Authorship Analysis: Identifying The Author of a Program*

Ivan Krsul Eugene H. Spafford†

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{krsul,spaf}@cs.purdue.edu

Technical Report TR-96-052

September 4, 1996

Abstract

Authorship analysis on computer software is a difficult problem. In this paper we explore the classification of programmers' style, and try to find a set of characteristics that remain constant for a significant portion of the programs that this programmer might produce.

Our goal is to show that it is possible to identify the author of a program by examining programming style characteristics. Within a closed environment, the results of this paper support the conclusion that, for a specific set of programmers, it is possible to identify the author of any individual program. Also, based on previous work and our observations during the experiments described herein we believe that the probability of finding two programmers who share exactly those same characteristics should be very small.

1 Introduction

There are many occasions in which we would like to identify the source of some piece of software. For example, if after an attack on a system we are presented with a piece of the software responsible for the attack, we might want to identify its source. Typical examples of such software are Trojan horses, viruses, and logic bombs[1].

Other typical circumstances will require that we trace the source of a program. Proof of code re-engineering, resolution of authorship disputes and proof of authorship in courts are but a few of the more typical examples of such circumstances. Often, tracing the origins of the source requires that we identify the author of the program.

Given that software evolves over time, that programmers vary their programming habits and their choice of programming languages, and that software gets reused, it seems unlikely that, given a piece of software, we will identify the programmer who wrote it out of the millions of programmers who develop software.

However, in this paper we show that the identification process in computer software can be made reliable for a subset of the programmers and programs.

*This paper has been submitted to Computers and Security and is currently under consideration.

†Contact person for questions concerning the paper.

[1] [GS92] defines Trojan horses as programs that appear to have one function but actually perform another function; viruses as programs that modify other programs in a computer, inserting copies of themselves; and logic bombs as hidden features in programs that go off after certain conditions are met.

Our objective is to classify programmers and to find a set of characteristics that remain constant for a significant portion of the programs that they might produce.

What makes us believe that identification of authorship in computer software is possible? People work within certain repeated frameworks. They use those things that they are more comfortable with or are accustomed to. Humans are creatures of habit, and habits tend to persist. That is why, for example, we have a handwriting style that is consistent during periods of our life, although the style may vary as we grow older.

Likewise for programming, we can ask: which are the programming constructs that a programmer uses all the time? These are the habits that will be more likely entrenched, the things he consistently and constantly does and that are likely to become ingrained.

This identification process is also analogous to attempting to find characteristics in humans that can be used later to identify a specific person. Eye and hair coloring, height, weight, name and voice pattern are but a few of the characteristics that we use on a day-to-day basis to identify persons. It is, of course, possible to alter our appearance to match that of another person. Hence, more elaborate identification techniques like fingerprinting, retinal scans and DNA prints are also available, but the cost of gathering and processing this information in large quantities is prohibitively expensive. Similarly, we would like to find the set of characteristics within a program that will be helpful in the identification of a corresponding programmer, and whose computation can be automated with a reasonable cost.

However, it is in literature where we find the closest parallel. Authorship analysis in literature has been widely debated for hundreds of years, and a large body of knowledge has been developed [Dau90]. Authorship analysis on computer software, however, is different and more difficult than in literature.

Several reasons make this problem difficult. Authorship analysis in computer software does not use the same stylistic characteristics as authorship analysis in literature. Furthermore, people reuse code, programs are developed by teams of programmers, and programs can be altered by code formatters and pretty printers.

2 Background

Spafford and Weeber suggest that it might be feasible to analyze the remnants of software, typically the remains of a virus or Trojan horse, and identify its author. They theorize that this technique, they call *software forensics*, could be used to examine and analyze software in any form; be it source code for any language or executable images [WS93].

Among the measurements that Spafford and Weeber suggest are the preference for certain data structures and algorithms, the compiler used, the level of expertise of the author of a program, the choice of system calls made by the programmer, the formatting of the code, the use of pragmas or macros that might not be available on every system, the commenting style used by the programmer, the variable naming convention used, and the misspelling of words inside comments and variables. However, Spafford and Weeber provided no statistical evidence that might support their theory.

The work presented on this paper is the result of an attempt at determining the validity of their assumptions on a subset of software forensics we call *authorship analysis*. If proven correct, at least four basic areas can benefit directly from the development of authorship analysis techniques:

1. In the legal community, there is a need for methodologies that can be used to provide empirical evidence to resolve authorship disputes.
2. In the academic community, it is considered unethical to copy programming assignments. While plagiarism detection can show that two programs are equivalent, authorship analysis can be used to show that some code fragment was indeed written by the person who claims authorship of it.
3. In industry, where there are large software products that typically run for years, and contain millions of lines of code, it is a common occurrence that authorship information about programs or program fragments is nonexistent, inaccurate or misleading. Whenever a particular program module or program needs to be rewritten, the author may need to be located. It would be convenient to be able to determine the identity of the programmer who

wrote a particular piece of code from a set of several hundred programmers so he can be located to assist in the upgrade process.

4. Real-time misuse detection systems could be enhanced by inclusion of authorship information. A programmer signature constructed from the identifying characteristics of programs constitutes a pattern that can be used in the monitoring of abnormal system usage [Krs94].

2.1 Related Work

In literature, the question of Shakespeare's identity has engaged the wits and energy of a wide range of people for more than two hundred years. Such great figures as Mark Twain, Irving Wall and Sigmund Freud have debated this particular issue at length [HH92]. The issue of identifying program authorship was explored by Cook and Oman [OC89] as a means for determining instances of software theft and plagiarism. Finally, Spafford and Weeber suggested that it might be feasible to analyze the remnants of software, typically the remains of a virus or Trojan horse, and identify its author [WS93].

In actual practice, there are documented cases where people have used ad-hoc techniques, based on programming style, to make conclusions about authorship of programs[Spa89, Spa88, LS93].

2.2 Authorship Analysis in Literature

Hundreds of books and essays have been written on this topic, some as early as 1837 [Dis37]. Specially interesting is W. Elliott's attempt to resolve the authorship of Shakespeare's work with a computer by examining literary minutiae, from word frequency to punctuation and proclivity to use clauses and compounds [EV91].

For three years, the Shakespeare Clinic of Claremont Colleges used computers to see which of fifty-eight claimed authors of Shakespeare's works matched Shakespeare's writing style. Among the techniques used was a modal test that divided a text into blocks, fifty-two keywords in each block, and measured and ranked eigenvalues, or modes[2]. Other more conventional tests examined were hyphenated compound words, relative clauses per thousand, grade-level of writing, and percentage of open – and feminine – ended lines [EV91].

Although much controversy surrounds the specific results obtained by Elliott's computer analysis, it is clear from the results that works attributed to Shakespeare fit a narrow and distinctive profile. W. Elliot and R. Valenza write in [EV91] that "our conclusion from the clinic was that Shakespeare fit within a fairly narrow, distinctive profile under our best tests. If his poems were written by a committee, it was a remarkably consistent committee. If they were written by any of the claimants we tested, it was a remarkably inconsistent claimant. If they were written by the Earl of Oxford, he must, after taking the name of Shakespeare, have undergone several stylistic changes of a type and magnitude unknown in Shakespeare's accepted works."

2.3 Authorship Analysis With Programming Style

Cook and Oman define *markers* as occurrences of certain peculiar characteristics, much like the markers used to resolve authorship disputes of written works. The markers used in their work are based purely on typographic characteristics.

To collect data to support their claim, they built a Pascal source code analyzer that generated an array of Boolean measurements for the following program characteristics:

- Inline comments on the same line as source code.
- Blocked comments (two or more comments occurring together).
- Bordered comments (set off by repetitive characters).
- Keywords followed by comments.

[2] Rather than representing keyword occurrences, modes measure patterns of deviation from a writer's rates of word frequency.

- One or two space indentation occurred more frequently.
- Three or four space indentation occurred more frequently.
- Five space indentation or greater occurred more frequently.
- Lower case characters only (all source code).
- Upper case characters only (all source code).
- Case used to distinguish between keywords and identifiers.
- Underscore used in identifiers.
- BEGIN followed by a statement on the same line.
- THEN followed by a statement on the same line.
- Multiple statements per line.
- Blank lines in program body.

To test their hypothesis, Cook and Oman collected these metrics for eighteen short programs by six authors. The programs were taken from textbook example code for three tree-traversal algorithms (inorder traversal, preorder traversal and postorder traversal) and one simple sorting algorithm (bubble sort).

Cook and Oman claim that the results of the experiment were surprisingly accurate. The reported results are encouraging, but our further reflection shows that the experiment was fundamentally flawed. It failed to consider that textbook algorithms are frequently processed by code beautifiers and pretty printers, and that different problem domains will demand different programming methodologies. The implementation of the three tree-traversal algorithms involved only slight modifications and hence were likely to be similar.

