Temporally Robust Software Features for Authorship Attribution

Steven Burrows, Alexandra L. Uitdenbogerd and Andrew Turpin
School of Computer Science and Information Technology
RMIT University
GPO Box 2476V, Melbourne 3001, Australia
{stein.burrows,alexandra.uitdenbogerd,andrew.turpin}@rmit.edu.au

Abstract

Authorship attribution is used to determine the creator of works among many candidates, playing a vital role in software forensics, authorship disputes and academic integrity investigations. The evolving coding style of individuals may degrade the performance of systems that attribute authorship of source code, and has not been previously studied. This paper uses a collection of six programming assignments with guaranteed relative timestamps from 272 students to examine evolution of coding style.

We find that the problem domain of the software developed has a large affect on the ability to attribute authorship, and that coding style does change over time regardless of the requirements that are coded. The outcomes suggest that it takes at least three programming tasks for coding style to settle, and that at least one piece of code in the same problem domain as the code to classify is necessary for accurate authorship attribution.

In the final part of the paper we analyze low level code features to discover simple features that appear immune to evolution of coding style, and use them to improve effectiveness of our system from 79% to 82% ($p < 0.01$, z-test).

Keywords: Coding style evolution, authorship attribution, adversarial information retrieval.

1 Introduction

The goal of authorship attribution is to determine the author of unknown or contentious samples of work. To do this, stylistic analyzes of earlier work samples from candidate authors are conducted to look for similarities with the disputed work. Manual analysis is impractical for large problems, and a wealth of algorithms have been implemented to automate the process including support vector machines, decision trees, neural networks and discriminant analysis. The majority of authorship attribution research has been conducted on natural language documents; for example, see Zhao and Zobel [18]; and some for source code; for example, see Burrows et al. [2, 4].

An author’s writing style evolves over time, which results in the earliest work samples becoming the least reliable indicators of current writing style. For example, Can and Patton [5] studied the changes in writing style of two Turkish authors spanning twenty-seven and fifty-six years respectively. With work samples organized into ‘old’ and ‘new’ categories, they found a statistically significant difference in the average word length between these categories and obtained 92% classification accuracy when attributing the samples to one of these categories.

No previous experiments have allowed for the temporal variation in coding style when attempting authorship attribution. Burrows et al. [4] give a detailed comparison of source code authorship attribution approaches from six other research groups, and none of the studies incorporated timestamps. Such an investigation may reveal when authorship attribution effectiveness results stabilize, indicating how long it takes coding style to settle, particularly for students.

We use a collection of 1,632 programming assignments belonging to six consecutive assessment tasks within three semesters from 272 students to investigate changing style effects on source code authorship attribution. An information retrieval approach based on the work of Burrows et al. [4] is implemented whereby the work samples are tokenized, converted into n-grams, indexed in groups of ten authors using an open source search engine, and queried against the indexes to determine the most likely author in turn. Our average classification rate is 78.52%, which is competitive with their reported results [4].

To investigate temporal effects on style, classification effectiveness for each coding task is reported individually to determine which have significantly reduced effectiveness, suggesting when coding style stabilizes. Several variations are then considered where some candidate past, current and/or future matches are omitted to simulate a real-life
scenario such as not having access to work samples from the future. The final phase takes a fine-grained look at the individual features that make the strongest contributions towards authorship attribution effectiveness, which are used in a refinement that increases effectiveness from 78.52% to 82.35%.

2 Applications of Authorship Attribution

Krsul and Spafford [10] describe four broad applications of authorship analysis techniques: authorship disputes, plagiarism detection, large software project maintenance and real-time misuse detection that we now discuss in turn.

Courts of law may be required to present evidence to resolve authorship disputes between multiple claimants of the same work. Lange and Mancoridis [11] give other legal examples such as criminal prosecution of authors of malicious code such as viruses and worms, and execution of no-competition contract clauses whereby programmers are forbidden to work for rival companies for a fixed period after the end of an employment arrangement. Expert witnesses may be required to give evidence to authorship traits and computer software may be used to expedite the collection and summary of evidence.

In plagiarism detection investigations, authorship analysis techniques can complement plagiarism detection software. Plagiarism detection software is good at identifying works with matching parts [3, 8, 13, 15], but they are not suited to discovering the true author of works. However, there is some overlap in the fields of plagiarism detection and authorship attribution where metric-based plagiarism detection systems are used [9].

