Using Classification Techniques to Determine Source Code Authorship

Brian N. Pellin
Computer Sciences Department
University of Wisconsin
1210 Dayton Street
Madison, WI, USA 53706-1685
Email: bpellin@cs.wisc.edu

Abstract

The ability to test authorship of source code is a useful technique. We present techniques for using statistical machine learning to accomplish this task. We translate source code into abstract syntax trees and then split up the trees into functions. The tree for each function is considered a document, with a given author. This collection is fed to an SVM package using a kernel that operates on tree structured data. The classifier is trained with source code from two authors, and is then able to prediction which of the two authors wrote a new function. We achieve between 67% and 88% classification accuracy over the set of programs we examined. This demonstrates that our techniques are useful either alone, or in concert with other methods.

1 Introduction

The ability to identify the author of a program is useful for a variety of reasons. Intellectual property is one of today's biggest buzz words. The ability for an author to demonstrate that another organization is using a program that he wrote is useful. Additionally, it is valuable to have methods that can help an author prove that he wrote a program.

Statistical machine learning techniques have been applied successfully to many natural language problems. However, these techniques do not need to be limited to natural languages. They can be used to learn about artificial languages as well. Even though programming languages are in many ways more restrictive than natural languages, there is still a large degree of freedom in how functionality is implemented[14]. Authorship classification is possible, because different authors will make consistent use of different programming styles.

A variety of previous work has examined source code classification for many different purposes including authorship attribution. However, most of these techniques involved extracting features from the source code such as variable names and keyword frequencies. These techniques derive a flat list of features, but they throw away most of the syntactic information present in the source code.

In order to utilize the syntactic structure, we will not examine raw source code. Instead, we investigate the abstract syntax trees for programs. We'll feed the AST's to a tree based kernel machine (a variant of support vector machines). Additionally, we break up our AST's at the granularity of a function. The size of a function is convenient, because they are small enough to generate a large number of examples, yet large enough to contain a reasonable amount of information.

Our classification problem is set up as a binary classification between two authors. We train our classifier by using functions from two similar programs written by different authors. Then, this classifier can be used to predict which of the two authors wrote an unlabeled function. Using this technique we are able to achieve good results. Our datasets achieve between 67%- 88% accuracy, and we are able to achieve this data with reasonably sized training sets.

The rest of this paper proceeds as follows: In section 2 we examine the works of others in this area and describe how our work relates. Section 3 describes how we use abstract syntax trees to represent programs, and section 4 explains how the AST's are used in the classifier. Then, we test our classification scheme in section 5 and discuss the results. Finally, we draw conclusions in section 6.

2 Related Work

Authorship analysis is a field that has been studied by many in many different contexts. A good example of a classic question of authorship, is that of Shakespeare's works. Several academics have questioned whether all of the works traditionally credited to Shakespeare were truly written by him[15]. Elliot attempted to use computers to determine the authorship of Shakespeare's writings[12]. He examined word frequency, punctuation, and other language details for consistency throughout the documents.

In addition to searching natural language text for authorship information, several people have attempted to apply similar techniques to determining the authors of computer programs. In general, most of these studies have focused on learning from high level source code. Usually, it does not make sense to look at machine code for authorship information since most programmers do not write machine code directly. Additionally, modern optimizing compilers perform enough program transformations that stylistic differences between programmers are unlikely to be preserved in machine code.

Ugurel et al were not interested in authorship, but they did apply natural language processing techniques to source code[17]. They looked at several open source programs that were generally available. Then, they extracted all of the alphabetic words from the project's source code and documentation. They used these features to train an SVM classifier. The training sets were set up to determine the programming language the source was written in, as well as, the general topic that the program fit into[1].

Krsul and Spafford invested methods of extracting "fingerprints" from source code[14]. They created a large number of metrics that they would extract from source code and used them to form a feature list. Some examples of metrics they used are: average line length, average variable name length, ratio of `while` to `do` to `for` loops, etc. The idea was that in combination these metrics would serve to identify different programmers. In our work, we ignore most of these metrics and focus on syntactic structure alone. It is very likely though that these techniques could be combined with ours to build an even better classifier.

We use the classifier that was implemented by Moschitti[16]. The classifier is a modification of SVM-light[13] to support a tree based kernel. Traditional SVM methods operate on a flat feature set. The input data typically takes the form of a vector of real numbers. Instead, Moschitti's kernel operates on data in a tree structure. He studied the effectiveness of applying his tree based classifier to parse trees of the English language, this way his model would include information about parts of speech of each word occurrence. We use the same classifier to examine a program's abstract syntax tree.