It is worth noting that a Boolean measurement is inadequate for most of the measurements Cook and Oman suggest. Consider, for example, the metric dealing with the existence of blank lines in a program. In our experience, most programmers organize their programs so that some structural information is readily available by glancing at the program. Programmers spread their code, using blank lines to separate logically independent blocks. Hence, it is likely that many programs would have at least some blank lines separating their logical components.

For experimental verification, we constructed a simple program that counts the number of blank lines in almost 1,000 C files obtained from a local source code repository. The smallest file was only two lines long, the largest almost 7,000. Figure 1 shows the blank line distribution in C programs. Note how the distribution of percentages of blank lines is clustered around the ten percent mark. We can conclude, then, that a boolean measurement for this metric would not be appropriate.

We also analyzed 178 C files for indentation information. The smallest program (or file) was a few dozen lines, the largest almost 7,000. Figure 2 shows the distribution of mean and maximum indentation values. Not surprisingly, most of the programs analyzed have their indentations in the two, four and eight spaces marks. What was surprising was the large variation found. Some programs were overwhelmingly inconsistent, allowing for lines to be indented up to 16 spaces. A better indentation measurement must include a consistency value.

Cook and Oman briefly explored the use of software complexity metrics to define a relationship between programs and programmers, concluding that these are inadequate measures of stylistic factors and domain attributes[OC89]. Two other studies by Berghel and Sallach [BS84] and Evangelist [Eva84] support these findings.

2.4 Software Forensics

Spafford and Weeber suggested that a technique they called *software forensics* could be used to examine and analyze software in any form, be it source code for any language or executable images, to identify the author[WS93].

Spafford and Weeber wrote the following of software forensics:

> "It would be similar to the use of handwriting analysis by law enforcement officials to identify the authors of documents involved in crimes, or to provide confirmation of the role of a suspect.

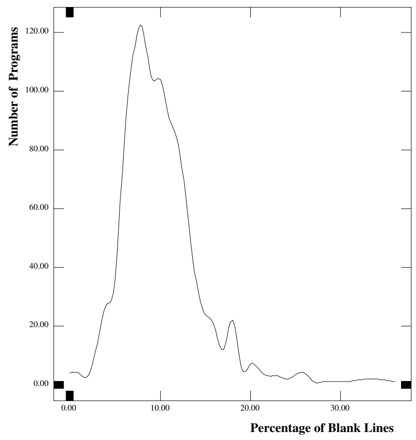

Figure 1: Blank Line Distribution in C Programs. Note that the distribution is clustered around the 10% mark. This clearly shows that a boolean measurement for blank lines would not be appropriate.

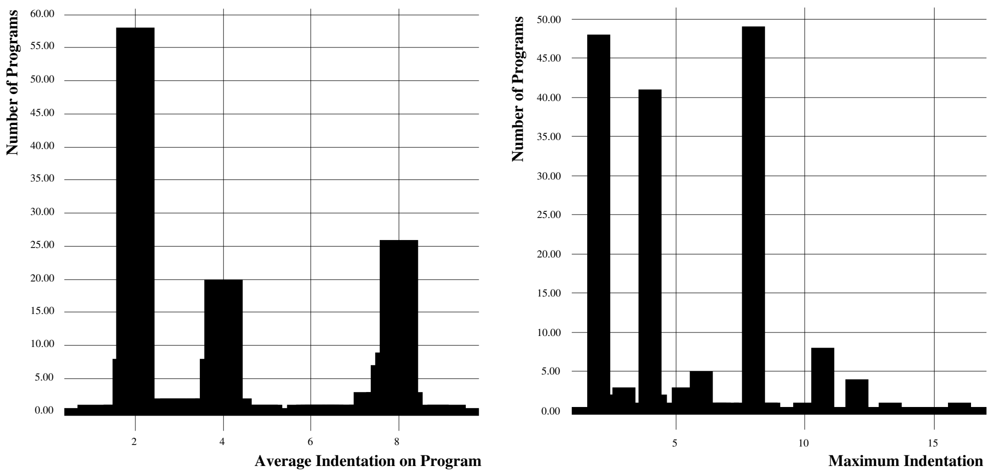

Figure 2: Average and Maximum Indentation for C Programs. Most of the programs have their indentations in the two, four and eight spaces marks. Note the large variations found. A good indentation measurement should include a consistency value.

Handwriting analysis involves identifying features of the writing in question. A feature of the writing can be anything identifiable about the writing, such as the way the i's are dotted or average height of the characters. The features useful in handwriting analysis are the writer-specific features. A feature is said to be writer-specific if it shows only small variations in the writings of an individual and large variations in the writings of different authors.

Features considered in handwriting analysis today include shape of dots, proportions of lengths, shape of loops, horizontal and vertical expansion of writing, slant, regularity, and fluency. Most features in handwriting are ordinary. However, most writing will also contain features that set it apart from the samples of other authors, features that to some degree are unusual. A sample that contains i's dotted with a single dot probably will not yield much information from that feature. However, if all of the o's in the sample have their centers filled in, that feature may identify the author."

This has a high degree of correlation with the identification of program authors using style analysis [OC89]. However, software forensics is really a superset of authorship analysis using style analysis because some of the measurements suggested by Spafford and Weeber include, but are not limited to, all the measurements made by Cook and Oman.

The list of measurements suggested by Spafford and Weeber is comprehensive, but the derivation of some of these are difficult to automate. Consider, for example, what they say about spelling and grammar measurements:

> Many programmers have difficulty writing correct prose. Misspelled variable names (e.g. TransactoingReciept) and words inside comments may be quite telling if the misspelling is consistent. Likewise, small grammatical mistakes inside comments or print statements, such as misuse or overuse of em-dashes and semicolons might provide a small, additional point of similarity between two programs.

From a small set of programs, we gathered the names of variables and found, among others, that the following were not recognized by the local spell checker: `talk_w_oracle`, `om_recv`, `oracle_mssg`, `collis_service`, `result_recv`, `result_send`, `net_colls`, `tcpchksum`, `sec_host`, `info_my_socket`, `msgc`, `remdata`, `CharToInputType`, `StateToNextState`, `NoOutputChars`, `ErrorType`, and `proto`.

We also extracted all the comments and strings from several programs of a graduate student at Purdue University and found that the following words, among others, were not recognized by our local spell checker: `algo`, `cmdAckGet`, `corrida`, `Entrando`, `Expiro`, `respuesta`, and `resultado`.

Even when we look at the original words in their context, it is practically impossible to determine when a word was misspelled. Also, some of the words the spell checker did not recognize are in Spanish. This gives us some information about the origin or educational background of the programmer, and adds to the complexity of automating these measurements.

3 Challenges

We expect the programming characteristics of programmers to change and evolve. Education is only one of many factors that have an effect on the evolution of programming styles. Not only do software engineering models impose particular naming conventions, parameter passing methods and commenting styles; they also impose a planning and development strategy. The waterfall model [GJM91], for example, encourages the design of precise specifications, utilization of program modules and extensive module testing. These have a marked impact on programming style.

The programming style of any given programmer varies also from language to language, or because of external constraints placed by managers or tools[3]. Out of the set of measurements that allow our model to identify the

[3]The use of the Motif, GL, PLOT-10 or GKS libraries generally demands that the application be structured in some fashion or may impose naming conventions.

authorship of a program, can we identify those that have been contaminated and ignore them for our analysis? A good example would be the analysis of code that has been formatted using a pretty-printer. Would it be possible for the authorship analysis system to recognize that such a formatter has been used, identify the pretty-printer and compensate by eliminating information about indentation, curly bracket placement and comment placement? Conceptually similar would be the recognition of tools used that force onto the programmer a particular programming style. For example, could the authorship analysis tool recognize the usage of Motif and compensate for variable naming conventions imposed by the tool?

Finally, among the most serious problems that must be resolved with authorship analysis is the reuse of code. All the work performed up to date on this subject assumes that a significant part of the code being analyzed was built and developed by a single individual. In commercial development projects, this is rarely the case.

3.1 An Example

As a simple example of the differences in programming style among programmers, consider the programs shown in Figure 3. These are variations on a program written by graduate students at Purdue University. The specification given for the function was "Write a function that takes three parameters of type pointer to strings. The function must concatenate the first two strings onto the third string. The use of external calls is not permitted."

We notice that the approach taken by all programmers was similar; all functions work exclusively with pointers, and all loops search for the null character at the end of each string. Three of the four programs use the `while` loop and three of the four programs use auxiliary variables. All four programs contain two loops and all loops have the same functionality.