Authorship attribution techniques can also be helpful in large software project maintenance for projects that have contributions by multiple authors over many years. When maintenance of a code segment is needed, the identification of the original author may be required if the code is lacking authorship documentation.

Real-time misuse detection is another authorship attribution application whereby locally compiled programs could be compared to the style of previous samples of the same author and malicious programming traits. However Krsul [10] cite two barriers towards the successful application of such systems. Firstly, interpreters, compilers, other tools and operating systems would all need to implement these metrics. Secondly, code must be compiled locally so that external systems couldn’t be used to avoid malicious action.

3 Methodology

In this section we describe the corpora of source code we used for experiments and the classification system used for authorship attribution.

3.1 Corpora

We have created a collection with guaranteed relative timestamps, which contains solutions to six C programming assignment tasks by 272 authors totaling 1,632 work samples. Each author completed the same six assignment tasks, with two assignment tasks coming from each of three large courses in our department. Each of the three courses are referred to as Semester 1, Semester 2 and Semester 3, and the six tasks are named S1A1 (Semester 1, Assignment 1), S1A2, S2A1, S2A2, S3A1 and S3A2.

Importantly, the three semesters form a prerequisite chain whereby Semester 1 is a prerequisite for Semester 2, and Semester 2 is a prerequisite for Semester 3, which guarantees relative timestamp information. The collection comprised an average of 1,107 lines of code per submission, with the average lines of code per assignment being 667, 1,681, 836, 1,415, 1,138 and 898 respectively. All assignments were made anonymous to comply with ethics requirements of our university.

3.2 Classification System

In order to study the effects of evolving coding style on authorship attribution, we implemented the system described by Burrows et al. [4]. The training data they used to tune their system parameters is very similar to our collection here, comprising C programming assignments biased towards the most prolific contributors in a university assignment submission repository spanning 1999 to 2006. Hence we adopt the same parameters that they used in their work.

The following steps give a brief overview of how an authorship attribution experiment proceeds with the system.

1. Input a set of \( m \) authors for which the experiment is to be conducted, with \( k \) pieces of work for each author.

2. Tokenize the \( m \times k \) pieces of source code into token streams using operator, keyword and white space features.

3. Convert the token streams into n-grams (\( n = 6 \)) by sliding a window of size \( n \) across the token stream one position at a time. That is, a token stream of \( t \) tokens generates \( t - n + 1 \) n-grams.

4. Index all of the n-grams generated by the \( m \times k \) token streams using the Zettair open source search engine (http://www.seg.rmit.edu.au/zettair).

5. Use the Zettair search engine to return a ranked list of streams using each of the \( m \times k \) streams as a query in turn. The similarity between the query stream and the other streams is calculated using the Okapi BM25 similarity measure [14].
6. Assign authorship to the author of the top ranked stream, ignoring the query stream in the returned list. Classifications assigned to the author of the query stream are marked as correct.

Using this methodology, the effectiveness reported is the total number of queries marked correct as a proportion of the total number of queries. In the experiments reported in this paper, we randomly selected 250 subsets of ten authors from the 272 possible as input to the process. The effectiveness reported is then the mean over these 250 runs.

4 Experiments and Results

To study the effects of course material within a semester and evolving style, we vary the pieces of work that are used for each author. As a baseline, we include all pieces of work \((k = 6)\) for all authors, as has been done in previous authorship attribution studies.

4.1 Baseline

When all pieces of work are included for each author \((k = 6)\) effectiveness is 78.52%. This compares favorably to the previous work by Burrows et al. [4], which reported an effectiveness of 76.78%.

We expect some combination course effects and temporal effects contributing towards this result. Course effects come about from students using similar features for assignment tasks dictated by the course material, which may recur for other assignments within the same course. For example, a course may comprise of material requiring the implementation of fundamental computer science data structures, and students may be asked to implement several assignments on this topic each requiring dynamic memory allocation constructs.

Temporal effects come about from the evolving and maturing coding style of individuals. For example, students may imitate coding practices given by examples in the early stages of their studies before settling on their own coding practices. A study by Anderson et al. [1] conducted on three novice programmers over the first thirty hours of learning to program concluded that example and analogy initially drove programming behavior. Therefore some of the earliest work samples may be poor indicators of current coding style.

