Authorship Identification and Verification of JavaScript Source Code

An Evaluation of Techniques

Wilco Wisse
Authorship Identification and Verification of JavaScript Source Code

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Wilco Wisse
born in Borsele, the Netherlands
Authorship Identification and Verification of JavaScript Source Code

Author: Wilco Wisse
Student id: 4034171
Email: w.c.wisse@student.tudelft.nl

Abstract

The increasing number of criminals that exploit the speed and anonymity of the Web has become of increasing concern. Little effort has been spent to trace the authors of malicious code. To that end we investigated authorship identification and verification of JavaScript source code. We evaluated three character based approaches and propose a new domain specific approach. What is new in the domain specific analysis approach, is that it represents code by a parse tree to extract structural features. The evaluation of the techniques with open source code from GitHub, turned out that the approaches that use character $n$-gram features achieved the best performance. However, the combination of $n$-gram and domain specific features turned out to be complementary, resulting in a higher performance. Techniques that used similarity based classification were especially successful if a limited amount of training data were available, while feature vector based techniques were mainly successful when a large amount of training data were available and in an authorship verification context. By means of code minification we evaluated how the classification accuracy is affected by removing authorship information from the source code. Code minification has shown to significantly deteriorate the performance of the authorship analysis methods. Especially the compression based technique is robust against code minification.

Thesis Committee:
Chair: Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor: Dr. C.J. Veenman, Netherlands Forensic Institute
Committee Member: Dr. A. Bacchelli, Faculty EEMCS, TU Delft
Committee Member: Dr. C. Hauff, Faculty EEMCS, TU Delft
Preface

The master thesis that is lying in front of you is the result of almost one year research at the NFI. Although this writing is only the last stage of a journey, the work as a whole could be expressed by the words of novel writer E.L. Doctorow: “Writing is like driving at night in the fog. You can only see as far as your headlights, but you can make the whole trip that way.” I am using this opportunity to express my gratitude to everyone who helped and supported me to keep the show on the road. First, this thesis would not happen to be possible without the help of Andy Zaidman. I appreciated his useful feedback and the enthusiasm and patience in which he supported me throughout my thesis. It is also with immense gratitude that I acknowledge the help of Cor Veenman. The valuable brainstorm sessions provided me many new insights and I liked the high level of freedom he gave me during the project. Besides Andy and Cor, I am grateful to the other committee members for reviewing my work. I would also like to thank my colleagues at Kecida. It was a real pleasure to do my research in such an interesting team. In particular, I would like to thank Gert Jacobusse for offering suggestions on the thesis topic and his ideas on the work in progress. Last but not the least, I would like to thank my family. None of this would have been possible without their love, encouragement and prayer.

Wilco Wisse
The Hague, the Netherlands
December 11, 2014
## Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>1 Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Authorship analysis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Research objectives</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Thesis organization</td>
<td>6</td>
</tr>
<tr>
<td>2 Preliminaries and Related Work</td>
<td>7</td>
</tr>
<tr>
<td>2.1 The authorship analysis process</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Source code authorship identification</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Conclusion</td>
<td>21</td>
</tr>
<tr>
<td>3 Approach</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Character based approaches</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Domain specific approach</td>
<td>30</td>
</tr>
<tr>
<td>3.3 Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>4 Experimental Setup</td>
<td>39</td>
</tr>
<tr>
<td>4.1 Validation procedure</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Source code collections</td>
<td>41</td>
</tr>
<tr>
<td>5 Results</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Authorship identification</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Authorship verification</td>
<td>56</td>
</tr>
<tr>
<td>6 Discussion</td>
<td>59</td>
</tr>
<tr>
<td>6.1 Comparing classification techniques</td>
<td>59</td>
</tr>
<tr>
<td>6.2 Comparing the effectivity of features types</td>
<td>60</td>
</tr>
<tr>
<td>6.3 Applying authorship verification</td>
<td>61</td>
</tr>
</tbody>
</table>
The cover of this work shows one of the oldest known sentences in western old Dutch: “Hebban olla vogala nestas hagunnan hinase hic anda thu uuat unbidan vue nu.” Probably the fragment dates to 11th century. Although the author of the fragment is unknown, the writing style still reveals some clues of authorship, such as the origin of the writer. Recent research concluded that the fragment was probably written by a woman or from a female perspective. Clearly, the writing style of a text conveys much information of the respective author. Similar to texts, a ‘fingerprint’ of writing style can be extracted from program source code. In this thesis the focus is on authorship analysis of JavaScript source code.