However, these programs are far from identical. A closer scrutiny of these programs will reveal that:

1. Programmer one prefers the use of `for` loops instead of the `while` loop.
2. Only programmer three has the function header comments boxed in completely, programmer one has partial boxing, and the rest have no boxes at all.
3. Programmers one and four have chosen temporary variables with names of the form xxx1, xxx2, xxx3, etc. Only programmer two has chosen temporary variable names that reflect the use that will be given to the variable.
4. One of the programs has a significant bug: The return of the function is undefined when the two input strings are empty (i.e. it fails to *do nothing* gracefully [KP78]).
5. The program for programmer one has an incorrect comment.
6. Only one of the programmers has consistently indented his programs using three spaces. The rest used only two spaces.
7. The placement of curly braces ({}) is distinct for all programmers.

This is not an exhaustive list of differences. It is just an example that illustrates the types of features that we can examine to distinguish among programmers.

3.2 Authorship Analysis vs. Plagiarism Detection

It is important to realize that authorship analysis is markedly different from Plagiarism Detection. Plagiarism can be defined as the complete, partial or modified replication of software, with or without the permission of the original author.

Notice that according to this definition, plagiarism detection can not tell if two entirely different programs were written by the same person. Also, the replication need not maintain the programming style of the original software.

Consider, for example, the case where a student plagiarizes some original work, and proceeds to make several stylistic changes. Specifically, old comments are removed and new comments are added, indentation and placement of

```c
/*************************************************************
 * This function concatenates the first and second string into
 * the third string.
 *************************************************************/
void strcat(char *string1, char *string2, char *string3)
{
  char *ptr1, *ptr2;

  ptr2 = string3;
  /*
   * Copy first string
   */
  for(ptr1=string1;*ptr1;ptr1++) {
    *(ptr2++) = *ptr1;
  }
  /*
   * Copy first string
   */
  for(ptr1=string2;*ptr1;ptr12++) {
    *(ptr2++) = *ptr1;
  }
  /*
   * Null terminate the resulting string
   */
  *ptr2 = '\0';
}
```

```c
/*
 * concatenate s2 to s1 into s3.
 * Enough memory for s3 must already be allocated. No checks !!!!!!
 */

mysc(s1, s2, s3)
  char *s1, *s2, *s3;
{
  while (*s1)
    *s3++ = *s1++;

  while (*s2)
    *s3++ = *s2++;
}
```

```c
/* -------------------------------------------------------------- */
/* strcat(s1, s2, s3)                                             */
/* Append strings s1 and s2, and copy result into s3.             */
/* Requires that sufficient memory for s3 is already allocated.   */
/* --------------------------------------------------------------*/

void strcat (s1, s2, s3)
char *s1;
char *s2;
char *s3;
{
   char* src  = s1;
   char* dest = s3;

   while (*src) {
      *(dest++) = *(src++);
      };

   src = s2;
   while (*src) {
      *(dest++) = *(src++);
      };

   *dest = '\0';
};
```

```c
/*
    str_cat(char *str1, char *str2, char *str3)
    concatenates string str1 & str2 into str3
    */
void str_cat(char *str1, char *str2, char *str3)
{
  char *aux1,* aux2;

  aux1 = str3;
  aux2 = str1;

  while (*aux2 != 0)             /* CIoopy str1 -> str3 */
    {
      *aux1 = *aux2;
      aux1++;
      aux2++;
    }

  aux2 = str2;
  while (*aux2 != 0)             /* Append str23 -> str3 */
    {
      *aux1 = *aux2;
      aux1++;
      aux2++;
    }
                        /* End str3 with a null character */
  *aux1 = 0;
}
```

Figure 3: Style Variations on a Program. Four functions written by four graduate students at Purdue from the same specification. Note that one can find identifying characteristics on each.

brackets are changed to match her programming style, variables are renamed to something she feels more comfortable with, the order of functions is altered and `for` loops are changed to `while` loops.

While plagiarism detection needs to detect the similarity between these two programs, authorship analysis does not. For the purpose of authorship identification, these two programs have distinct authors.

Many people have devoted time and resources to the development of plagiarism detection [Ott77, Gri81, Jan88, Wha86], and a comprehensive analysis of their work is beyond the scope of this paper. We can, however give a simple example that will illustrate how measurements traditionally used for plagiarism detection are ill suited to authorship analysis.

Consider the two functions shown in Figure 4. Both are structurally and logically equivalent. However, the second function has undergone several stylistic changes that mask the identity of the original programmer. Plagiarism detection systems might consider them identical. Traditional software engineering metrics, used commonly for plagiarism detection, will yield similar values. But authorship analysis should not consider them identical. If both authors happen to write these functions independently, and this is a common occurrence, our system should identify them as unique.

4 Our Experiment

The term *Software Metric* was defined by Conte, Dunsmore and Shen in [SCS86] as: "Software metrics are used to characterize the essential features for software quantitatively, so that classification, comparison, and mathematical analysis can be applied." What we are trying to measure, for establishing the authorship of a program, is precisely some of these features. Hence, the term software metric, or simply metric, is appropriate to describe these special characteristics.

Although theoretically possible, it would be impractical to compare style metrics across different development platforms. Among similar languages such as C, Modula and Pascal, the same metrics might be used successfully with similar results. This might not be true if C is compared with Prolog or LISP. These programs belong to three different programming paradigms (Structured Programming, Logic Programming and Functional Programming) and there are large differences among them. Many of the metrics we could use for identifying authorship in one of these programming languages will be of no use in the others.

Hence, in this paper we will limit ourselves to the stylistic concerns of C source code. Programmers are comfortable using it and the language is commonly used in the academic community and in industry.

4.1 Sources for the Collection of Metrics

We considered metrics for authorship detection from a wide variety of likely sources:

- Oman and Cook [OC91] collected a list of 236 style rules that can be used as a base for deriving metrics of programming style.
- Conte, Dunsmore and Shen [SCS86] give a comprehensive list of software complexity metrics.
- Kernighan and Plauger [KP78] give over seventy programming rules that should be part of good programming practice.
- Van Tassel [Tas78] devotes a chapter to programming style for improving the readability of programs.
- Jay Ranade and Alan Nash [RN93] give more than three hundred pages of style rules specifically for the C programming language.
- Henry Ledgard gives a comprehensive list of C programming proverbs that contribute to programming excellence [Led87].

```
/*********************************************
 * Subroutine for checking a string for TABS
 * Parameters: s1:String to examine
 * Return: Boolean indicating the existence
 *         of a tab character
 ********************************************/
int strchk(char *s1)
{
  char *ptr1;

  ptr1 = s1;
  while(*ptr1 != 0)
    if(*ptr1++ == '\t')
      return(1);
  return(0);
}
```

```
#define TRUE 1
#define FALSE 0
/* Checks the existence of \t in string */
int check_for_tab_in_string(string)
char *string;
{
  char *character_pointer;

  /* Loop until found or we reach EOLN */
  for(character_pointer = string;*character_pointer != NULL;)
  {
    /* check to see if we found TAB */
    if(*(character_pointer++) == 9)
    {
      /* Success!! */
      return(TRUE);
    }
  }
  /* No tab */
  return(FALSE);
}
```

Two program modules that might be considered similar in plagiarism detection systems. They would not be considered similar in authorship analysis.

Figure 4: Plagiarism detection vs. authorship analysis. Both functions are functionally equivalent, and one may be a plagiarized version of the other. However, they have distinctive styles. Authorship analysis should not consider their authors to be the same.

Many other sources have influenced our choice of metrics [LC90, BB89, OC90, Coo87] but do not contain a specific set of rules, metrics or proverbs.

All these sources provide material from which we may derive useful metrics. Because of the large number of rules and metrics available, we decided to divide our metrics into three categories:

- We would like to examine those metrics that deal specifically with the layout of the program. In this category we will include such metrics as the ones that measure indentation, placement of comments, placement of brackets, etc. We will call these metrics *Programming Layout* metrics.
  All these metrics are fragile because the information required can be easily changed using code formatters and pretty printers. Also, the choice of editor can significantly change some of the metrics of this type. The use of the Emacs editor with standard macros, for example, encourages consistent indentation and curly bracket placement. Furthermore, many programmers learn their first programming language in university courses that impose a rigid and specific set of style rules regarding indentations, placement of comments and the like.
- Also useful are the metrics that deal with characteristics that are difficult to change automatically by pretty printers and code formatters, and are also related to the layout of the code. In this category we include those metrics that measure the statistical distribution of variable lengths, comment lengths, etc. We will call these metrics *Programming Style* metrics.
- Finally, we would like to examine metrics that we hypothesize are dependent on the programming experience and ability of the programmer. In this category we will find such metrics as the statistical distribution of lines of code per function, usage of data structures, etc. We will call these metrics *Programming Structure* metrics.