Classification accuracy for the six tasks is plotted in Figure 1 (solid line with circles). For the S1A1 and S1A2 tasks, success rates were lowest at 63.80% and 53.84% respectively. Effectiveness peaked at 96.48% and 93.80% respectively for Semester 2 queries and then Semester 3 queries dropped off to 79.44% and 83.76%. The lower result for S1A2 may seem surprising here. If course influence is important, then S1A2 queries should match S1A1 candidates frequently, but this is difficult if S1A1 submissions demonstrate poorest programming style. Hence these initial results suggest that both course and temporal effects are apparent, which motivates the separate investigations of these effects.

4.2 Course Matching

Figure 1 presents results for two further experiments designed to explore the affect of the course material on task effectiveness. The first variation (dashed line with squares) only allows candidate matches from within the same semester as the query \((k = 2)\). For example, S2A1 queries can only be matched against S2A1 or S2A2 documents, and so forth. The second variation (dotted line with triangles) is the opposite, only allowing matches from outside the current semester \((k = 4)\).

The current semester variation is clearly more effective than the baseline that includes all pieces of work per author, indicating that the course for which code was written has a strong effect on the ability to attribute authorship. Overall classification effectiveness is 89.67% compared to 78.52% for the baseline.

The variation that excludes work from the same semester as the query is of particular interest. Firstly we note, that effectiveness for all assignments plummeted below the baseline, adding further evidence that course material has a strong influence on the ability to attribute authorship. Overall effectiveness is just 51.91%. Secondly, unlike the baseline and current-semester variants, there is an increase
in effectiveness from using S1A1 queries to S1A2. Little variation exists between the remaining queries. As course influence is removed from this experiment, we can suggest that individual programming styles improve rapidly across the first semester resulting in increasing accuracy up to the start of the second semester when results largely plateau. Therefore we speculate that programming work samples created in the first six months of learning to program are particularly unreliable as markers of programming style.

4.3 Assignment Matching

Two further variations on the baseline are given in Figure 2. Firstly, classification results only allowing current assignment and past match candidates are considered (dashed line with squares) causing overall effectiveness to drop from 78.52% to just 52.83%. This variation simulates the scenario of not having access to documents created after the query. In particular, S1A1 queries cannot be attributed at all (effectiveness would be 16.67% higher if these were included), and S1A2 effectiveness dropped from 53.84% to 38.12% as these queries had to exclusively rely on the very first assignment, which is often unreliable as discussed in Section 4.2.

For both Semester 2 and Semester 3, the effectiveness in assigning authorship to the second assignment (S2A2 and S3A2) was markedly higher than for the first assignment. For S2A1 and S3A1 queries, effectiveness dropped by 42.16% and 32.52% relative to the baseline, however, there was only one less success for S2A2, and the results remained unchanged for S3A2. These reinforce the observations from the previous section that having samples of work from the same course/semester dramatically improves the ability to attribute authorship.

The opposite variation is provided in Figure 2 for comparison, and only allows current assignment and future candidate matching. As might be expected, the see-saw pattern (dotted line with triangles) is reversed, as queries from the first assignment in each semester have the second assignment available for matching, while the second assignment does not have the first available. Hence, the effectiveness for assignment one in each semester is higher than the effectiveness in semester two, which demonstrates course effects.

The average classification score is 55.81%, which is higher than its counterpart (52.83%) but still far behind the baseline (78.52%). This provides us with evidence for temporal effects: as coding style stabilizes over time, matching against future samples of work is more effective than matching against past samples.

5 Individual Features

Given that the previous section shows strong evidence for both course and temporal effects in the evolution of coding style, it would seem that an authorship attribution system should try and mitigate these effects. In this section we investigate the individual features of the source code in our collection: the tokens prior to the formation of n-grams and indexing in the Burrows method [4] we employed. Particularly, we aim to identify tokens that are unique to students, but not unique to courses as possible markers of individual style.

We measure between-assignment and between-student dispersion of a token as the entropy [16] of the token’s probability distribution over either assignments or students respectively. For example, if a token occurs \( f_i \) times in all source code for the \( i \)th assignment, and \( F \) times in the collection, then the entropy of that distribution is

\[
- \sum_{i=1}^{6} f_i / F \log_2(f_i / F).
\]

Entropy is often thought of as a measure of information content. Low entropy indicates that there is little information; that is, the feature is likely to occur in very few of the assignments/student’s code. In the extreme case of zero entropy then the feature only occurs in one assignment/student’s code: it is entirely predictable where the feature occurs. The other extreme is where all possible occurrences are equally likely. For our situation, therefore, we want high entropy of between-assignment occurrences (indicating that the use of the feature does not vary much within courses) and low entropy for between-student occurrences (showing that the feature is peculiar to some students) for the feature to be a good candidate for authorship attribution.
Table 1. Entropy of feature distribution for the six assignments (En–A) and the 272 students (En–S) for all features averaging at least fifty instances per assignment. Lower scores indicate larger variation within groups. Bold cases are discussed in the text.