1.1 Background

The activity of authorship analysis can at least be dated back to the middle ages when the scholasticism dominated the academics in Europe. In that time, knowing the authorship of a text was essential to determine the text its veracity [48]. A typical example of authorship analysis is the investigation whether all works of Shakespeare have correctly been claimed to be written by himself, or that there is evidence that William Shakespeare is the same person as William Shakspere [23]. Others tried to use authorship identification techniques to determine the author of the letter to the Hebrews by comparing different texts of the New Testament [69].

The basis of authorship analysis is that language is flexible enough in format to allow humans to create certain habits in how the language is used. If texts are handwritten, the analysis of physical characteristics such as the shape of dots and regularity provide clues of authorship [72]. Also printed texts contain much authorship information, such as the frequency of individual words, the preference to use short or long sentences and punctuation style. Programming languages share many linguistic concepts with natural languages [81]. Both can be expressed by textual symbols that have semantics. Also, both are systematically constructed according to a grammar that describes the formation
and composition of words. Although the syntax definition of programming languages is more strict than natural languages, there is still a certain amount of flexibility that enables programmers to develop their own programming style [33, 19].

In general, authorship analysis has numerous applications. It may be used in cases of academic dishonesty and to resolve issues in litigation and theft. The focus on authorship analysis on JavaScript source code is mainly of interest due to the rapid growth in information technology and the increasing popularity of the Internet. The Web is relatively unprotected and an increasing number of criminal activities are exploiting its convenience and anonymity. Much effort has been spent to detect malicious content. However, less attention has been paid to identification of the respective authors. JavaScript is commonly used on webpages. The identification of the developer of fragment of JavaScript code forms a supplementary source of authorship information that can play a crucial role in a forensic investigation, such as tracing the authors of fraudulent websites. As such, source code authorship analysis is part of the field of software forensics, which comprises a multitude of program analysis tasks such as malicious code analysis and source code plagiarism detection [72].

1.2 Authorship analysis

We first introduce the authorship analysis problem. In general, the term authorship analysis embodies a multitude of methods for examining the characteristics of a document to draw conclusions on its authorship [84]. This can be divided in at least three subproblems, namely authorship identification, authorship characterization and authorship verification. Because a generally accepted definition is lacking and these terms are used inconsistently in literature, we first define clearly how we will use these terms.

1.2.1 Authorship identification

The aim of authorship identification is to determine the most likely author of a document with unknown authorship by examining other writings of the candidate authors [84]. In this work we define the term authorship identification as follows:

**Definition 1 (Authorship identification).** Given a set of authors $A = \{a_1, a_2, \ldots, a_n\}$, a set of training documents $D = \{d_1, d_2, \ldots, d_m\}$, and a surjective label function $l : D \rightarrow A$ which defines the authorship of each training document. The authorship identification problem is then to predict $\text{author}(d)$, the author of an unlabeled document $d$.

A distinction can be made between closed and the open set authorship identification [83]. In the closed set problem, the author of document $d$ is a priori known to be one of the authors in $A$. In contrast to the closed set problem, the open set problem does not assume that $d$ necessarily has been written by one of the candidate authors. It can thus be written by a previously unseen author for which no training documents have been provided.
1.2. Authorship analysis

![Diagram](image)

**Figure 1.1:** The difference between (a) closed set authorship identification and (b) authorship verification. In this example, \(a_i\) are authors and \(d\) is an unlabeled document.

**Definition 2** (Open vs. closed set authorship identification). The authorship identification problem is a closed set problem if and only if it is given that \(\text{author}(d) \in A\).

Plagiarism detection is not the same as authorship identification, but is related. Plagiarism detection is concerned with the detection of replication of the work from another author’s work in a document, and hence is concerned with authorship analysis within a document. Similarly, source code authorship identification is different from identifying code clones in software.