4.2 Metrics Considered

From all the sources mentioned in Section 4.1, we extracted a series of potentially useful software metrics. Even though we describe these metrics as indivisible measurements, in practice we might calculate several values for them. For example, for metric STY1a we might calculate a median and a standard deviation.

Also, unless explicitly stated, all the metrics consider only the text inside function bodies. We do not examine include files or type declarations because there is no consistent method for differentiating between those declarations that are imported from external modules, and those that are native to the programmer.

4.2.1 Programming Layout Metrics

- Metric STY1: A vector of metrics indicating indentation style [RN93, pages 68–69]:
  - Metric STY1a: Indentation of C statements within surrounding blocks.
  - Metric STY1b: Percentage of open curly brackets ({) that are alone in a line.
  - Metric STY1c: Percentage of open curly brackets ({) that are the first character in a line.
  - Metric STY1d: Percentage of open curly brackets ({) that are the last character in a line.
  - Metric STY1e: Percentage of close curly brackets (}) that are alone in a line.
  - Metric STY1f: Percentage of close curly brackets (}) that are the first character on a line.
  - Metric STY1g: Percentage of close curly brackets (}) that are the last character in a line.
  - Metric STY1h: Indentation of open curly brackets ({).
  - Metric STY1i: Indentation of close curly brackets (}).

  Figure 5 illustrates some of the different indentation styles we can differentiate using this vector of metrics.
- Metric STY2: Indentation of statements starting with the `else` keyword.
- Metric STY3: In variable declarations, are variable names indented to a fixed column? Figure 6 shows an example of two program fragments we could distinguish using this metric.

```
if(a==b) {                 if(a==b)
  b = 1;                     {b =1;
}                            }


if(a==b) { b = 1;          if(a==b) { b = 1; }
}


if(a==b) {                 if(a==b)
  b = 1;                   {
  }                          b = 1;
                           }
```

Figure 5: Indentation Styles and Placement of Brackets. The vector of metrics for STY1 should help to differentiate these program fragments.

```
main()                          main()
{                               {
  int i;                          int           i;
  char *cptr;                     char          *cptr;
  static float number;            static float  number;
  short int j;                    short int     j;
  .                               .
  .                               .
  .                               .
}                               }
```

Un-indented variable declarations   Indented variable declarations

Figure 6: Variables Indented to a Fixed Column. Metric STY3 would help distinguish among this two program fragments.

- Metric STY4: What is the separator between the function names and the parameter lists in function declarations? Possible values are spaces, carriage returns or none.
- Metric STY5: What is the separator between the function return type and the function name in function declarations? Possible values are spaces or carriage returns.
- Metric STY6: A vector of metrics that will help identify the commenting style used by the programmer. The vector will be composed of:
  - Metric STY6a: Use of borders to highlight comments.
  - Metric STY6b: Percentage of lines of code with inline comments.
  - Metric STY6c: Ratio of lines of block style comments to lines of code.
- Metric STY7: Ratio of white lines to lines of code [RN93, pages 70–71].

4.2.2 Programming Style Metrics

- Metric PRO1: Mean program line length (characters per line) [BM85].
- Metric PRO2: A vector of metrics that will consider name lengths.
  - Metric PRO2a: Mean local variable name length.
  - Metric PRO2b: Mean global variable name length.
  - Metric PRO2c: Mean function name length.
  - Metric PRO2d: Mean function parameter length.
- Metric PRO3: A vector of metrics that will tell us about the naming conventions chosen by the programmer. This vector will consist of:
  - Metric PRO3a: Some names use the underscore character.
  - Metric PRO3b: Use of temporary variables[4] that are named XXX1, XXX2, etc. [KP78], or "tmp," "temp," "tmpXXX" or "tempXXX" [RN93].
  - Metric PRO3c: Percentage of variable names that start with an uppercase letter.
  - Metric PRO3d: Percentage of function names that start with an uppercase letter.
- Metric PRO4: Global variable count to mean local variable count ratio. This metric could potentially tell us something about the programmer's propensity to use global variables.
- Metric PRO5: Global variable count to lines of code ratio. This variation of the previous metric might give us a better metric for measuring the frequency of usage of global variables.
- Metric PRO6: Use of conditional compilation.
- Metric PRO7: Preference of either `while`, `for` or `do` loops.
- Metric PRO8: Does the programmer use comments that are nearly an echo of the code [KP78, page 143] [RN93, page 82]?
- Metric PRO9: Type of function parameter declaration. Does the user use the standard format or the ANSI C format?

4.2.3 Programming Structure Metrics

- Metric PSM1: Percentage of `int` function definitions.
- Metric PSM2: Percentage of `void` function definitions.
- Metric PSM3: Program uses a debugging symbol or keyword[5]. We would specifically be looking at identifiers

[4] It can be argued that all local variables are temporary and no global variable is temporary. However, in this paper we will follow the convention that a variable is temporary if it there is no direct association between its name and its (assumed) use.

[5] Debugging is difficult. Many nonstandard techniques have been developed [RN93], and we cannot hope to identify all forms of debugging symbols. However, there are some techniques that are widely used and we will concentrate on these.

or constants containing the words “debug” or “dbg” [RN93, pages38–53]. Figure 7 shows three common debugging styles in C.

```c
#define DBG(x) printf(x)

main() {
  DBG("main\n");
  Parse_Input();
  DBG("Finished\n");
}
```

```c
#define DEBUG 1

main() {
#ifdef DEBUG
  printf("main\n");
#endif
  Parse_Input();
#ifdef DEBUG
  DEBUG("Finished\n");
#endif
}
```

```c
main(argc, argv)
int argc;
char *argv[];
{
  if(  strcmp(argv[1] == 0 )
    debug = 1;
  else
    debug = 0;
  if(debug) printf("main\n");
  Parse_Input();
  if(debug) printf("Finished\n");
}
```

Figure 7: Examples of Debugging Styles. There are many frequently used debugging techniques, and programmers frequently prefer one over the other. The choice of debugging technique can be a useful metric.

- Metric PSM4: The assert macro is used.
- Metric PSM5: Lines of code per function [KP78, BM85].
- Metric PSM6: Variable count to lines of code ratio. This metric could identify those programmers who tend to avoid reusing variables, creating new variables for each loop control variable, etc.
- Metric PSM7: Percentage of global variables that are declared static.
- Metric PSM8: The ratio of decision count to lines of code. To simplify the computation of this metric, we have chosen to modify the definition of decision count as given in [SCS86]. We do not count each logical operator inside a test as a separate decision. Rather, each instance of the `if`, `for`, `while`, `do`, `case` statements and the `?` operator increases our decision count by one.
- Metric PSM9: Is the `goto` keyword used? Software designers and programmers still rely on these [BM85].
- Metric PSM10: Simple software complexity metrics offer little information that might be application independent [OC89]. The metrics that we could consider are: cyclomatic complexity number, program volume, complexity of data structures used, mean live variables per statement, and mean variable span [SCS86].
- Metric PSM11: Error detection after system calls that rarely fail. Some programmers tend to ignore the error return values of system calls that are considered reliable [GS92, page 164]. Thus, a metric can be obtained out of the percentage of reliable system calls whose error codes are ignored by the programmer. Also, some programmers tend to overlook the error codes returned by system calls that should never have them ignored (e.g. `malloc`). We can define this metric as a vector of the following items:
  - Metric PSM11a: Are error results from memory related system calls ignored? Specifically, we would be looking at malloc(), calloc(), realloc(), memalign(), valloc(), alloca() and free().
  - Metric PSM11b: Are error results from I/O routines ignored? Specifically, we would be looking at open(), close(), dup(), lseek(), read(), write(), fopen(), fclose(), fwrite(), fread(), fseek(), getc(), putc(), gets(), puts(), printf() and scanf().
  - Metric PSM11c: Are error results from other system calls ignored? We would be looking at chdir(), mkdir(), unlink(), socket(), etc.
- Metric PSM12: Does the programmer rely on the internal representation of data objects? This metric would check for programmers relying on the size and byte order of integers, the size of floats, etc.
- Metric PSM13: Do functions *do nothing* successfully? Kernighan and Plauger in [KP78, pages 111–114] and Jay Ranade and Alan Nash in [RN93, page 32] emphasize the need to make sure that there are no unexpected side effects in functions when these must do nothing. In this context, functions that do nothing successfully

are functions that correctly test for boundary conditions on their input parameters. We must note that it is an undecidable problem to determine the correctness of an arbitrary function[HU79].

