Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics

Robert Lange
Department of Computer Science
Drexel University
Philadelphia, PA 19104, USA
rcl24@drexel.edu

Spiros Mancoridis
Department of Computer Science
Drexel University
Philadelphia, PA 19104, USA
spiros@cs.drexel.edu

ABSTRACT

We have developed a technique to characterize software developers' styles using a set of source code metrics. This style fingerprint can be used to identify the likely author of a piece of code from a pool of candidates. Author identification has applications in criminal justice, corporate litigation, and plagiarism detection. Furthermore, we can identify candidate developers who share similar styles, making our technique useful for software maintenance as well. Our method involves measuring the differences in histogram distributions for code metrics.

Identifying a combination of metrics that is effective in distinguishing developer styles is key to the utility of the technique. Our case study involves 18 metrics, and the time involved in exhaustive searching of the problem space prevented us from adding additional metrics. Using a genetic algorithm to perform the search, we were able to find good metric combinations in hours as opposed to weeks. The genetic algorithm has enabled us to begin adding new metrics to our catalog of available metrics. This paper documents the results of our experiments in author identification for software forensics and outlines future directions of research to improve the utility of our method.

Categories and Subject Descriptors

I.2.6 [**Computing Methodologies**]: Artificial Intelligence-Learning

General Terms

Experimentation, Measurement, Languages

Keywords

Genetic algorithm applications, pygene, author identification, software forensics, software metrics

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

1. INTRODUCTION

A major research topic in the field of software forensics is author identification. Author identification involves "having samples of code for several programmers and determining the likelihood of a new piece of code having been written by each programmer" [7]. This paper examines author identification via source code style. With respect to software authorship, style consists of any decision over which the developer has discretion. Gray [7] identifies three categories of style: the algorithms used to solve a problem, the cosmetic layout of the code, and the choices made when naming variables and functions. Stylistic variation among developers is expected.

We put forth a method of extracting metrics from source code and representing these metrics as histogram distributions. Stylistic elements of the latter two categories can be represented readily using our metric histograms and we plan to support more advanced code-complexity metrics in the future. We hypothesize that for some set or sets of metrics, the source code written by different developers will display measurably different histogram distributions and that these distributions will vary between developers more than they vary within the code belonging to one developer.

Measuring the differences between histogram distributions of code under scrutiny with those associated with code from a pool of known developers should produce a ranked list of likely authors. Given the hypothesis, the actual author of the code should be the first author selected in the list in most cases. This strong form of author identification is useful in several real world applications, including:

Criminal Prosecution Author identification can be used by authorities to identify the author of a piece of malware, given a pool of code belonging to suspects.

Corporate Litigation If a company suspects a former employee of violating a no-compete clause of a contract, author identification can help to determine whether the employee wrote the suspect code.

Plagiarism Detection Our technique can be used by academic institutions to help determine whether suspect code was written by a claimed author or another.

Even in cases where the actual author is not identified, knowing which developers' styles are most similar to that of the code under scrutiny can be useful. For example, in a corporation with a pool of developers available to maintain a piece of legacy code, the developer whose style most re-

sembles that of the original author may be most productive in maintaining the code.

The goal of this paper is to answer the following questions:

- How well can histogram distributions of code metrics identify the style of a developer?
- What combinations of metrics provide the best identification of a developer's style with the code under scrutiny?

To test our technique's ability to identify the author of an arbitrary source code project, we have implemented a toolkit and tested it using a set of Free Software projects as described in section 4. The computational resources and time necessary to search for good metric combinations prevented us from implementing many metrics. We found that using a genetic algorithm to perform a randomized heuristic search returned acceptable metric combinations in a fraction of the time necessary for an exhaustive search. Thanks to our use of a genetic algorithm, we are able to begin adding new metrics again. This paper describes our method, experimental data set, tools, and the results of our analysis. We conclude by presenting direction and goals for continuing research on this topic.

2. RELATED WORK

Numerous papers have been written about software forensics and the prospects of identifying authors based on code characteristics. Oman and Cook [10] performed preliminary investigations into style and its relationship with authorship. Spafford and Weeber discussed preliminary concepts such as structure and formatting analysis [14]. Sallis compared software authorship analysis to traditional text authorship analysis [12].

Gray, Sallis, and MacDonell have published multiple articles fleshing out concepts of author identification via metrics [7] and case-based reasoning [13]. They have also tackled author discrimination [9] via case-based reasoning and statistical methods.