### 1.2.2 Authorship verification

Authorship verification (also known as authorship discrimination or similarity detection) involves determining whether two documents have been written by the same author, without necessarily identifying the author. In other words, for two documents \(d_1\) and \(d_2\) the problem is to decide whether \(\text{author}(d_1) = \text{author}(d_2)\) [6, 84]. Basically, this is a special case of the open set authorship identification problem with only one author and with only one available training document. Others however also used the name verification problem for the one author open set problem with multiple training documents available for the target author [48, 83]. Figure 1.1 conceptually shows the authorship identification and authorship verification problem. The main difference between verification and closed set identification is that verification is a ‘one versus the rest of the world’ problem, so there is no candidate author set established in advance, as is the case in identification. We define the authorship verification problem as follows:

**Definition 3** (Authorship verification). The authorship verification problem is an open set authorship identification problem with one candidate author, i.e. \(|A| = 1\).

### 1.2.3 Authorship characterization

Another field of authorship analysis is authorship characterization [84], which has been called authorship profiling in literature as well [48]. This problem addresses the question how characteristics of the author of a document can be identified, for example the gender, the educational background and language familiarity. This field does not aim to answer
the question which person wrote a piece of writing, but to predict personal characteristics of the author.

1.3 Research objectives

Authorship identification has been applied on several programming languages, including Pascal [64], C [53], Java [55, 70, 19] and C++ [45, 57]. In this work the focus is in particular on authorship analysis of JavaScript source code. JavaScript is an interpreted object-oriented scripting language which shares many syntactic concepts from C and Java. The investigation of authorship analysis of JavaScript code is mainly of significance because of its wide availability in its source code form. First, JavaScript is an interpreted language, which means that the source code is executed directly, without compiling the program into machine-language instructions. Secondly, JavaScript is very commonly embedded within webpages and runs client-side. Overall, the source code is widely available on the Web so that authorship analysis of JavaScript can be crucial in cybercrime investigations, such as tracing the authors of malicious websites. This brings us to our main research question:

Main Research Question. To what extent are authorship analysis techniques effective on JavaScript source code?

Authorship analysis is a broad term. Therefore, we narrowed the main research question down to a number of more specific questions. These are detailed in the remainder of this section.

1.3.1 Reviewing the literature

Source code authorship analysis has not been investigated quite extensively. Nevertheless, a number of authors have addressed this problem. An overview of existing research in the form of a literature research should give an understanding of state of the art software analysis techniques. The results of the literature study also form the basis for the remaining research questions. The objective of the literature study is formulated in the first sub-question:

Research Question 1. What is the state of the art in software authorship analysis?

1.3.2 Comparing authorship analysis techniques

Authorship analysis is a special case of the categorization of text documents into a predefined number of classes, and is thus a text classification problem. Broadly speaking, authorship analysis techniques can be divided into similarity based approaches where a similarity metric is employed for classification, and feature vector based approaches that employ machine learning algorithms for classification. The different techniques are evaluated on various problem complexities by regulating the number of candidate authors and the amount of available training data per author. In our study we evaluate
1.3. Research objectives

one similarity based approach and three feature based approaches that use a Support Vector Machine (SVM) as classification algorithm. This is summarized in the second sub-question:

Research Question 2. What is the difference in performance between similarity based and feature vector based techniques, by considering the number of candidate authors and the number of available training samples per author?

The characteristics that are used in the analysis process are known as features. Next to the used classification technique, a fundamental difference between analysis techniques is the syntactical complexity of the language the features are extracted from. Some approaches extract features from a character based level and are thus language independent. Other approaches, however, use a higher level language representation to analyze the documents. Since high level features are specific to a certain language, we refer to these methods as domain specific techniques. In this work we propose a new domain specific approach dedicated to JavaScript and compare it with three existing (language independent) character based approaches from literature. New to the domain specific approach is that we consider structural parse tree features. In natural language, syntactical patterns in the parse tree have found to be strong markers of writing style. In contrast to natural languages, programming languages can be parsed unambiguously. Nevertheless, we have not found techniques that mined syntactical features this way. This brings us to the next sub-question:

Research Question 3. What is the effectivity of domain specific JavaScript features compared to character based features?

1.3.3 Applying authorship verification

We will investigate both the authorship identification and verification problem. Several studies in literature considered authorship identification of source code, but no studies have been found that addressed authorship verification. This study aims to contribute to the field by embedding the implemented authorship analysis techniques in an authorship verification context.