[1] Such as databases, games, math, networking, or word processors

Source Code

```
System.out.println("Num=" + (5 * 6));
```

Abstract Syntax Tree

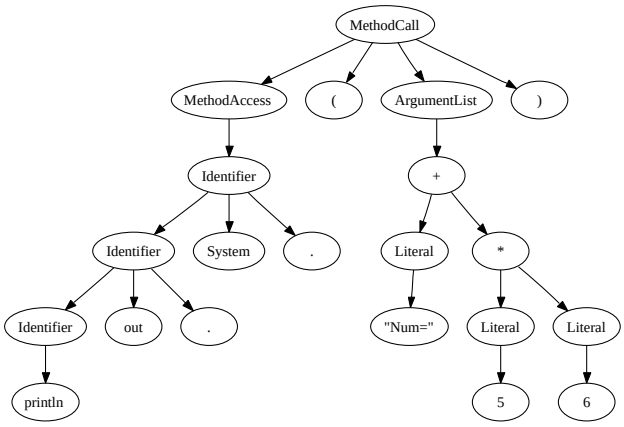

Figure 1: **Source Code / AST** *A fragment of source code, and its corresponding abstract syntax tree. Note: Some single parent, single child internal nodes have been trimmed to make the diagram more readable.*

This is only a small sample of the work that has occurred in this large research area. Our work differs in the fact that we directly use a program's tree structure to represent it. We predict that the best authorship tools will evolve from the combination of several different methods.

3 Viewing Source Code as Abstract Syntax Trees

When we analyze programs, we examine the program's abstract syntax tree. An abstract syntax tree is a finite tree, in which the internal nodes are labeled by operators, and the leaves are operands[9].

Java is our language of choice in performing our experiments. It has a few nice properties. First, there is a large amount of publicly available software written in Java. This software gives us a wide variety of programs on which to test our classifier. Additionally, it is significantly easier to parse than C++, and it does not use a preprocessor like C or C++. The example in Figure 1 shows the abstract syntax tree for a simple piece of source code.

We use this representation for a number of reasons.

First, the AST is a natural way to view a program. The AST is completely agnostic of spacing in the original text of the source code. The patterns in the spacing of the original code may be a good indicator of who wrote the source code, because different programmers will use different styles. However, spacing patterns are trivial to destroy without altering the program. One could simply eliminate all unnecessary whitespace or run the program through a pretty printer and any trace of authorship present in the spacing would be destroyed.

Another good reason to use AST's as our documents, is that our classifier is exposed to the entire program structure. Most of the classification methods discussed in Section 2 threw out most of the program structure during feature extraction. These methods would extract features like term counts or some small scale elements of program structure. Our method gives us the ability to look at the structure of the entire program.

Additionally, we have chosen to break trees up at the granularity of methods. So, each AST in our training set represents one method. This is a useful break down, because it increases the number of documents in our collection and reduces the complexity and size of each document. Yet, functions yield large enough AST's to contain rich information. Additionally, we suspect that most programmer specific patterns in programs exist within methods. However, exploring trees at different granularities would be an interesting avenue for future work. Smaller granularities might also make it easier to study programs in which multiple authors have written distinct parts of a method.

4 Authorship Classification of Abstract Syntax Trees

Once we have transformed the source code into abstract syntax trees, the next step is to make use of the tree in classification. We use a kernel machine that is designed to operate on data in a tree structure. A collection of abstract syntax trees are then used to train our classifier. Finally, the trained classifier is used to predict the authorship of other AST's.

4.1 Tree Kernel

Most traditional forms of SVM learning look at documents with a flat representation. These flat representations typically take the form of a vector of features. Flat representations work well in many cases, but, it may not be possible to represent all of the structural properties of a program in a flat representation.

Kernel functions provide a method for expressing the similarity of two objects without explicitly defining their feature space[16]. The result is that they allow a much greater degree of freedom in defining the feature representation, than support vector machines.

The primary kernel we evaluated on AST's we will call the Subset Tree Kernel. When computing the similarity between two trees, this kernel conceptually counts the number of subtrees that the two trees share. A higher the number of matching subtrees means that the two trees are more similar. These comparisons are made between all of the trees in our training data in order to form a classification boundary. The number of subtrees is exponential in the size of the tree, so the algorithm's strength is that it is able to compute similarity between trees in polynomial time. The full details are described in the Collins and Duffy paper[11].

4.2 Training and Classification

Our work supposes that every programmer has a certain programming style that is reflected in the abstract syntax tree. We believe that this style takes the form of repeated substructures in the abstract syntax tree. This is why we chose the tree kernel described in the previous section.