<table>
<thead>
<tr>
<th>Token</th>
<th>Count</th>
<th>En–A</th>
<th>En–S</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot; SPACE &quot;</td>
<td>10232897</td>
<td>2.49</td>
<td>7.93</td>
</tr>
<tr>
<td>&quot; \n &quot;</td>
<td>1808827</td>
<td>2.52</td>
<td>8.01</td>
</tr>
<tr>
<td>Lowercase literal</td>
<td>938219</td>
<td>2.50</td>
<td>7.99</td>
</tr>
<tr>
<td>&quot; ( &quot;</td>
<td>622548</td>
<td>2.51</td>
<td>8.01</td>
</tr>
<tr>
<td>Camelcase literal</td>
<td>504263</td>
<td>2.48</td>
<td>7.78</td>
</tr>
<tr>
<td>&quot; \t &quot;</td>
<td>443138</td>
<td>2.53</td>
<td><strong>6.71</strong></td>
</tr>
<tr>
<td>&quot; \n &quot;</td>
<td>335086</td>
<td>2.46</td>
<td>7.97</td>
</tr>
<tr>
<td>&quot; \w &quot;</td>
<td>270022</td>
<td>2.52</td>
<td>7.85</td>
</tr>
<tr>
<td>Other literal</td>
<td>236174</td>
<td>2.52</td>
<td>7.81</td>
</tr>
<tr>
<td>&quot; -&gt; &quot;</td>
<td>166478</td>
<td><strong>2.02</strong></td>
<td>7.96</td>
</tr>
<tr>
<td>&quot; [ &quot;</td>
<td>158671</td>
<td><strong>2.31</strong></td>
<td><strong>7.10</strong></td>
</tr>
<tr>
<td>&quot; \w &quot;</td>
<td>144188</td>
<td>2.43</td>
<td>7.89</td>
</tr>
<tr>
<td>Titlecase literal</td>
<td>118777</td>
<td><strong>2.20</strong></td>
<td>7.86</td>
</tr>
<tr>
<td>Uppercase literal</td>
<td>114434</td>
<td>2.42</td>
<td>7.68</td>
</tr>
<tr>
<td>if</td>
<td>109837</td>
<td>2.47</td>
<td>8.00</td>
</tr>
<tr>
<td>int</td>
<td>108436</td>
<td>2.52</td>
<td>7.93</td>
</tr>
<tr>
<td>&quot; \n &quot;</td>
<td>102873</td>
<td>2.50</td>
<td><strong>5.92</strong></td>
</tr>
<tr>
<td>&quot; . &quot;</td>
<td>84297</td>
<td>2.44</td>
<td>7.70</td>
</tr>
</tbody>
</table>

Maximum Possible 2.58 8.09

For our collection, the maximum possible entropy values are $-\log_2 \frac{1}{6} = 2.58$ between assignments, and $-\log_2 \frac{1}{272} = 8.09$ between students. Table 1 shows the entropy scores for all features that have at least fifty instances per program on average. The three cases with lowest between-assignment and between-student entropy values marked in bold are now discussed.

Tabs (\t) and carriage returns (\n) have a high between-assignment entropy combined with a low between student entropy, indicating their potential for predicting program authorship. Conversely, the ‘->’ and Titlecase literal symbols have low between-assignment entropy and a relatively high between-student entropy, making them a better predictor of course than author. Concerning the square-bracket token, the low En–S value occurs with a low En-A value, indicating that it is unlikely to be of use for authorship attribution. Upon further investigation, we located a single outlier file that contained 29,232 opening square bracket tokens out of 102,318 tokens total.

In Figure 3, an example is provided showing how contrasting between-student entropy scores came about for the carriage return and parenthesis tokens. The x-axis represents all 272 students bucketed into thirteen groups and sorted for clarity. The parenthesis tokens demonstrate a normal distribution with no zero-score or extremely high outliers. The carriage return tokens demonstrate a skewed distribution with many zero-score values and a much higher range. Features with a low entropy like this demonstrate stronger potential as authorship attribution markers.