- Metric PSM14: Do comments and code agree? Kernighan and Plauger write in [KP78] that "a comment is of zero (or negative) value if it is wrong". Ranade and Nash [RN93, page 89] devote a rule to the truth of every comment. Even if the comments were initially accurate, it is possible that during the maintenance cycle of a program they became inaccurate. Because we cannot determine the stage of development where the incorrect comment was introduced, we will consider all incorrect comments[6] in this metric.
- Metric PSM15: More than any other type of software metric, those that deal with the development phase of a project would help to identify the authorship of a program. Consider, for example, whether comments are placed before, during or after the development of a program, the choice of editor, the choice of compiler, the usage of revision control systems, the usage of development tools, etc. Unfortunately, this information is not readily available. Test programs, intermediate versions, debugging code and the alike are discarded after the final version of the program is finished.
- Metric PSM16: Quality of software. We could use software metrics that deal with the quality of software to assess the level of experience of the programmer. Typically, software quality metrics are related to software development standards and try to measure the reliability and robustness of software.
  These metrics will not be useful. In the worst case, we would be measuring the care that the programmer has taken to develop a piece of code as well as the level of expertise of the programmer. Furthermore, it is possible for an experienced programmer to get low software quality scores and for a beginner to get high scores (if he followed a textbook algorithm for his program).

A software analyzer built for the lcc C compiler front-end developed at Princeton [Han91] was used to generate most of the raw data, including all the programming structure metrics. Once the calculation of these metrics had been performed, a series of Perl programs were used to collect the metrics that depended on the information that was discarded by the C preprocessor. Indentation, commenting style and line lengths are examples of the measurements collected by these scripts.

4.3 Experiment Structure

The experimental data for this paper was gathered in three distinct stages:

- A preliminary stage helped us determine the proper methods for calculating the metrics, eliminated those metrics that were clearly inappropriate for our purposes, and coexisted with the tool development phase[Krs94].
- A pilot experiment was performed with a small number of programmers, each of whom wrote three short and simple programs[Krs94, KS95]. To determine the effect of problem domains on our analysis, the programs were oriented to the three areas where we thought we could have the greater variations in style: computationally intensive programs, I/O intensive programs and data structure intensive programs[7]
- Once the preliminary experiment showed that the desired set of metrics could be analyzed, we designed and executed a larger, more formal experiment in which to test our prototype.

For the final experiment, a series of programs were collected from a total of 29 students, staff and faculty members at Purdue University. The distribution for the programs are shown in Table 1.

We included programs from a wide variety of programming styles and for different problem domains. Roughly one third of the student programs were programming assignments from a graduate level networking course, one third of the programs were programming assignments from a graduate level compilers course and one third of the programs were from miscellaneous graduate level courses, including databases, numerical analysis and operating systems. Of the programs submitted by the faculty members, half are oriented towards numerical analysis and half oriented towards compiler construction and software engineering.

---

[6]Deciding that a comment is wrong can only be done manually by careful examination of the source code. Because it involves the semantic analysis of English sentences, it is unlikely that this process will be automated soon.

[7]A detailed description of the programs can be found in [Krs94, KS95]

Table 1: Distribution of Programs

Group Identification	Programs
Students 1(Projects for the Fall 1993 term)	57
Students 2 (Programs developed for other terms)	6
Pilot 1 (Programs developed by students for the pilot experiment)	18
Pilot 2 (Programs developed by experienced programmers for the pilot experiment)	6
Faculty (Miscellaneous programs by faculty members)	7
TOTAL	88

5 Analysis of Results

Several methods can be used to analyze the metrics gathered. The method used during the early phases of the experiments is a statistical analysis method called discriminant analysis. This method, described in [SAS, JW88] is a multivariate technique concerned with separating observations and with allocating new observations into previously defined groups.

On subsequent phases of the experiments we used the neural networks and likelihood classifiers that the LNKnet software, developed at the MIT Lincoln Laboratory, provides. The software is described in [KL95].

5.1 Discriminant Analysis with SAS

This technique works best with those metrics that show little variation between programs (for a specific programmer) and large variations among programmers. Unfortunately, analysis of the metrics collected shows that these two criteria are not necessarily correlated. Initially, we calculated the standard error by programmer for every metric, and eliminated those that showed large variations because they identify those style characteristics where the programmer is inconsistent. Surprisingly, most of the metrics that showed large variations among programmers were eliminated as well. The performance of our statistical analysis with the remaining metrics was discouraging, with only twenty percent of the programs being classified correctly.

The step discrimination tool provided by the SAS program [SAS] should theoretically be capable of eliminating metrics that are not useful from the statistical base. Unfortunately, this tool was not helpful because it failed to eliminate any of the metrics from our set.

To resolve this issue, we built a tool that helped us visualize the metrics collected. For each continuous metric (i.e. real valued metric) the tool displayed two graphs that showed the variation of the metric within programs for each programmer and the distribution of values for each metric for all programmers.

For each discrete metric (i.e. boolean metrics and set metrics), the tool produced a graph that showed the consistency of each programmer for each metric. In these figures, vertical lines represent a programmer jumping from one value to the next in two consecutive programs. Hence, a good discrete metric is one that shows variations in values and no jumps.

Table 2: Classification by Programmer

% of programs correctly classified	Number of programmers
100%	17
77%	3
75%	1
71%	1
50%	1
33%	2
25%	1
20%	1
0%	2

With this analysis, we chose a small subset of our metrics for the final statistical analysis[8].

The success rate for this first classification attempt was 73%. Of all the programs analyzed, we were able to identify the authors for 73% of them. Individual percentages of correctly classified programs are shown in table 2.

Initially, we were surprised to see that seasoned programmers, a faculty member, and graduate students of Computer Science were all included in the group of programmers with identification rates of 100

1. The programs for the faculty member (three programs averaging 300 lines of code each) were developed over several years and address different problem domains.
2. Three of the six programmers who helped with the development of the programs for the pilot study were correctly classified 100% of the time. The programs for each of these programmers addressed different problem domains.
3. The programmers who were correctly classified have different backgrounds.

For the programmers who were classified less than 50% of the time, we looked at their code to find out why we failed to classify them (two programmers were never classified correctly). We were surprised to find that one had varied his programming style considerably from program to program in a period of only two months[9].

Other misclassified programmers showed a consistent programming style. This fact is a clear indication that either the metrics chosen for our experiment were not comprehensive enough to distinguish among them, or the statistical tool used is not comprehensive enough. The programs for these programmers are far from identical, as subsequent inspection of their code revealed. For one of the programmers who was never classified correctly, for example, we could find several characteristics that remained consistent throughout: comments at the end of code blocks (see Figure 8), heavy reliance on C macros, and use of RCS headers are but a few of these characteristics.

Our experiment also helped us predict the performance of the metrics when a program not included in the original database is considered. For each program, we removed it from the database and later told SAS to classify it. As expected, the results average 73%. However, this stage of our experiment shed some light as to the consistency of the misclassification. Mainly, some programmers are misclassified consistently. Programmer 18 was misclassified consistently as programmer 12, programmer 19 as programmer 17, programmer 11 as programmer 18, and programmer 26 as programmer 9. We can conclude that even though the metrics are not good enough to classify these programmers correctly, the misclassification is not random. A more refined set of metrics could help distinguish among these programmers.

[8] Specifically, metrics PRO1M, mean for PRO2a, mean for PRO2b, mean for PRO2c, PRO3d, PRO5, PSM1, PSM6, mean for STY1a, STY1b, STY1c, STY1d, STY1e, STY1f, mean for STY1i, mean for STY2, STY6b, STY6c, STY7, PRO8, PSM3, STY4 and STY5.

[9] It is likely, although we do not have definite proof to confirm our suspicion, that the student who submitted these programs relied quite heavily on class projects from previous semesters.

```
main() {
.
.
.
  while ( !eof ) {
    ch = getch();
    if( ch == 0 ) {
         .
         .
         .
    } else {
         .
         .
         .
    } /* if( ch == 0 ) */
  } /* while ( !eof ) */
} /* main() */
```

Figure 8: Example of a consistent coding style we missed. One of the programmers we failed to classify always added comments at the end of code blocks to indicate what block was ending. A metric could be added to our list to look for this kind of behavior.

The statistical analysis tools used provide little support for ranking the performance of individual metrics. The removal of any one metric from the analysis can have negative or positive effects, independent of the quality of the metric.

5.2 Statistical Analysis and Neural Network with LNKnet

The LNKnet software developed at the MIT Lincoln Laboratory simplifies the application of some of the most important statistical, neural networks, and machine learning pattern classifiers. Table 3, extracted from the LNKnet User's Guide [KL95], shows the classification mechanisms available.