This assumption seems intuitively reasonable. Some programmers use lots of temporary variables, and others write long expressions instead. One author might make extensive use of if-then-else-if blocks, and another might use case statements more often. Our experiments in Section 5 will validate this assumption.

As long as there are some commonly used subtrees that differentiate two different programmers, then our kernel will do a good job of finding a boundary that separates them in the training data. As unique structures between two programmers increase, the classification accuracy should increase as well.

5 Experiments and Results

We evaluated the effectiveness of our abstract syntax tree classifier over a collection of programs written by different authors. Our experiments aim to answer the following question. Given a set of methods labeled with one of two authors and an unlabeled method, can we determine the author of the unlabeled method? This section describes the set up of our experiments and their results.

5.1 Methodology

In the ideal setup for evaluating the effectiveness of our classifier, we would hand two programmers a specification for a software project, and ask them to individually implement the programs. This way, we could be assured that the differences in the abstract syntax trees were a reflection of the differences in the two authors' styles. Unfortunately, that was not possible for this experiment.

To approximate this ideal we have selected pairs of open source programs written in Java. Each pair are two programs that implement close to the same functionality and are written by different authors. For example, we look at two different implementations of a CVS client. This allows us to minimize the differences in the two programs that are not due to authorship differences.

First, we extract the Java program sources from both projects. Then, all of the source files are fed into yaxx[10]. Yaxx is a project that includes a Java parser. After parsing the source code, yaxx outputs the abstract syntax tree in XML format.

Next, we break up each XML tree at the granularity of methods. Also, we remove the names of identifiers. Removing the names of identifiers makes the trees more general. This allows our classifier to focus on structure alone. Previous technique have already done a good job of capture differences in variable naming. Last, the trees are converted from XML, into the format that our modified version of SVM-light requires. The trees from the first program are labeled as positive examples, and the trees from the second are labeled as negative examples.

The SVM package we used had a problem with trees larger than 15K bytes, so we eliminated all such trees from our datasets. These trees made up less than 5% of our data, so it is unlikely that this had a large impact on our results. However, it would be worthwhile to get the source code and try and repair this problem in future analyses. These large trees contain a lot of information that could potentially improve the classification accuracy.

Then, we used 10 fold cross validation to measure the accuracy of our classifiers. This method entails splitting up the examples at random into 10 sets. Then, 10 experiments are run. In each experiment, one of the 10 sets is used at test data, and the remaining 9 are concatenated together into the training set. The results from each of the ten runs are then be averaged.

5.2 Data Sets

The first data set we collected, was the source code from two open source Java implementations of CVS clients, jCVS[4] and Gruntspud[2]. After our preprocessing we had approximately 1800 examples.

Positive Class	Negative Class	Accuracy
gCVS	jCVS	88.47%
Beauty	Jalopy	78.59%
Jetty	Jigsaw	71.26%
JWebmail	Jwma Webmail	67.42%

Figure 2: **Classifier Experiments** *The average classification accuracy using 10 fold validation over a collection of datasets.*

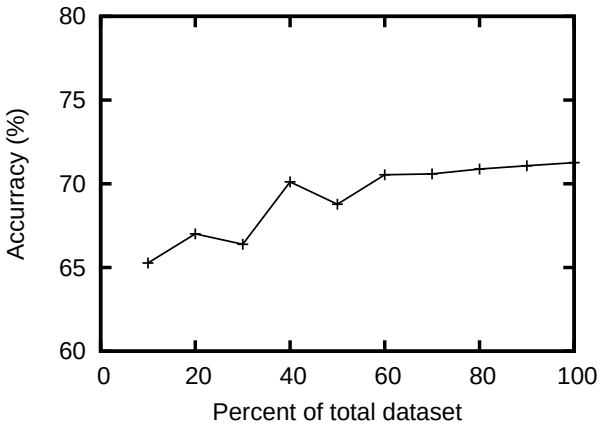

Figure 3: **Classification Accuracy vs. Size** *This graph shows how the accuracy is affected by the size of the dataset. These results are from the Jetty/Jigsaw collection, showing from the full records down to 10% of the total number of records.*

Additionally, we examined two Java web servers, Jetty[5] and Jigsaw[6]. These were larger programs and yielded about 7900 AST's. Since this was our largest dataset we also used it to test how the size of our training data effected the classification accuracy.

Also, we used the code from two Java implementations of Java pretty printers: BeautyJ[1] and Jalopy[3]. This collected contained about 2100 examples.