More recently, Ding and Samadzadeh used statistical analysis to create a fingerprint for Java author identification [5]. Their technique makes use of several dozen metrics (e.g., mean function name length) which they have statistically correlated to identify developers. They determined that metrics relating to code layout have a higher significance in identifying authors than other metrics. Their metrics extraction technique is similar to our own. However, where they use mostly scalar metrics derived from the source code, our metrics are formulated as histogram distributions. Our results are roughly comparable to theirs, with approximately one third of the metrics.

3. TECHNIQUE

The method we have developed for style fingerprinting involves the formation of histogram distributions of metrics that were extracted from the source code. Classification of developer style is done by measuring the difference between the histogram derived from a target project and the corresponding histogram of a known user.

3.1 Histograms

No single code metric data point is likely to imply much about an developer's style. We propose a technique in which code metrics are represented as histogram distributions. The canonical example of this is the *line-len* metric which measures the number of characters in one line of source code. In a histogram representation of *line-len*, the x-axis of the histogram corresponds to every recorded line length, in other words the set of non-negative integers. For every point on the x-axis, the corresponding y-axis point represents the number of times a line of that length existed in the code.

After generating a raw histogram from the source code, we normalize it by dividing the value of each y-axis point by the sum of all y-axis points. This ensures that the sum of all y-values on the normalized histogram distribution is 1. Normalization is essential to the ability to compare a large mass of code and a small mass of code. Without normalization, the y-axis values would be unbounded, and thus a large mass of code would inherently produce a histogram that differed greatly from the histogram generated by a small mass of code. With normalization, the y-axis values are bounded, so that the "shape" of the two histogram distributions can be compared, without respect to the magnitude of the raw y-axis values. See Figure 1 for an example of a normalized histogram.

3.2 Classification of Developer Style

Classification is completed when a decision is made about whether a subject falls into one category or another. In the case of author identification, the decision is whether a piece of code was written by a particular developer.

The classification method we propose is the nearest neighbor search [2] using a general distance. Specifically, the difference between the styles of developers i and j is given by Equation 1 where M is the set of chosen metrics, $X(m)$ is the set of x-axis coordinates associated with metric $m \in M$, and $f_i(m, x)$ is the function of histogram values of developer i associated with the coordinate $x \in X(m)$.

$$E = \sum_{m \in M} \sum_{x \in X(m)} \mid f_i(m,x) - f_j(m,x) \mid \qquad (1)$$

The nearest neighbor search identifies the developer with the minimal difference in histogram distributions from that of the code under scrutiny.

3.3 Precision

Nothing about the classification method proposed requires that it produce a single result. In fact, the nearest neighbor classifier can produce a ranked list of developers in order of descending likelihood of authorship. When we speak of the precision of a result, we mean how large the ranked list must be to capture the true author of the code. 1-precision means that the classifier identified the true author as the most likely author of the code. In other words, the style of the code under scrutiny most resembled that of other code written by the same developer. 2-precision indicates that the true author was ranked as the second-most likely author by the classifier and that the style of the code of one unrelated developer was a closer match to the style of the code under scrutiny than that of other code written by the true author. 3-precision indicates the true author was ranked behind two other authors, and so forth.