The Neural Networks and Classification Algorithms in this software are sophisticated enough to eliminate from consideration those metrics that contribute little to the classification. Hence, we included in this experiment the following 49 metrics:

- Real Metrics.
  - PRO1M: Average program line length.
  - PRO1S: Standard deviation for metric PRO1M.
  - PRO2aM: Average local variable name length.
  - PRO2aS: Standard deviation for metric.
  - PRO2bM: Average global variable name length.
  - PRO2bS: Standard deviation for metric PRO2bM.
  - PRO2cM: Average function name length.
  - PRO2cS: Standard deviation for metric PRO2cM.
  - PRO2dM: Average function parameter length.
  - PRO2dS: Standard deviation for metric PRO2dM.

Table 3: LNKnet Algorithms.

	Supervised Training	Combined Unsupervised-Supervised Training	Unsupervised Training (Clustering)
Neural Network Algorithms	Back-Propagation (BP) Adaptive Stepsize BP Cross-Entropy BP Hyperspace Classifier Committee	Radial Basis Function (RBF) Incremental RBF (IRBF) Top-2-Diff IRBF Nearest-Cluster Classifier	Leader Clustering
Conventional Pattern Classification Algorithms	Gaussian Linear Discriminant Quadratic Gaussian K-Nearest Neighbor (KNN) Condensed KNN Binary Tree Parzen Window Histogram	Gaussian Mixture (GMIX) Classifier Diagonal/Full Covariance GMIX Diagonal/Full Covariance GMIX Tied/Per-Class Centers GMIX	K-Means Clustering E&M Clustering
Feature Selection Algorithms	Canonical Linear Discriminant Forward and Backward Search using N-fold Cross Validation		Principal Components Analysis

- PRO3c: Percentage of variable names that start with an uppercase letter.
- PRO3d: Percentage of function names that start with an uppercase letter.
- PRO4: Global variable count to average local variable count ratio.
- PRO5: Global variable count to lines of code ratio.
- PSM1: Percentage of `int` function definitions.
- PSM2: Percentage of `void` function definitions.
- PSM5: Lines of code per function.
- PSM6: Variable count to lines of code ratio.
- PSM7: Percentage of global variables that are declared static.
- PSM8: The ratio decision count over lines of code.
- STY1aM: Indentation of C statements within surrounding blocks.
- STY1aS: Standard deviation for metric STY1aM.
- STY1b: Percentage of open curly brackets ({) that are alone in a line.
- STY1c: Percentage of open curly brackets ({) that are the first character in a line.
- STY1d: Percentage of open curly brackets ({) that are the last character in a line.
- STY1e: Percentage of closed curly brackets (}) that are alone in a line.
- STY1f: Percentage of closed curly brackets (}) that are the first character on a line.
- STY1g: Percentage of closed curly brackets (}) that are the last character in a line.
- STY1hM: Indentation of open curly brackets ({).
- STY1hS: Standard deviation for metric STY1hM.
- STY1iM: Indentation of close curly brackets ({).
- STY1iS: Standard deviation for metric STY1iM.
- STY2M: Indentation of statements starting with the else keyword.

- STY2S: Standard deviation for metric STY2M.
- STY6b: Percentage of lines of code with inline comments.
- STY6c: Ratio of lines of block style comments over lines of code.
- STY7: Ratio of white lines over lines of code.

- Discrete Variables.
  - PRO3a: Names use the underscore character.
  - PRO3b: Use of temporary variables.
  - PRO6: Use of conditional compilation.
  - PRO8: Does the programmer use comments that are nearly en echo of the code?
  - PSM3: Program uses a debugging symbol or keyword.
  - PSM4: The assert macro is used.
  - PSM9: Is the goto keyword used?
  - STY3: In variable declarations, are variable names indented to a fixed column?
  - STY6a: Use of borders to highlight comments.
  - PRO7: Preference of either while, for or do loops.
  - STY4: What is the separator between the function name and the parameter list in function declarations?
  - STY5: What is the separator between the function return type and the function name in function declarations?

The neural networks and classification algorithms in LNKnet require large numbers of data that must be divided into a training data set, a evaluation data set, and a test data set. Because we only have 88 points (programs) in our data set, we were forced to use N-fold cross-validation. The LNKnet User's Guide describes this as "the idea of cross-validation is to split the data into N equal-sized folds and test each fold against a classifier trained on the data in the other folds. The cross-validation error rate is obtained from summing the errors from the tests."

LNKnet provides three data normalization methods that either scale or rotate the input space[KL95]. Simple normalization rescales each input feature independently to have a mean of 0 and a variance of 1[10]. Principal Components Analysis (PCA) rotates the input space to make the direction of greatest variance the first dimension[11]. Linear Discriminant Analysis (LDA) assumes that classes and class means can be modeled using Gaussian distributions, and rotates the input space to make the first dimension the direction along which the classes can be most easily discriminated[12].

A feature search is an algorithm that attempts to find the best input features in a database. From the LNKnet User's Guide:

> "To run a feature search, the program generates a series of feature lists and for each list performs a cross-validation test on a classifier....
>
> There are three directions for the selection of features for inclusion in the features list tests by the classifier. The search can go forward, backward, or forward and back. In a forward search, each feature is tried singly and the feature which gets the best classification rate is selected as the first feature. The remaining features are tested in combination with the first feature and the best of them is added as the second feature. Features are added this way until there are none left to add.
>
> In a backward search, the program starts with all of the features selected and tries leaving each one out. The feature which the classifier did best without is selected as the last feature. The program goes on taking features away until none are left. The idea of a backward search is that there may be some set of features which do well when they are together but do poorly individually. This set of features would not be found by a forward search.

[10] This compensates for the differences in the means and variances of the input dimensions.
[11] The remaining orthogonal dimensions correspond to directions of decreasing variance in the original input space.
[12] The remaining dimensions are ordered by decreasing ability to be used to discriminate the classes.

> A forward and backward search combines the two search methods above. The program starts searching forward with no features selected. When it has added two features, it searches for one to take away. It continues then, adding two and taking away one, until it has added all of the available features. This forward and backward search can find some interdependencies in the in input features which are not found using the other two searches."

Although we tried all the classifications provided by LNKnet, not all of the proved to be suitable for our purposes. In the following subsections we list those classification algorithms that yielded error rates below the 30% mark.

5.2.1 Multi-Layer Perceptron

We were able to train a Multi-Layer Perceptron (MLP) neural network to classify the programmer in our test data with error rates as low as 2% This was achieved using LDA normalization, 4-fold cross-validation, 100 nodes in the hidden layer, and cross-entropy as a cost function[13].

The most significant metrics for this run were selected using a forward and backwards feature search using MLP with 4-fold cross-validation as the search algorithm. Figure 9 shows how the classification errors drop sharply with the addition of few metrics.

The feature search for this classification mechanism identified the following metrics as the most telling: PRO1M (Average program line length), PRO2bS (Standard deviation for global variable name length), PRO2cM (Average function name length), PRO1S (Standard deviation for program line length), PRO2aM (Average local variable name length), PRO2aS (Standard deviation for local variable name length), PRO2cS (Standard deviation for function name length), PRO2dM (Average function parameter length), PRO3d (Percentage of function names that start with an uppercase letter), PRO3a (Names use the underscore character), STY1g (Percentage of closed curly brackets that are the last character in a line), PRO3b (Use of temporary variables), STY1iS (Standard deviation for indentation of close curly brackets), PSM2 (Percentage of void function definitions), and STY5 (What is the separator between the function return type and the function name in function declarations?).

Note, however, that the same classification algorithm performs much more poorly when the data set is not normalized using LDA. Figure 10 shows the error rates obtained during a forwards and backwards feature search using the MLP classifier on data normalized using PCA normalization. The best error rate was 26%, and the algorithm needed 40 metrics to achieve this result. The metrics needed were PRO2cM, PRO1M, PRO3b, PRO2dM, PRO2bM, PRO2bS, PRO1S, STY2S, PSM8, PRO3d, PSM4, STY1e, STY2M, PRO6, PSM6, STY1b, STY1d, PSM2, PRO2aS, STY1aM, STY6b, PSM5, STY1aS, STY3, PRO8, PSM1, PRO2dS, PRO2aM, PSM9, PSM3, PRO4, STY1f, PRO7, STY1hS, STY6c, STY6a, STY1iM, STY5, STY7, STY1hM.

5.2.2 Gaussian Classifier

Gaussian classifiers are a form of likelihood classifiers. From the LNKnet manual, "these estimate a scaled probability density function or likelihood for each class, $p*(X \mid A)P(A)$ where A again represents a class label,X is the input feature vector for a pattern, $p(X \mid A)$ is the likelihood of the input data for class A and $P(A)$ is the prior probability for class A. For a given test pattern, the class which has the highest likelihood times the class prior probability is selected as the class of a pattern." [KL95].

