approach previously in order to identify conceptually-related methods [6, 7].

Acknowledgments

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] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[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 C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[5] R. Koschke et al. *3rd International Workshop on Sofware Clones*. Kaiserslautern, Germany, 2009.

[6] Scott Grant and James R. Cordy. Automated concept location using independent component analysis. In *WCRE 2008*, pages 138–142, Antwerp, Belgium, 2008.

[7] Scott Grant, David Skillicorn, and James R. Cordy. Topic detection using independent component analysis. In *LACTS 2008*, pages 23–28, Atlanta, GA, USA, 2008.

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

[9] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE 2009*, page (to appear), Vancouver, Canada, 2009.

[10] T. Kolenda, L. Hansen, and J. Larsen. Signal detection using ICA: application to chat room topic spotting. In *ICA 2001*, pages 540–545, San Diego, CA, USA, 2001.

[11] Rainer Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007.

[12] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, March 2006.

[13] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *IWPC 2005*, pages 33–42, St. Louis, Missouri, USA, 2005.

[14] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *WCRE 2004*, pages 214–223, Delft, Netherlands, 2004.

[15] Chanchal K. Roy and James R. Cordy. An empirical study of function clones in open source software. In *WCRE 2008*, pages 81–90, Antwerp, Belgium, 2008.

[16] Chanchal K. Roy and James R. Cordy. Scenario-based comparison of clone detection techniques. *ICPC 2008*, pages 153–162, 2008.

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