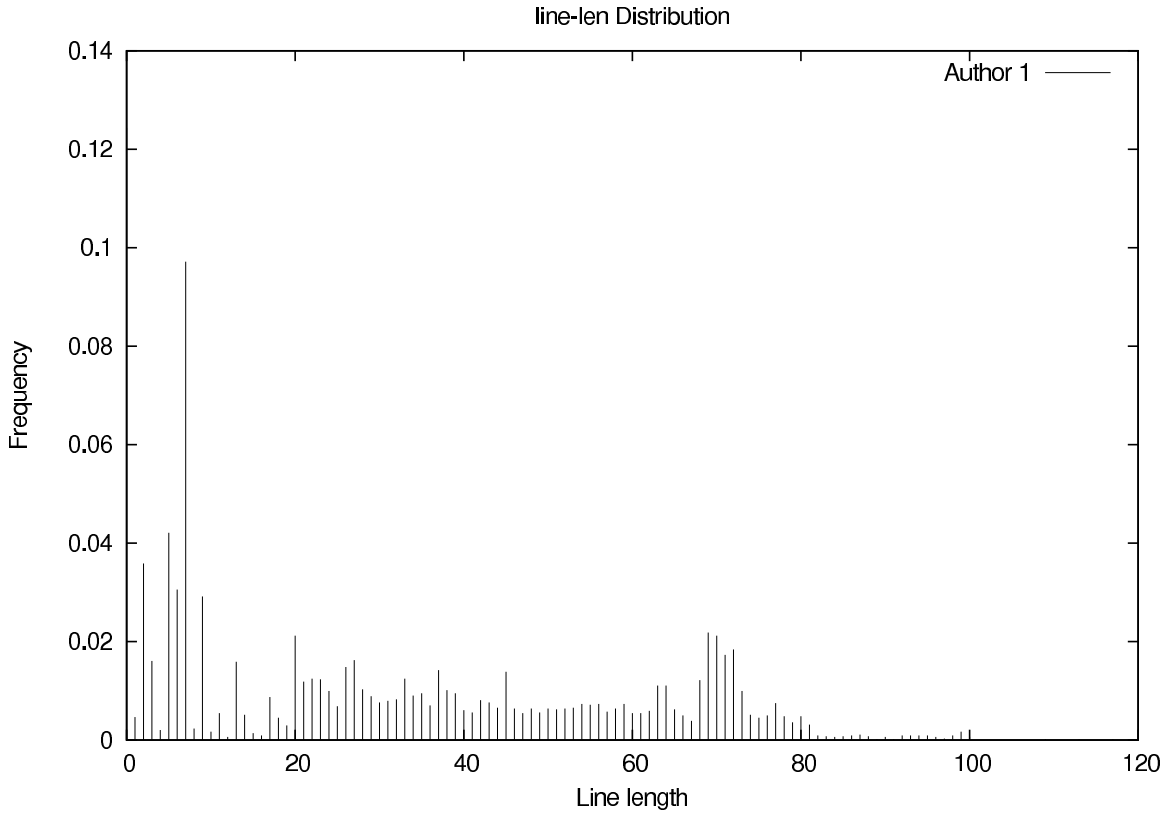

Figure 1: An example line-len histogram distribution

4. DATA SET SELECTION CRITERIA

Careful consideration was made for the contents of the case study data sets. The data must represent a diverse population of developers and provide enough information from each developer to ensure that a valid comparison of their styles could be made. The data set must also be large to reduce the chance of good results occurring merely by chance. Finally, the data must be as close to "real world" data as can be reasonably obtained for open academic study.

4.1 Simplifying Assumptions

Certain simplifying assumptions have been made to allow an initial investigation into the feasibility of our approach and of style fingerprinting in general. In later stages of investigation, we plan to remove these restrictions to take into consideration code that is more likely to be found in industrial practice.

Single Author Author discrimination involves identifying how many developers worked on a piece of code and over what sections of the code each developer exercised influence [7]. While this ability is essential for most real world purposes of author identification, we consider it to be an advanced functionality. To avoid any confusion caused by attempting to match multiple developers to a single code project, we have opted to limit the projects considered for use in the data set to original projects with a single author.

Single Language In principle our technique is independent of the programming language used. In practice, some amount of variation on metrics and some development of extractors will be necessary to adapt our tools to analyze multiple languages. To maximize time spent on data analysis, the initial data set consists of Java source code exclusively. Our tool framework can be extended easily, so additional metrics for use with other languages can be added as necessary.

4.2 Considerations

In addition to the above simplifying factors, several considerations factored into the decision of how to build the data set. To ensure that the data would provide a useful test of our approach, the data set met the following list of requirements.

Breadth of Source The execution of the experiment consists of two phases: a supervised learning phase in which the tool is fed source code from known authors, and a testing phase in which the tool is given a source project and required to rank a list of authors in order of similarity of style. In order to separate developer-specific styles (the wheat) from project-specific styles (the chaff) we perform two testing phases per developer. As such, at least three projects are necessary from each known developer: one for the learning set and two for testing.

It is vital that source code files belonging to the same project not be used in both the learning and testing sets. This cross-pollination may inadvertently bias the tool toward selecting the correct author, as project-wide conventions are likely to exist. Therefore, each

author in the data set has code from no less than three independent projects.

Note also that although none of our metrics examines the full contents of words, author and project names (for example, existing in comments at the top of a file) have been stripped from code files to eliminate even the slightest hint of bias.

Depth of Source Before a developer is considered for inclusion in the data set, at least one thousand source lines of code (SLOC) must be available per project. This requirement is vital to ensure that a sufficient amount of code is provided for analysis.

4.3 Current Test Data

Our initial data set consisted of 24 Free Software projects written by twelve developers and hosted on SourceForge [1] along with several academic projects contributed by 8 students. We experienced success in finding metric combinations that could classify 11 out of 12 of the SourceForge authors and 6 of the 8 student authors correctly for this data set. Our success led us to be concerned that this type of data set was insufficiently challenging to represent real world code. This experience led us to establish the rules that only substantial projects be considered and that no developer could be considered without 3 independent projects.

