Vector Space Analysis of Software Clones

Scott Grant, James Cordy
School of Computing, Queen's University
Kingston, Canada

Introduction

Software Clones

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. Code clones can cause problems for software maintenance and evolution, making them a popular topic in software comprehension.

Our approach

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.

Background

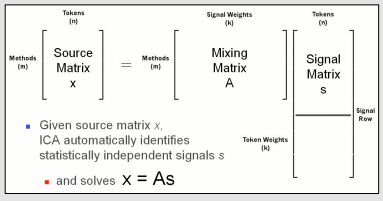
Vector space

An n-dimensional space in which representations of the code blocks we analyse are stored. We start with a matrix generated from the input source code, where each row corresponds to a single 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 method *i* in our source contains token *j* in an ordered list of tokens that span the corpus.

Independent Component Analysis

ICA 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 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.



Explanation of Method

Identifying Clones

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 to transform the original document-term matrix into a decomposition of matrices used to identify relationships between the source data. We can use the earlier definition of ICA as ***x = As*** to do something similar. 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.

Ordering Code Blocks by Similarity

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. In this way we get an ordered list of related documents spanning the entire range of the document set.

Each score is the Euclidean distance between the points in three-dimensional space. 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, and demonstrated by the significantly different vector orientations.

Method Summary

Step 1
Construct a method-token matrix using the non-unique tokens found in our source code.

Step 2
Reduce the matrix dimensionality using SVD.

Step 3
Apply ICA to our reduced matrix, save the results.

Step 4
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.

Step 5
Calculate the nearest neighbour scores of each method using the previous matrix.

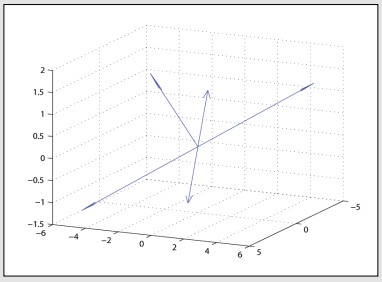
Example

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.

non-unique = {dog, fleas, has, my, that, the}

Our input matrix will necessarily be a 5x6 matrix, with the five rows representing the input documents s_1 through s_5, and the six columns representing the non-unique tokens.

After the matrix has been processed using ICA, we generate the document value matrix using $DV = xs^T$. The figure below shows how the points map when plotted as vectors.



An Ordered List of Similar Methods

This technique can provide an estimate on the likelihood that code blocks are clones, relative to the rest of the source, with great certainty.

```
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);
}
```

First Percentile Nearest Neighbour

```
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;
}
```

Tenth Percentile Nearest Neighbour

Conclusion

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.