Research Question 4. What is the effectivity of authorship analysis techniques in authorship verification compared to authorship identification?

1.3.4 Addressing authorship analysis complications

Existing approaches report promising results, but various assumptions are done to guarantee the feasibility of the approach. For instance, most studies assume the code under investigation was written by a single author, that the coding style is not influenced by external factors and that the coding style of the author is consistent over time and across projects. A question that needs to be asked, if whether such assumptions are realistic. There are various complications which may deteriorate authorship analysis techniques,
such as the reuse of code and external constraints placed by managers or tools [53, 55]. As a result the analysis techniques may not generalize well to code that is more likely to be found in a forensic investigation or in industrial practice. It is far beyond the scope of this work to address all the situations which may negatively influence the effectivity of authorship analysis. Therefore we only address two problems, which are related to the external impact on coding style. These are addressed in the following research question:

Research Question 5. What is the influence of code collaboration and code minification on the classification performance?

Code minification is the process of removing all redundant characters from source code, for instance by stripping the layout and shortening variable names. This is usually done for data compression purposes. Next, we investigate the ability to identify authors when they collaborated in the same project. Obviously, this may be problematic as they influence each other, for instance by using the same variables, objects and other constructs.

1.4 Thesis organization

This thesis has been organised in the following way. We begin in Chapter 2 by a literature review that should give an understanding of state of the art authorship analysis techniques. The results of the literature study will form the basis for the remaining chapters. In Chapter 3, the authorship analysis techniques are introduced that will be investigated. The experimental setup to evaluate the techniques and the results are described in Chapters 4 and 5, respectively. Subsequently, the main findings and limitations are discussed in Chapter 6. Finally, we conclude and give recommendations for further research in Chapter 7.
In this chapter we aim to establish a scientific background of source code authorship analysis. We start by the general process of authorship analysis. The second part moves on to authorship identification of source code and we investigate how this task has been approached in literature. Because many concepts discussed in this chapter are not limited to software but also apply to texts, we use the word document to refer to a text in natural language as well as a piece of source code.

2.1 The authorship analysis process

In general, authorship identification involves the categorization of documents into a predefined number of classes, and is thus a classification problem. Figure 2.1 depicts some important aspects of most authorship identification studies [79]. In order to be able to classify unlabeled documents, the documents in the collection are uniquely identified by performing a number of measurements on them, which are known as features. In general, if documents are represented by $n$ different features, they form a feature vector $\mathbf{x}_i \in \mathbb{R}^n$ and identify a document $d_i$ in an $n$ dimensional feature space. The role of the classifier is then to divide the feature space of the training samples into regions that correspond to the different classes, which is known as training the classifier. If a feature vector of an unknown document falls in the region of one of the classes it is classified to that class. This prediction can be either correct or incorrect. In the latter case a misclassification has occurred which will deteriorate the performance of the classifier. In an experimental setting, a part of the documents from the document collections are hold out as test set to be able to compare the predictions of the classifier with the actual class labels. A resulting performance measure $p_C$ gives an indication of the performance of the classifier on the used dataset. Overall, there are large differences in the choice of features extracted from the document, the classification model used for classification and the performance evaluation of the model. These aspects will briefly be discussed in the remainder of this section. Section 2.1.1 addresses different choices for feature
2. Preliminaries and Related Work

2.1 Feature extraction

In the feature extraction phase, a feature set is composed. A feature set is a set of predefined characteristics of the document which should be able to characterize the particular writing style of an author. Such features are referred to as stylometric features, style markers, or style features in literature. In authorship identification of natural language, Stamatatos distinguished five feature types based on the syntactic complexity of the language representation [73]. We briefly review the taxonomy in Table 2.1. In Section 2.2 we will see that feature extracted from source code can be organized in a similar manner. We now pay some attention to two feature types that are relevant for the remainder of this thesis, namely character n-gram features and syntactical features.