Finally, we tested two webmail programs: JWebmail[7] and JWMA Webmail[8]. This was our smallest sample containing only 1360 examples.

We selected all of these datasets with express purpose of showing the differences in authorship. The functionality provided by all of the pairs are very similar. So, most of the differences between them is a reflection of the differences in the way the authors' implemented them.

5.3 Results

The results of the experiments over our datasets are shown in Figure 2. These accuracies show a good deal of

promise. However, it also appears that the classification accuracy is highly sensitive to the dataset. The percentages range from 67% - 88%. This seems reasonable. The closer in style that two authors are the harder it is to find a good separation boundary between the two classes.

These results are strong enough to demonstrate that this method has promise. In many cases, we were able to achieve a high classification accuracies by examining and comparing abstract syntax trees written by two different authors.

We also wanted to gain an idea of how the number of training examples affect the classification accuracy. So, we looked at our largest dataset, the Jetty/Jigsaw programs. We measured the accuracy by taking random samples of 10% - 100% of full dataset, and using 10 fold cross validation. Figure 3 shows the results. It appears that at the sizes we experimented with, that the classification was not highly sensitive to training set size. In general, there was a trend of increasing accuracy as the number of training examples increased. However, it took ten times more training data to get a 5% increase in classification accuracy. We also did not see any evidence of over fitting at the training set sizes we examined.

6 Conclusions

Our results demonstrate that classification of source code by training a tree based kernel machine with abstract syntax trees is an effective method. Additionally, we have shown that it is effective on medium size projects. We believe that the abstract syntax tree is particularly useful for authorship classification, because the AST is a direct reflection of the code the author writes.

However, we believe that our methods are likely to apply to other classification tasks as well. An interesting avenue of future work would be to apply our same methodology to other datasets. For example, classifying programs into general program types. For example, separating programs into categories such as webserver or text editors. There is a lot to be gained from our technique, because it is able to consider the entire structure of a program, whereas other methods have to extract features and convert them into a flat feature set.

Additionally, there is a lot more to explore in the realm of authorship attribution with our methods. This work explored classification between a group of two authors. However, it is not yet clear how well this technique would extend to more than two authors.

Also, this method has some limitations. First, it requires that you know the set of possible authors. If you have no clue who the author is, then there is no clear way to construct a training set. Our methods are also vulnerable to manipulation of the source code. An advanced source code translator or obsfuscator could destroy the patterns that our classifier uses to identify authors.

Despite all of these limitations, our methods still have a great deal of potential. For the class of problems we do solve, we are able to achieve very reasonable accuracies. Additionally, it should be possible to combine the tree based techniques with many of the features discussed in Section 2. These combinations should yield an even more accurate classifier. It definitely makes sense to investigate source code classification using tree structures in the future.

References

[1] Beautyj: Customizable java source code transformation tool. `http://beautyj.berlios.de/`.

[2] Gruntsput cvs client. `http://gruntspud.sourceforge.net/`.

[3] Jalopy. `http://beautyj.berlios.de/`.

[4] jcvs. `http://www.jcvs.org/`.

[5] Jetty java http server. `http://jetty.mortbay.org/jetty/index.html`.

[6] Jigsaw - w3c's webserver. `http://www.w3.org/Jigsaw/`.

[7] Jwebmail. `http://jwebmail.sourceforge.net/`.

[8] Jwma webmail. `http://jwma.sourceforge.net/`.

[9] Abstract syntax tree, 2006. `http://en.wikipedia.org/wiki/Abstract_syntax_tree`.

[10] Yaxx: Yacc extension to xml, 2006. `http://yaxx.sourceforge.net/`.

[11] M. Collins and N. Duffy. New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 263–270, Morristown, NJ, USA, 2001. Association for Computational Linguistics.

[12] W. Elliot and R. Valenza. Was the earl of oxford the true shakespeare? *Notes and Queries*, pages 501–506.

[13] T. Joachims. Making large-scale support vector machine learning practical. *Advances in kernel methods: support vector learning*, pages 169–184, 1999.

[14] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 514–524, 1995.

[15] J. F. Mitchell. *Who Wrote Shakespeare?* Thames & Hudson, 1999.

[16] A. Moschiiti. A study on convolution kernels for shallow semantic parsing. In *42-nd Conference on Association for Computational Linguistic (ACL-2004)*, Barcelona, Spain, 2004.

[17] S. Ugurel, R. Krovetz, C. L. Giles, D. M. Pennock, E. J. Glover, and H. Zha. What's the code?: automatic classification of source code archives. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638, New York, NY, USA, 2002. ACM Press.