Subject to the simplification factors and other considerations, 60 Free Software projects were selected from SourceForge [1] for use as the primary challenge set in the case study. Twenty developers each authored 3 of the projects. Each project was examined and determined to be reasonably independent of its author's other projects. Due to the length of the lists, neither set of projects have been included in the bibliography. The authors of this paper will provide a copy of the data sets on request.

The challenge data set is divided into three sets: one learning set and two testing sets. One project from each developer is placed into each set. The learning set consists of 1371 files with approximately 240,000 SLOC and approximately 8.3MB worth of data. The first testing set consists of 1149 files, 200,000 SLOC, and 6.6MB of data. The second testing set consists of 1548 files, 230,000 SLOC, and 8MB of data.

5. EXPERIMENTAL METHOD

We developed a set of tools and scripts for extracting and manipulating code metrics. This section describes the tool architecture, use, and initial results.

5.1 Metrics

Each metric is designed such that a meaningful histogram distribution can be constructed from it. Abstractly, each histogram distribution represents the "shape" of a piece of an author's fingerprint.

Metrics histogram distributions are generated using two distinct methods. Text-based metrics are extracted using simple plain-text analysis of the source code. The code is treated as a document, and examined for content such as simple character strings. Text-based metric extraction is very quick and simple to program, however it is difficult to capture higher-level program understanding with text-based methods. For this, we parse the source code and extract metrics with full understanding of the syntax of the code.

Several of the metrics use the word "word" as part of their metric description. In this context, "word" refers to a chunk of text that is delimited by whitespace or some syntactical characters, which may lex to more than one lexeme, but which is treated as a single chunk for purposes of the metric. For example, `ns.obj.memb.func()` is treated as one word for the purposes of the word-based metrics.

The *line-len* metric was discussed in Section 3.1. The following list enumerates the additional 17 metrics. [1] Unless otherwise noted, the y-axis value of each histogram represents the relative frequency of the corresponding x-axis value.

access measures the relative frequencies of the Java class member access protection scopes: public, protected, and private. We have observed that different developers hold different opinions on the utility of the class member access control system, and we hope to capture their preferences with this metric.

brace-pos measures the relative frequencies of curly brace positions throughout the code. This metric was inspired by well-known online flame wars over the "correct" positioning of curly braces. The x-axis values enumerate the following possible configurations: an open (left) brace alone on a line; an open brace as the leftmost non-whitespace character on a line; an open brace on the interior of non-whitespace characters on a line; an open brace as the rightmost non-whitespace character on a line; and the same four configurations repeated for the close (right) brace.

comment measures the relative frequency of uses of the three different types of Java commenting: block comments, line comments, and JavaDoc comments.

control-flow measures the relative frequency of the use of various control-flow techniques available in Java. The x-axis values enumerate the following possible control flow options: for, foreach, while, do-while, if, switch, throw, and function-call.

indent-space indent-tab measure the indentation whitespace used at the beginning of each line. We have observed that different users have different conventions for indentation, and this metric attempts to capture that difference. The x-axis value represents the number of the given whitespace characters at the beginning of each line.

inline-space inline-tab measure the whitespace that occurs on the interior areas of each non-whitespace line. We have observed that some users tend to minimize interior whitespace, while others carefully add interior whitespace, typically to align syntactic elements vertically within the source file. The x-axis value represents the number of the given whitespace characters on the interior of each line.

[1]Note that metrics are named as space-delimited words. Certain metrics appear grouped together in this list for the sake of brevity and because they function similarly to one another, but they are separate metrics as far as classification is concerned.

trail-space trail-tab measure the trailing whitespace at the end of a line. We have observed that while most users ignore whitespace at the end of lines, while some users tend to carefully eliminate such useless whitespace. The x-axis value represents the number of the given whitespace characters at the end of each line.

period measures the use of the namespace member and class member operator in the Java language, both of which are represented by a period. *period* is intended to help determine whether a developer prefers to string together long constructs such as `ns.obj.memb.func()` and `object.function().nextFunction()`. The histogram x-axis value represents the number of period operators in a singe logical identifier.

underscore measures the use of the underscore character in identifiers. We have observed that many people do not use underscore characters in their identifiers, and most developers have predominantly zero underscores or predominantly many underscores, with sharp distinction between the two groups. The x-axis values represent the number of occurrence of an underscore per identifier.