Character n-gram features

The application of character n-grams has proven to be quite successful in both authorship identification of texts and of source code [9]. In general, an n-gram is a consecutive sequence consisting of units of length n. By using a sliding window of length n, a document of t units will generate \( t - n + 1 \) n-grams. For instance, the code fragment ‘function(){}’ has 12 characters and generates the following set of \( t - n + 1 = 5 \) character 8-grams: function, unction(), nction(), ction(){, tion(){}. Although a large number of n-grams in a large document will be redundant, many of the n-grams will implicitly capture stylistic information. For instance, the frequency of the n-gram ‘function’ may determine the number of function declarations per lines of code, while ‘unction’ also captures that no space is used between the identifier and the parenthesis. The n-gram ‘ction{}’ further indicates that zero parameters have been defined in this particular function definition. A risk of character n-grams is that they may also capture contextual information that is closely associated with the particular domain. Thus if the training documents are not representative for the domain, the classifier may become overtrained [48]. In general, the features should be as independent as possible with regard to the context of the training samples to avoid overtraining of the classifier.
2.1. The authorship analysis process

Syntactical features

In source code, the structure of a program can conveniently be expressed by the Abstract Syntax Tree (AST), which is the result of parsing the code. Although programming languages can be parsed unambiguously (in contrast to natural languages), we have found no techniques that mined structures from the AST. In natural language, syntactical features have been proven to be good features for authorship identification [46]. A wide range of basic syntactical features have been used, of which most notable function words and Part of Speech (POS) tags [59]. POS tags are grammatical word and sentence categories such as nouns, noun phrases and verb phrases. However, since the performance of natural language processing tools greatly improved, syntactic trees can be generated with high accuracy nowadays. Syntactic trees show how the POS tags are structured by a rooted and ordered tree. Baayen [3] was the first who used rewrite-rule frequencies extracted from the parse tree. Rewrite rules represent the combinations of a node and its immediate constituents in the tree. A rewrite rule such as $A \rightarrow B, C$ results in a parent node $A$ with children $B$ and $C$ somewhere in the syntactic tree. Related to this is the frequency of every pair of tags $(A, B)$ in the tree [63] and to consider the depth of the syntax trees in the document [41]. There are also a number of works that considered more complex syntactic structures in the parse trees, such as feature sets of

<table>
<thead>
<tr>
<th>Text representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A very simple text representation is to consider it as a sequence of characters. A basic example of a measure based on this representation is the frequency distributions of characters. A character $n$-gram representation is a very well-known character based representation. Also data compression based methods are character based. The intuition behind compression based techniques is that a unlabeled document compresses the best with the training documents of its author because they have a high degree of similar patterns.</td>
</tr>
<tr>
<td>Lexical</td>
<td>Syntactic features provide code measures of a higher level of abstraction than lexical features. Syntactic features have the advantage that they are less topic dependent than lexical features and are related to how an author syntactically organizes his sentences. [38]. The most common syntactical features are function words, grammatical elements such as Part of Speech (POS) tags and rewrite rules frequencies in the grammar [46].</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Semantic analysis of a text can provide clues of authorship as well. An example is the degree of usage of synonyms.</td>
</tr>
<tr>
<td>Semantic</td>
<td>Domain specific features are not related to a particular text representation, but use features typical for a specific text domain. Examples are the analysis of greetings in emails or the layout in WYSIWYG’ed documents [38].</td>
</tr>
</tbody>
</table>

Table 2.1: Feature extraction in authorship identification based on different text representations abstraction levels.
2. Preliminaries and Related Work

tree fragments [46], closed and maximal frequent subtrees [56] and skeleton structure representations that only contain the structure of the subtrees without the syntactic categories [59]. Additionally, one approach in literature expressed how the elements appear in syntactic trees by extracting \(n\)-grams within the tree structure [80]. Others use syntactic relations in the parse tree to extract \(n\)-grams from the original text, by taking the order into account the elements appear in the parse tree [71].

Feature selection

Feature selection methods show that a subset of features may lead to significant improvements in classification performance in comparison to using the full feature set [61]. The goal of feature selection is to find the features that make a significant contribution and to eliminate features which only add noise in the classification process. Feature selection is not a trivial process however. A feature may seem to be irrelevant when examined individually, but in combination with other features it may be of interest.

2.1.2 Classification

Having considered feature extraction, we now look at the classification process. A classifier predicts the authorship of previously unseen documents based on the extracted features from the training documents. The purpose of training is to capture the various regularities from the training data as well as possible. There is however often a tradeoff between the bias and the variance of a classifier, which both contribute to the prediction error [30]. With a low bias and a high variance the learning algorithms will discover the regularities of the particular training set very well and as a consequence performance of the classifier on the training set is high. However, with a high variance the learner may be sensitive to idiosyncrasies of the training set and does not generalize well to previously unseen data. In such cases the learner is overtrained (also called overfit). Clearly, there is a tension between minimizing both the the bias and the variance of a learner. We now discuss some classification approaches that have commonly been used in authorship analysis studies.