We believe indentation should be one of the strongest markers of authorship but this isn’t obvious from the results in Table 1 for the ‘SPACE’ token because we would expect the between-student entropy score to be closer to that of the tab (\t) token given that both tokens are used for indentation. From these results, we realized that we did not distinguish between spaces used for indentation (usually many spaces at once) and spaces used to separate key operators and operands (usually a single space) in our tokenization of the input source code. Many 6-grams comprised entirely of spaces are expected for indentation, and there is no mechanism to detect deeper indentation instances. Therefore we explore a refinement to improve the use of white space next.

Figure 3. Comparison of the use of parenthesis and carriage return features over all students (bucketed into thirteen groups). The carriage return has the lowest entropy.
Table 2. Entropy of feature distribution for the six assignments (En–A) and the 272 students (En–S) for white space features with at least ten instances per work sample. Lower scores indicate larger variation within groups. Bold cases are discussed in the text.

<table>
<thead>
<tr>
<th>Token</th>
<th>Count</th>
<th>En–A</th>
<th>En–S</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;SPACE01&quot;</td>
<td>1675961</td>
<td>2.51</td>
<td>7.98</td>
</tr>
<tr>
<td>&quot;SPACE03&quot;</td>
<td>366565</td>
<td>2.50</td>
<td>7.89</td>
</tr>
<tr>
<td>&quot;SPACE06&quot;</td>
<td>229320</td>
<td>2.48</td>
<td>7.88</td>
</tr>
<tr>
<td>&quot;SPACE09&quot;</td>
<td>131577</td>
<td>2.46</td>
<td>7.76</td>
</tr>
<tr>
<td>&quot;SPACE12&quot;</td>
<td>88908</td>
<td>2.47</td>
<td>7.71</td>
</tr>
<tr>
<td>&quot;SPACE02&quot;</td>
<td>68845</td>
<td>2.42</td>
<td>6.03</td>
</tr>
<tr>
<td>&quot;SPACE04&quot;</td>
<td>64918</td>
<td>2.53</td>
<td>7.05</td>
</tr>
<tr>
<td>&quot;SPACE15&quot;</td>
<td>47415</td>
<td>2.46</td>
<td>7.48</td>
</tr>
<tr>
<td>&quot;SPACE18&quot;</td>
<td>25261</td>
<td>2.44</td>
<td>7.28</td>
</tr>
<tr>
<td>&quot;SPACE08&quot;</td>
<td>24078</td>
<td>2.53</td>
<td>6.65</td>
</tr>
<tr>
<td>&quot;SPACE07&quot;</td>
<td>22958</td>
<td>2.49</td>
<td>6.91</td>
</tr>
<tr>
<td>&quot;SPACE09&quot;</td>
<td>17344</td>
<td>2.49</td>
<td>6.74</td>
</tr>
</tbody>
</table>

Maximum 2.58 8.09

5.1 Improving Effectiveness

Up to this point, white space has been managed by a single feature that needs improvement for capturing indentation information as discussed in the previous section. We now consider forty white space tokens representing contiguous sequences of one to forty white spaces to better represent even the most deeply indented programs. Similar to Table 1, in Table 2 we show how the use of these features varies between assignments and students for all white space features with at least ten instances per program average.

The ‘SPACE01’ token is still most prevalent but has now been reduced to 16.38% of its prior volume. Space tokens in multiples of three come next (‘SPACE03’, ‘SPACE06’, ‘SPACE09’, ‘SPACE12’) indicating the strong preference for code blocks to be indented in multiples of three spaces. Then ‘SPACE15’ and ‘SPACE18’ tokens have the next highest between-student entropy scores. Of most interest is the drop in entropy scores for the remaining five white space features (‘SPACE02’, ‘SPACE04’, ‘SPACE08’, ‘SPACE07’ and ‘SPACE05’) indicating wider spread of scores. Of this group, ‘SPACE04’ and ‘SPACE08’ have the highest En-A scores, indicating that they may be good authorship markers. Just as for the ‘<’ token in the previous table, the low En-S and low En-A score for ‘SPACE02’ is partly due to a very unusual submission.