switch switch-case measure the complexity of switch statements used in the code. Java switch statements permit multiple case statements to resolve to a single "case group" of executable code. In *switch*, the x-axis values represent the number of case-groups per switch statement, while in *switch-case*, the x-axis values represent the number of actual case statements in the switch.

line-words measures how densely the developer packs code constructs on a single line of text. The x-axis value represents the number of words on a line.

word-first-char measures the frequency of the first characters used in identifiers, primarily with the hope of catching users of prefix and Hungarian notation. The x-axis value represents the integer value of the first character of an identifier.

word-len measures the length of identifiers. With this metric we hope to discriminate between developers with terse naming styles and developers with verbose naming styles. The x-axis value represents the length of an identifier.

5.2 Author Matching with the Classifier

The classifier accepts two pieces of input: an unknown batch of source code to be matched, and a list of metrics (out of the list of possible metrics) to consider. In our experimental setup, the classifier examines an entire source code project at a time. We have done this for performance reasons, to reduce the number of comparisons the classifier makes. The classifier runs the extractors on the unknown code, generates normalized, interpolated histograms, and then compares the resultant unknown histogram set to the pool of known users' histogram sets according to the list of metrics it was given.

For each pair of a learned histogram and an unknown histogram, the classifier calculates the general distance between the histograms. The classifier then ranks each known user according to the distance. With a good metric combination, a short distance will mean that the two developers' styles are similar.

5.3 Testing and Definition of Success

In this experimental setup, the classifier must classify 40 projects (2 sets of 20 projects) amongst 20 developers. For purposes of this experiment, a success is defined as the classifier classifying both projects belonging to a given author correctly. No "partial credit" is given if the classifier incorrectly classifies the author of a file.

5.4 Initial Results

Initially, the classifier was tested using only a few simple metric configurations, such as single metrics or all metrics for classification. This helped us verify that our code was implemented properly.

Four of the metrics performed quite well. The *line-len* metric successfully classified 45 percent of the testing set, while *brace-pos* and *line-words* classified 25 percent successfully and *word-first-char* classified 20 percent successfully. The all-metrics test performed poorly, with a 30 percent success rate. Clearly, some subset of the metrics will be needed to get better results.

Using a single metric, the classifier was able to classify all 40 projects in just a few seconds on an AMD Athlon 1600. However, the classifier runs in time roughly proportional to the number of metrics used for classification, so testing with more metrics will take proportionately longer. Typical runtime for an execution of the test is approximately 30 to 40 seconds. Because runtime of the classifier is directly proportional to the number of metrics used, the all-metrics test ran longest at about 55 seconds.

6. GA SEARCH

With the validation of the initial single-metric matching technique, the question became: which of the metric combinations produces the best classification? Without any prior knowledge of how results are distributed across the various combinations, an exhaustive search seemed reasonable. Seventeen metrics result in $2^{18} - 1 = 262,143$ possible metric combinations.

Even with a typical runtime of about 30 seconds per execution, this process would require at least 65 days to complete on a uniprocessor. Even if parallelized to 12 nodes, the process would take more than one week of solid compute time to complete. While even this figure represented the outer edges of feasibility for our research group, each addition of a new metric would double the runtime of the job. As an open research problem, we had hoped to be able to add and alter metrics frequently. The time and resources necessary for each exhaustive search severely limited our ability to accomplish this, so we decided to replace the exhaustive search.

A genetic algorithm [6] seemed a reasonable alternative to an exhaustive search. From our early results with exhaustive searches on smaller versions of this problem, we knew that the search space contained many "good" solutions. Our classifier was written in the Python programming language, so we looked for a Python-based genetic algorithm toolkit. Pygene fit our needs [8].

Pygene is a Free Software genetic algorithms toolkit with a very simple interface. Little documentation is provided, aside from PyDoc API specifications and a handful of ex-

amples. However, the simplicity of the interface and the intuitive naming of classes and variables obviates the need for much documentation.

6.1 Problem Representation

The Pygene genetic algorithm toolkit provides three main building blocks from which to construct a GA: genes, organisms, and populations. Genes provide an abstraction for the lowest level piece of information in the GA, namely the individual problem parameters we wish to optimize. In our case, we wish to optimize the set of metrics active in our classifier in order to maximize the accuracy of the classification. We currently require each metric to be active or inactive, so the natural representation of a metric is a binary digit using Pygene's BitGene class.