Profile based versus instance based methods

Stamatatos distinguished two frequently used classification approaches in authorship identification, which are referred to as profile based\(^1\) and instance based methods [73]. Profile based and instance based methods differ in whether the training documents are treated individually (per document), or cumulatively (per author). This is explained in Figure 2.2. The key distinction between the methods is how the training data is sampled: as a general style for each author or a separate style for each document. In the profile based approaches all available training documents are concatenated per author in one big file and the features of this cumulative file forms the profile that characterizes

\(^1\)This term should not be confused with authorship characterization which is sometimes called profiling.
2.1. The authorship analysis process

![Diagram of authorship analysis process]

Figure 2.2: Profile based versus instance based authorship identification. Document \( u \) is the unlabeled document which has to be attributed to either author \( a_1 \) or author \( a_2 \). For both authors three training documents are available. In both approaches two features \( x_1 \) and \( x_2 \) are generated. The profile based model has two data points (for each cumulative profile one). The author of the suspect document then corresponds to the author whose profile has the smallest distance to suspect document \( x_u \). The instance based approach on the contrary has data points for all individual training examples \( x_{n,1} \) and a classification algorithm is employed to create a model of the two authors.

the writing style of each author. In contrast, the instance based paradigm considers the documents of the candidate authors as a set of distinct training documents. The profile based method has a straightforward classification process: a similarity metric is used to measure the distance between the profile of the unlabeled document and the cumulative profiles of the candidate authors. The unlabeled document is attributed to the author whose writing style (considered collectively in a single document) is most similar [49]. In the instance based approaches machine learning classifiers are typically used for classification.

**Feature vector versus similarity based methods**

Till now, we assumed all the instances (whether being represented per author or per document) are represented in the same numerical vector space, i.e. in \( \mathbb{R}^n \). In this
representation, each dimension corresponds to the same quantitative measurement on the document and the instances are all modeled in the same feature space, such that each dimension reflects a single measured pattern. We refer to these approaches as feature vector based approaches. In feature vector based approaches, a wide range of feature types can be combined to create an expressive representation, such as the mean sentence length, frequency of particular word-occurrences and syntactical features. The instance based methods in literature that are feature vector based usually employ a machine learning classifier. One of the best performing machine learning classifiers in the feature vector based approach is the Support Vector Machine (SVM). Support vector machines separate classes by defining hyperplanes between the classes in such a way that the hyperplanes are as far as possible from the closest members of the different classes. For a more elaborate description we refer to Section 3.1.2.

In contrast, in the similarity based approaches the instances are typically represented by character or lexical based features and classification takes place with a similarity function that quantifies the degree of shared information between the unlabeled document and the training samples. Most often, the similarity based approaches stick to the profile based paradigm. One key aspect of similarity based approaches is that instances may be modeled in a non-numerical, not well defined feature space. For example, some similarity based approaches use text compression algorithms to quantify the similarity between the samples [54, 76]. Hence, the samples are represented in their original textual form, and no feature vectors are extracted. Common n-grams (CNG) is perhaps the most representative similarity based approach, where the feature set of each author is formed by the most frequently used n-grams [15]. In this case each sample is modeled with a unique set of most frequently occurring character n-grams. This makes it difficult to transform it to a compact numerical representation, as in the case with feature vector based approaches. In addition, in similarity based methods it may be difficult to combine different kinds of stylometric features, since a similarity measure usually can only handle data that is homogeneous enough to determine the degree of similarity.

**Information retrieval approaches**

If the method is both similarity and instance based, the documents are represented individually, and the identification is done by obtaining the documents most relevant to the unlabeled document by means of a similarity measure. It is notable that in general, this is exactly one of the problems that information retrieval deals with. In general, the aim of information retrieval techniques is to obtain the documents most relevant to a certain information need. Similarly, the instance and similarity based methods obtain the samples and corresponding class labels (i.e. the information) most relevant to the unlabeled document (i.e. the information need). Therefore, these methods are sometimes refered to as information retrieval approaches [11, 50]. Stamatatos [73] seems to pertain the described properties of similarity based classification to the profile based paradigm, such as difficulty to combine different feature types. However, the information
2.1. The authorship analysis process

retrieval approaches show that this is not limited to the profile based classification\(^2\).