Gaussian classifiers are the most common and simple classifiers. A Gaussian classifier models each class with a Gaussian distribution centered around on the mean of that class. This classifier also performs exceedingly well with our test data, with an overall accuracy of 100%. In fact, a feature selection with the Gaussian classifier with 4-fold cross-validation as a search algorithm, and using an LDA normalization, shows that the error rates drop sharply using only a few of the metrics on our data set. Figure 11 shows how the classification errors drop sharply with the addition of few metrics.

[13] A description of these parameters can be found in the LNKnet User's Guide [KL95].

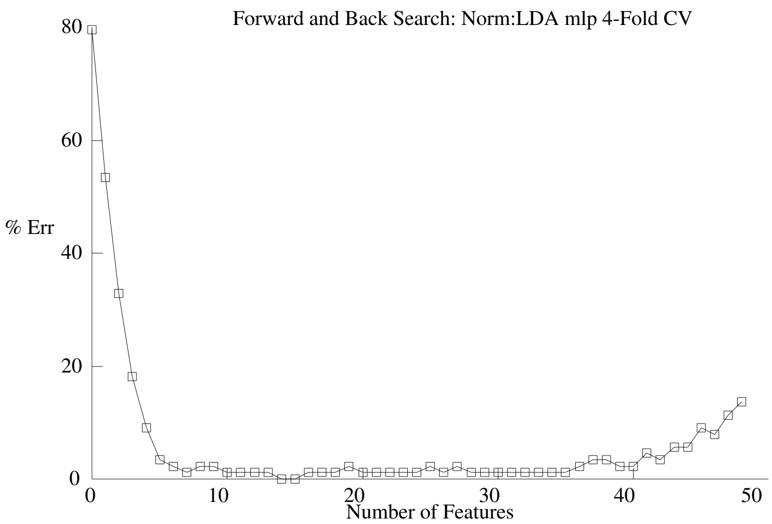

Figure 9: Feature search using MLP classifier and LDA normalization. This graph shows how the classification error drops sharply with the addition of very few metrics.

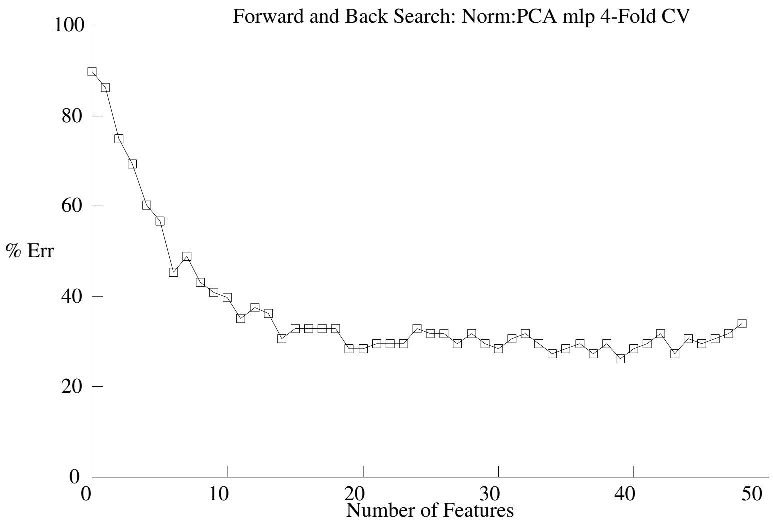

Figure 10: Feature search using MLP classifier with PCA normalization. The graph shows how this classification algorithm does not yield the sharp drop that LDA normalization does.

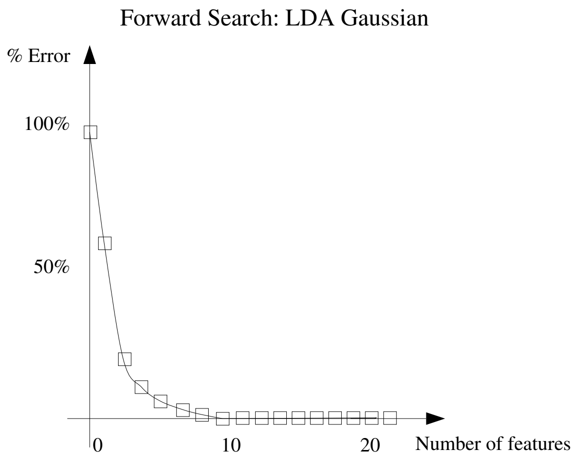

Figure 11: Feature search using Gaussian classifier with LDA normalization. This graph shows how the classification error drops sharply with the addition of very few metrics.

The feature search for this classification mechanism identified the following 6 metrics as the more significant: PRO1M (Average program line length), PRO2aM (Average local variable name length), PRO2cM (Average function name length), PRO1S (Standard deviation for program line length), PRO2bS (Standard deviation for global variable name length), and STY1c (Percentage of open curly brackets that are the first character in a line).

Note, however, that the same classification algorithm performs much more poorly when the data set is not normalized using LDA. Figure 12 shows the error rates obtained during a forwards and backwards feature search using the Gaussian classifier on data normalized using PCA normalization. The best error rate was 25%, and the algorithm needed 29 metrics to achieve this result. The metrics needed were PRO1S, PRO2bS, PRO3b, PRO2cM, PRO1M, PSM7, PRO2aM, PRO2bM, PRO3c, PSM8, PRO7, PRO3d, STY2M, STY1f, PSM6, PRO5, STY6b, PRO3a, PRO4, PSM5, STY1aM, STY1g, STY1iS, PSM9, PRO2dS, STY1hM, and PRO8.

5.2.3 Learning Vector Quantizer

The Learning Vector Quantizer (LVQ) is a Nearest Neighbor classifier. These work on the principle that a pattern is probably of the same class as those patterns nearest to it. The simplest algorithms is to store all the training patterns and to find distances to them all for each testing pattern. The LVQ training algorithm can improve the performance of a nearest cluster classifier by moving cluster centers [KL95].

The best error rates were obtained by using an LDA normalization with 4-fold cross-validation. The error rates are around the 22% mark.

5.2.4 Histogram

A Histogram classifier is a likelihood classifier. It estimates the likelihood of each class by creating a set of histograms for each input feature. Each input feature dimension is divided into a number of bins. The likelihood assigned to

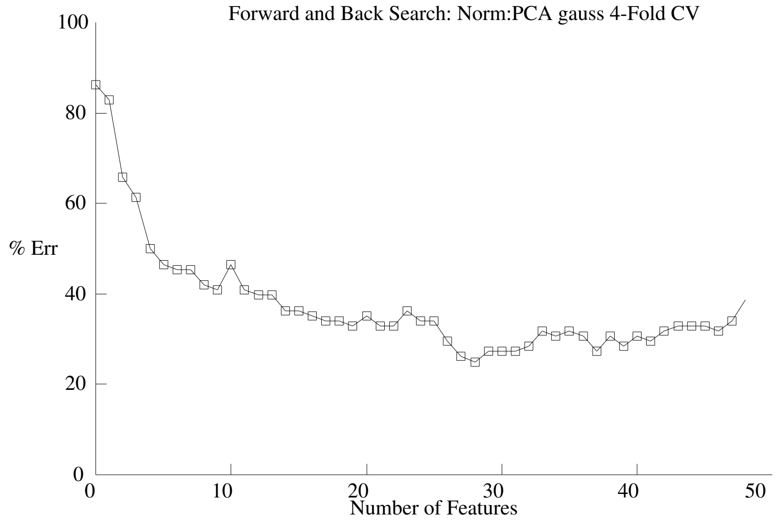

Figure 12: Feature search using Gaussian classifier with PCA normalization. The graph shows how this classification algorithm does not yield the sharp drop that LDA normalization does.

each bin is proportional to the number of training patterns that fall in that bin divided by the bin width. In testing, the likelihoods for each input dimension are multiplied to give an overall likelihood for each class [KL95].

The LNKnet Histogram classifier provides several options for dividing the input space into bins. The one that seems to work best for our particular application is the Uniformly Segmented Hypercubes option in which a fixed set of bins evenly divides the space into smaller hypercubes. Also, this option seems to work best with a PCA normalization for the dataset.

An optional per class diagonal Gaussian classifier can be used to determine the class of all patterns that fall outside histogram bins. This greatly improves the performance of the classification and hence seems to indicate that the data we are using is not naturally suited for classification using this algorithm. Our best classification had an error of 26%.