A genome is made of a string of genes and provides the "genetic" representation for the organism. In Pygene, an Organism subclass represents one complete candidate solution to the problem. In our case, each organism is one metric combination. Any Organism subclass must provide a fitness function, which evaluates the quality of the solution relative to other solutions. In our solution, the fitness function provides the bridge into our classifier.

Pygene offers two basic options for organisms. The standard Organism operates using single-helix reproduction, in which two mating organisms swap a certain number of genes and produce two new organisms. The MendelOrganism contains a pair of gene instances for each gene in the genome. For simplicity, we chose to implement the standard Organism rather than dealing with the extra complexity of the MendelOrganism.

Finally, the Population consists of a set of organisms. The population is initially composed of randomly selected organisms. The Pygene Population base class is simple in structure and function, containing the organisms and directing them to reproduce. The Population's operational parameters allow us to directly impact the operation of the genetic algorithm.

6.2 Fitness Function

The fitness function of a genetic algorithm provides a means of discriminating good solutions from bad solutions. In our case, we need to reward metric combinations which are able to identify the authors of the both testing sets with higher levels of precision. Each execution of a fitness function obviously corresponds to a full test run of the classifier.

We interfaced the classifier to the genetic algorithm so that, instead of outputting a ranked list of authors for each project in both testing sets, the classifier produced a single list identifying the number of correct classifications at each level of precision, as exemplified in equation 2. The list is read from left to right as, "1-precision resulted in 9 out of 20 correct, 2-precision resulted in 11 out of 20 correct," and so forth. Note that the list is monotonically increasing from left to right, as a correct classification at 1-precision guarantees a correct guess at 2-precision, and so forth.

$$[9, 11, 13, 13, 14, 14, 15, 15, 17, 17, 18, 18, 19, 19, 19, 20, ...] \quad (2)$$

Pygene expects a single numeric score as its fitness function. As we desire to maximize the correct classifications at the highest level of precision, we created equation 3 where f is the fitness function and L is the list of results.

$$f = \sum_{i=0}^{len(L)-1} L_i * len(L)^i \quad (3)$$

We encountered one unexpected problem early in our use of Pygene. We had assumed that Pygene executed the fitness function of each organism once per generation. However, the execution time of the GA was quite slow, and analysis of our log files showed the fitness function executing many times per organism per generation. Apparently, Pygene assumed the fitness function was inexpensive, which was not the case with our experiment. We corrected this problem by implementing a results-cache. The cache is implemented as an associative array with the organism genome as key and the fitness result as the value. This cache permits the execution of up to several hundred generations per day, compared to fewer than a dozen before the optimization.

6.3 GA Parameters

Pygene provides a number of operational parameters to control and modify the behavior of the genetic algorithm without requiring significant additional programming. The Population base class is the primary repository for these parameters. Population permits a set of initial organisms to be passed into the module, otherwise it builds organisms randomly, which we chose to allow it to do. Population parameters are available to control the initial population and the number of children to create each generation. Organisms are mated probabilistically based on fitness function, and Pygene offers two additional parameters to control the genetic makeup of a new generation. One parameter sets the number of lowest-fitness children to cull after the generation, before they are given an opportunity to mate. The other parameter activates elitism and sets the number of highest-ranking organisms to propagate unchanged into the next generation. Another parameter controls the chances that organisms will be selected to mutate. One final parameter permits the introduction of randomly-generated new organisms into the population, but this parameter is off by default.

Two additional important parameters are kept outside of the Population class. The Organism class (not the MendelOrganism) has a crossover rate parameter, which affects the percentage of the genome that is exchanged during mating. All Gene classes also have a mutation probability parameter, which sets the probability of the gene undergoing mutation if its owning organism is selected for mutation.

The observed behavior of the genetic algorithm depended heavily on the specific population and reproductive parameters. The default Pygene parameters were configured conservatively, with an initial population of 10, an elitism rate of 10, a child count of 100 and a child cull of 20. In our experience, these settings resulted in our initial experiments converging prematurely on poor solutions. At the time of this writing, we have Pygene set with an initial population of 300, with 300 children created in each generation. We observed that these numbers struck a good balance between generational processing time and convergence.

To further combat premature convergence, we disabled elitism. This posed no real loss to us, as a record of every organism created and its fitness is kept in our logs. We also minimized child culling to 10, for the same reason.

We increased the gene mutation probability to 5 per-

cent in order to encourage a measure of instability even after the genetic algorithm has begun to converge. Likewise, we activated the new organism feature and inserted 10 new, randomly-generated organisms into the population after each generation. We had intended these measures to ensure that the population remained active in the face of local maxima. However, they resulted in populations that failed to converge even after several days of processing. After additional experimentation we disabled the insertion of new organisms into the population and fixed gene mutation probability at 3 percent.