Using the new white space features, we repeated our 10-class authorship attribution experiment and found that effectiveness jumped from 78.52% (11,700/15,000) to 82.35% (12,352/15,000) with this feature change alone. This difference is statistically significant at the 99% confidence interval using a z-test for two proportions. Further feature refinements like this may improve results again, but we would expect these improvements to be more subtle as the change made with white space affected the feature represented by the largest number of tokens in our collection. Another sensible refinement could be to interpret literals differently to identify authors who use short or long identifier names, but we leave these refinements for future work.

6 Discussion

In this paper, we have adopted the information retrieval approach to authorship attribution that is recommended by Burrows et al. [4]. This approach tokenizes code into overlapping 6-grams, and then uses a TF×IDF approach to rank the query code against all other pieces of code [14]. That is, based on the frequency of the 6-grams within the query and target document (TF) and the frequency of the 6-grams in the entire collection (IDF). Burrows et al. appear to have developed this method without any regard for the evolution of coding style that an author undergoes, tuning their features and parameters on a collection that did not have information about when pieces of code were written relative to another.

Using the Burrows approach [4], we used a collection of 1,632 student programming assignments with guaranteed relative timestamps for our source code authorship attribution work. We reported effectiveness of the approach for each of six assignment tasks and found that results for the first two assignments students undertake were poor, suggesting that it takes at least one semester for students to develop some maturity in their programming style.

Variations of the above experiment reported results with candidate document matches from earlier, current and/or later tasks excluded. This allowed us to study course and temporal effects separately. Also, excluding later results allowed us to simulate a real-life scenario of these work samples not being yet available. We found that removing all futuristic matches greatly decreased the effectiveness of our method, but allowing matches from the current semester to remain was a good compromise with effectiveness only dropping by 3.21%. The value of having multiple work samples per course and/or author is apparent given the strong course effects shown.

Section 5 demonstrates that when timestamps are available, features that are robust to changing coding style can be found. In this paper we only investigated individual tokens, but there is scope to extend the approach to n-grams. Machine learning algorithms could also be used for the matching process.

Entropy was used to investigate how the use of low level features varies between assignments and students to identify
such robust features. Space features were thought to be a strong marker for authorship as they determine the indentation style, but we concluded that the weight added towards making authorship decisions was unsatisfactory. Making a substitution of this feature for consecutive blocks of spaces better captured indentation information and single-handedly resulted in a statistically significant effectiveness boost of 3.83%.

Given the strong course effects found in this study, it is not until the second task within a course that source code authorship attribution reaches a level that it would be practically useful. This implies that software engineers require at least one prior task in a domain before their coding style stabilizes. This has implications for those developing coding standards, and monitoring compliance within organizations.

It might be thought that this work would fail if all work samples adhered to consistent corporate or public domain coding standards making it impossible to distinguish author style. Example standards include language standards such as Sun’s Java coding standards [17], standards for integrated development environments such as Microsoft’s Visual Studio [12], company specific standards such as Geosoft’s C++ standards [7], and general guidelines for the wider community such as the infamous Ten Commandments for C Programmers [6]. These standards give guidelines for aspects of programming style such as comments, layout, naming conventions, source file organization and general readability. However, despite rigorous rules like these there is still sufficient room to express individual coding style. For example, short hand notations (a = a + b can be written as a += b), alternate programming constructs (for, while, and do-while loops; and if/else versus switch/case branching constructs) and choice of standard library routines (such as input/output function options) discern some preferences. Our method still remains effective with these types of alterations as each only creates localized changes in overlapping n-grams, which don’t adversely affect entire programs. For example, a single token substitution in a 6-gram program representation would change no more than six 6-grams.

Although this study was on a combined undergraduate and postgraduate student cohort, computing is a very dynamic discipline. Throughout the working lives of software engineers, they could be expected to learn several new languages, and so are in similar situations to the students. Hence, we believe our results will carry over to industry.

7 Conclusions

This paper represents an important advance in source code authorship attribution as previous studies have not allowed for the evolving coding style of programmers over time, nor the effect that the problem domain of the requirements might have on the evolution of coding style. We have demonstrated that both effects exist in our data. In particular:

- authorship can be attributed much more effectively if at least one other coding sample is available from the same problem domain as the queried work for all authors; and
- coding style does not stabilize until after the third programming task of an author.

These results have implications for educators, and software engineers that manage and undertake the coding process.

By using the entropy of the distribution of code features over time and authors, we located some features that mitigated against the evolution of coding style when attributing authorship. Accordingly, these features significantly improved the effectiveness of our authorship attribution system.

References