The ten metrics that seem to work best with this particular classifier are STY1iM (Indentation of close curly brackets), STY1g (Percentage of closed curly brackets that are the last character in a line), STY1aS (Standard deviation for indentation of C statements within surrounding blocks), STY1hM (Indentation of open curly brackets), PSM2 (Percentage of void function definitions), PRO6 (Use of conditional compilation), STY1d (Percentage of open curly brackets that are the last character in a line), PSM5 (Lines of code per function), PRO2bS (Standard Deviations for global variable name length), and PRO3d (Percentage of function names that start with an uppercase letter).

5.2.5 Binary Tree

The Binary Tree (BINTREE) classifier is rule based classifier. These partition the input space into decision regions using threshold logic nodes or rules. The BINTREE classifier works well with problems with a small number of uncorrelated input features, but may have difficulty on databases with high dimensionality [KL95].

The performance of the BINTREE classifier is less than optimal, with an error rate of 30%. However, we include this classifier here because it is possible that for some databases, the forward and/or backward feature search might be able to find a small number of uncorrelated metrics that might be sufficient to discriminate among the users in the database.

6 Discussion and Conclusions

The experiments we have performed for this paper support the theory that it is possible to find a set of metrics that can be used to classify programmers correctly. Within a closed environment, and with a limited number of programmers, it is possible to identify authorship of a program by examining some finite set of metrics.

At least two different classification methods (Gaussian Likelihood Classifiers and MLP Neural Networks) were able to classify the programs in our experiment with very low error rates (i.e. less than 2%). Four other methods were able to classify the programs with error rates below the 30% mark.

Note that it is puzzling, however, that these classification mechanisms were able to classify the programmer that varied his programming style tremendously over a brief period of time. Two explanations are possible: 1) The programmer is indeed consistent in a way that is not evident to the human eye. The consistencies can be detected by the classification mechanisms. 2) The programmer is indeed inconsistent, and the classification mechanisms have assigned him to a class of his own precisely because he was so inconsistent.

After close visual examination of the source code provided by all the programmers involved in our experiment, and analysis of distribution diagrams for some of the more useful metrics, we have reason to believe that programmers tend to show repeating patterns in their programs. As expected, programmers are skillful with a limited set of constructs, mainly those that are well known to them and that allow them to write programs faster and more reliably. It would be unrealistic to assume that any programmer can develop programs efficiently and correctly using an unfamiliar programming style. This does not only apply to the structure of the programs, but also to the look and feel of it; such metrics as, for example, average blank lines over lines of code can indeed remain surprisingly constant. Programmers organize information on the screen such that logically independent portions of the code can be easily recognized.

Even though we are satisfied with our choice of metrics, it is possible that with them we would not be able to correctly classify a larger set of programmers successfully. Experience and logic tell us that a small and fixed set of metrics are not sufficient to detect authorship of every program and for every programmer. Also, by no means do we claim that the set of metrics we examined is the only one that might yield stable measurements. During the data collection and analysis of the experiment, we noted that the following metrics might be of considerable use in future experiments:

1. Use of revision control system headers. We were surprised to see that a considerable portion of the programmers examined used the automatic identification and log features of the RCS Revision Control System. As an added bonus, such identification strings will provide the login name of the programmer in question[14].
2. Another metric that could have been used successfully is the use of literals in code versus the use of global constants.
3. One programmer's idea of debugging statements was commenting out the print statements. This was done consistently and it might provide another useful metric.

We do not expect that the metrics calculated for any given programmer would remain an accurate tag for a programmer for a long time, even though in our experiment we have correctly identified the only programmer who provided code developed over a number of years. Further research must be performed to examine the effect that time and experience has on the metrics examined on this document. Also, we theorize that we may be able to apply Bayesian techniques to improve the performance of the discriminant analysis performed with the SAS tool, avoiding the overhead imposed by the LNKnet neural networks. Further research is needed to verify this conjecture.

It would be logical to conclude that for the authorship analysis techniques to work, the metrics would have to be gathered continually over time. Compilers and operating systems would have to be enhanced and significant research

[14] It is easy to alter the user name in the RCS automatic identification feature, and as such, excessive confidence must not be placed on its accuracy

would have to be done in the development of operating systems to enforce the use of these metrics.

The results of this paper support the conclusion that within a closed environment, and for a limited set of programmers, it is possible to identify a particular programmer and the probability of finding two programmers that share exactly those same characteristics should be very small.

Acknowledgments

This work was partially supported by a gift from Sun Microsystems to the COAST Laboratory: that support is gratefully acknowledged. Thanks to those people who contributed source code for the experimental stages. Thanks to the COAST students and faculty members that contributed with comments and new ideas while proofreading this document. Thanks to the people at the MIT Lincoln Laboratory for the development of a great *free* tool (LNKnet).

References

[BB89] A. Benander and B. Benander. An empirical study of COBOL programs via a style analyzer: The benefits of good programming style. *The Journal of Systems and Software*, 10(2):271–279, 1989.

[BM85] R. Berry and B. Meekings. A style analysis of C programs. *Communications of the ACM*, 28(1):80–88, 1985.

[BS84] H. Berghel and D. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65–76, 1984.

[Coo87] Doug Cooper. *Condensed Pascal*. W. W. Norton and Company, 1987.

[Dau90] K. Dauber. *The Idea of Authorship in America*. The University of Wisconsin Press, 1990.

[Dis37] B. Disraeli. *Venetia*. New York and London, 1837.

[EV91] W. Elliot and R. Valenza. Was the Earl of Oxford the true Shakespeare? *Notes and Queries*, 38:501–506, December 1991.

[Eva84] M. Evangelist. Program complexity and programming style. In *Proceedings of the International Conference of Data Engineering*, pages 534–541. IEEE, 1984.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, first edition, 1991.

[Gri81] S. Grier. A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin*, 13(1):15–20, 1981.

[GS92] S. Garfinkel and E. Spafford. *Practical Unix Security*. O'Reilly & Associates, Inc., 1992.

[Han91] D. Hanson. Code generation interface for ANSI C. *Software - Practice and Experience*, 38:963–988, September 1991.

[HH92] W. Hope and K. Holston. *The Shakespeare Controversy*. McFarland & Company, 1992.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, first edition, 1979.

[Jan88] H. Jankowitz. Detecting plagiarism in student Pascal programs. *Computer Journal*, 31(1):1–8, 1988.

[JW88] R. Johnson and D. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, second edition, 1988.

[KL95] Linda Kukolich and Richard Lippmann. *LNKnet User's Guide*. MIT Lincoln Laboratory, MIT Technology Licensing Office, RM E32-300, 200 Carleton Street, Cambridge, MA 02142-1324, April 1995.

[KP78] B. Kernighan and P. Plauger. *The Elements of Programming Style*. McGraw-Hill Book Company, second edition, 1978.

[Krs94] Ivan Krsul. Authorhip analysis: Identifying the author of a program. Master's thesis, Department of Computer Science, Purdue University, 1994.

[KS95] Ivan Krsul and Eugene Spafford. Authorship analysis: Identifying the author of a program. In *Proceedings of the 18th National Information Systems Security Conference*, pages 514–524. National Institute of Standards and Technology, October 1995.

[LC90] A. Lake and C. Cook. STYLE: An automated program style analyzer for Pascal. *ACM SIGCSE Bulletin*, 22(3):29–33, 1990.

[Led87] Henry Ledgard. *C With Excellence: Programming Proverbs*. Hayden Books, 1987.

[LS93] T. A. Longstaff and E. E. Schultz. Beyond preliminary analysis of the WANK and OILZ worms: A case study of malicious code. *Computers & Security*, 12(1):61–77, 1993.

[OC89] Paul W. Oman and Curt R. Cook. Programming style authorship analysis. In *Seventeenth Annual ACM Computer Science Conference Proceedings*, pages 320–326. ACM, 1989.

[OC90] P. Oman and C. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, 1990.

[OC91] P. Oman and C. Cook. A programming style taxonomy. *Journal of Systems Software*, 15(4):287–301, 1991.

[Ott77] K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, 1977.

[RN93] J. Ranade and A. Nash. *The Elements of C Programming Style*. McGraw-Hill Inc., 1993.

[SAS] The SAS Institute. *SAS/STAT User's Guide. Volume 1, ANOVA–FREQ*, fourth edition.

[SCS86] H. Dunsmore S. Conte and V. Shen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, 1986.

[Spa88] Eugene H. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science. Purdue University, 1988.

[Spa89] Eugene H. Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19(1), January 1989.

[Tas78] Dennie Van Tassel. *Program Style, Design, Efficiency, Debugging, and Testing*. Prentice Hall, 1978.

[Wha86] G. Whale. Plague: Detection of plagiarism using program structure. In *Proceedings of the Ninth Australian Computer Science Conference*, pages 231–241, 1986.

[WS93] Stephen A. Weeber and Eugene H. Spafford. Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595, December 1993.