6.4 Results

Our initial experiments with Pygene were not directly productive toward the goal of identifying good metric combinations. Nevertheless, this experience gave us a feel for how Pygene's parameters controlled the execution of the algorithm. After choosing the parameters we felt would prevent premature convergence without totally destabilizing the algorithm, we executed Pygene and let it run until it converged to a stable, uniform population, which occurred after approximately 8 hours.

Using substantially the same parameters, we ran Pygene 8 additional times to see to what extent the result changed with each execution. The algorithm converged to highly similar-quality results on each execution, despite beginning with randomly chosen initial organisms at each execution. On 4 executions the algorithm converged to the same solution. Convergence typically became obvious after approximately 4 hours, with minor improvements coming afterward until stabilization into a totally uniform or oscillatory population. Even when left running for more than a day, the algorithm did not break out of its stable population. This leads us to believe that the algorithm was able to find a solution at or close to the optimal solution, even though it only searched a fraction of the problem space.

The best 1-precision metric combination achieved 55 percent correct classification. With 3-precision the metric combination was able to correctly classify 75 percent of projects. These values are encouraging and are likely to improve as more metrics are added to the experiment. The genetic algorithm has enabled us to continue adding metrics and we are working on new metrics at the time of this writing.

Precision	Success	Combination
1	11/20	inline-space inline-tab line-len line-words period trail-space trail-tab[2] underscore
2	14/20	brace-pos comment control-flow indent-space inline-space inline-tab line-len line-words period trail-tab underscore word-first-char word-len
3	15/20	brace-pos comment indent-space inline-space inline-tab line-len line-words period switch trail-tab underscore word-first-char

Table 1: Best metric combinations at 1, 2, and 3-precision

7. CONCLUSIONS AND FUTURE WORK

Several important conclusions can be drawn from the work done on this project. First, we have seen evidence that, given the proper metric combination, histogram distributions of code metrics can identify authors more than half of the time. Note that of the set of metrics we have implemented so far, the best combination appears to be *inline-space inline-tab line-len line-words period trail-space trail-tab underscore*, although this may change as a result of a substantial change in the data set. The best metric combination will likely need to be recalibrated for every data set.

Another conclusion we can draw is that if a combination of metrics identifies a developer's style strongly on one set of code, that same set will not necessarily identify the developer's style well on another set of code. To reduce the negative effect of project-specific rather than developer-specific style fingerprints, we now require no less than three independent projects per developer in our data set.

Further work needs to be done to improve the accuracy of our method and to prepare our technique for more real world use. Foremost on the list of tasks is improving the test data set and expanding the size and diversity of the data.

One of the most obvious ways to improve accuracy is to produce a wider selection of metrics from which to choose. These include additional text-based metrics as well as language-dependent metrics. The use of a genetic algorithm to find good metric combinations allows us to experiment with new metrics without experiencing an exponential growth in the time necessary to search for good metrics. We plan to add metrics vigorously in an effort to ramp up the quality of our results.

Entropy analysis is a technique that can be used to find data points with high between-developer variability and eliminate them from consideration. While we desire metrics that demonstrate a developer's whole style, rather than merely a few unique points, finding and eliminating ubiquitous data points would be helpful.

Finally, nearest neighbor is not the only classification strategy. We intend to implement other classification strategies (e.g., C4.5 decision trees [11], Vote Feature Intervals [4], or Bayesian networks [3]) to gauge their benefits to both author identification and style fingerprinting.

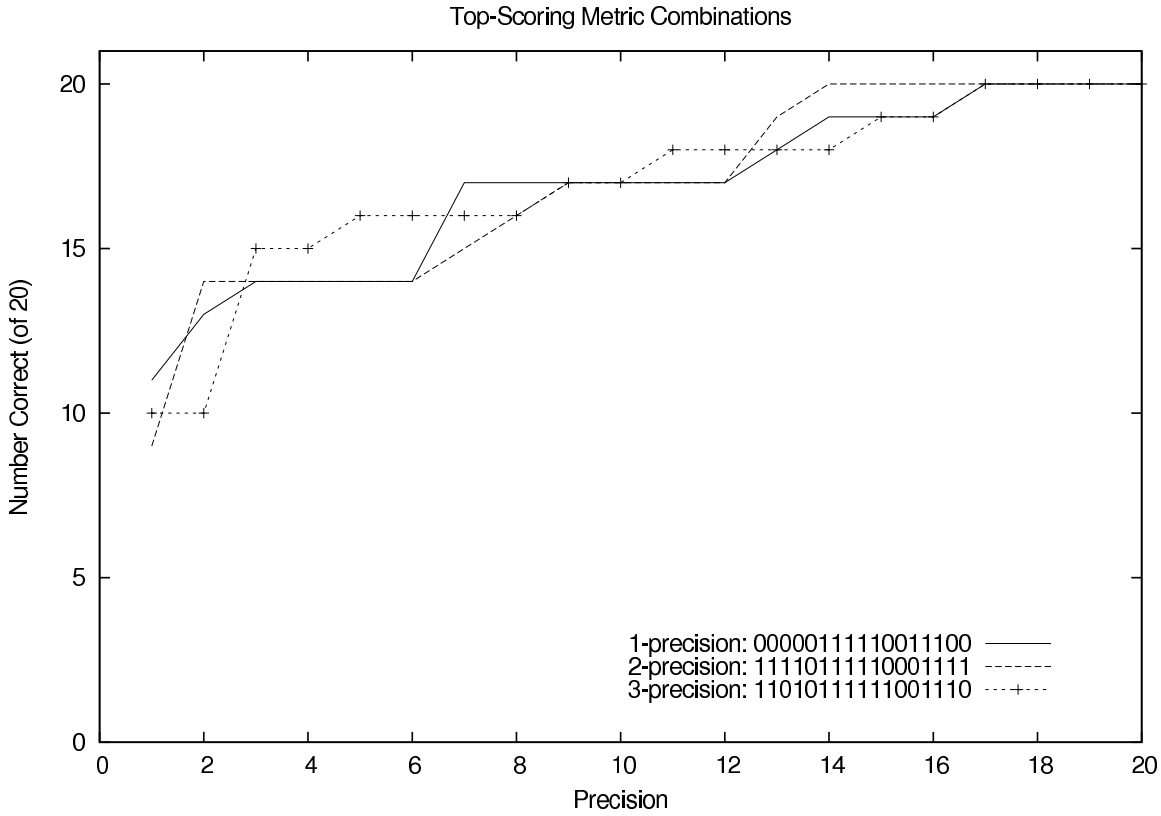

Figure 2: The top ranking metrics combinations for 1, 2, and 3-precision

8. REFERENCES

[1] http://www.sourceforge.net/.

[2] S. Berchtold, C. Böhm, D. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 78–86, 1997.

[3] R. Bouckaert. Bayesian network classifiers in weka. Technical Report 14/2004, The University of Waikato, Department of Computer Science, Hamilton, New Zealand, 2004.

[4] G. Demiroz and H. A. Guvenir. Classification by voting feature intervals. In *ECML '97: Proceedings of the 9th European Conference on Machine Learning*, pages 85–92, London, UK, 1997. Springer-Verlag.

[5] H. Ding and M. Samadzadeh. Extraction of Java program fingerprints for software authorship identification. *The Journal of Systems & Software*, 72(1):49–57, 2004.

[6] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[7] A. Gray, P. Sallis, and S. MacDonell. Identified: A dictionary-based system for extracting source code metrics for software forensics. *seep*, 00:252, 1998.

[8] J. Lenahan. Pygene Open Source Evolutionary Computation Tool. *SIGEVOlution Newsletter*, 1(2):27, 2006.

[9] S. Macdonell, A. Gray, G. MacLennan, and P. Sallis. Software forensics for discriminating between program authors usingcase-based reasoning, feedforward neural networks and multiplediscriminant analysis. *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on*, 1, 1999.

[10] P. W. Oman and C. R. Cook. Programming style authorship analysis. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 320–326, New York, NY, USA, 1989. ACM Press.

[11] J. Quinlan. *C 4. 5: Programs for Machine Learning*. Morgan Kaufmann, 1992.

[12] P. Sallis. Contemporary Computing Methods for the Authorship Characterisation Problem in Computational Linguistics. *New Zealand Journal of Computing*, 5(1):85–95, 1994.

[13] P. Sallis, S. MacDonell, G. MacLennan, A. Gray, and R. Kilgour. Identified: Software authorship analysis with case-based reasoning. *Proc. Addendum Session Int. Conf. Neural Info. Processing and Intelligent Info. Systems*, pages 53–56, 1997.

[14] E. Spafford and S. Weeber. Software forensics: Can we track code to its authors. Technical Report CSD-TR 92-010, Purdue University, Dept. of Computer Sciences, 1992.