**Classification in authorship verification**

So far, we considered classification models for authorship identification. In authorship verification training data is available of only one author (the target author). Classification techniques in authorship verification can be distinguished into two main categories [65]:

- **Intrinsic verification models.** In this category, authorship verification is seen as a one-class classification problem, where the only class, the target class, is determined by instances in the training data. So training exclusively takes place by the data in the set of documents of the known author. Class prediction is done by setting a certain decision line as a decision boundary around the training samples in the target class.

- **Extrinsic verification models.** In the former category, setting the decision boundary is difficult, because it is hard to decide how tightly the boundary should be fit around the target class [44]. It is also hard to find representative features that are sufficiently unique for the author. In such cases it can be advantageous to use not only positive data but to use additional documents from external sources to model an outlier class. In this way, the verification problem is transformed from a one-class to a binary class classification problem. The outlier class represents all documents not belonging to the target. Hence the decision boundary is based on the similarity of an unlabeled document to both the target and outlier class instead of only considering the target class. Due to the diversity in the outlier class (it represents all other authors not being the target author) a large number of representative negative training samples should be available.

2.1.3 Classifier evaluation

The classifier is usually evaluated with a quantitative performance measure. It is not difficult to see that it is not fair to use the same data set for training and then for testing. In the holdout method the data is partitioned into a training set used to learn the classifier and a separate test set for evaluating the classifier. A drawback of this method is that it reduces the number of training or testing instances. In cross validation techniques, both a large training set can be used while the dataset is tested well at the same time and the independence between train and test set is maintained [47]. Cross validation refers to validation techniques which repeatedly select test and train sets and then average the performance over multiple test/train set combinations. In \(k\)-fold cross validation the dataset is split into \(k\) equally sized subsets (the folds) and the classifier is trained on \(k - 1\) folds and tested with the remaining fold. This procedure is then

\(^2\)An example of an information retrieval approach in source code is the technique proposed by Burrows, which will be discussed in Section 2.2.1
repeated for all \( k \) combinations test and train set. Because each fold is used once as test set, every instance will be used for validation. In the extreme case, \( k \) is chosen very large such that each fold comprises only one instance. This is known as the \textit{leave-one-out} method. Although leave-one-out cross validation will test the dataset very thoroughly, a major drawback is its high computational complexity.

The eventual performance of the classifier can be expressed by several evaluation measures [66]. For multiclass problems such as authorship identification, the \textit{accuracy} is typically used, which is the proportion of correctly classified instances:

\[
a = \frac{TP + TN}{P + N},
\]

where \( P \) is the total number of positive samples in the test set, \( N \) the total number of negative samples in the test set, and \( TP \) and \( TN \) are the number of correctly predicted positives samples (true positives) and correctly predicted negative samples (true negatives) respectively.

In authorship \textit{verification} two objectives can be distinguished: a high precision \( p \) and a high recall \( r \),

\[
p = \frac{TP}{TP + FP}, \quad r = \frac{TP}{P},
\]

where the false positive rate (\( FP \)) is the fraction of incorrectly predicted positive samples. The precision is the proportion of correctly predicted positives. To achieve a high precision, the \( FP \)-rate should be low. Consequently a document should be predicted to be positive only if there is a very high confidence that this decision is correct. However, in this case the proportion of real positives that are predicted as such (i.e. the recall) may be low. A high recall can be achieved when a document is predicted to be positive also if there is low confidence. The tradeoff between precision and recall is equivalent to how tightly the decision line is set around the target class (or between the target and outlier class). The \( F1 \)-score is a measure that combines precision and recall by expressing the performance as the harmonic mean between them:

\[
F_1 = \frac{2 \cdot p \cdot r}{p + r}
\]

In authorship verification a Receiver Operating Characteristic (ROC) curve can give insight in the relation between the false positive rate (FPR) and the true positive rate (TPR) when varying the ratio between precision and recall. The TPR is identical to the recall: it is the fraction of positives that are predicted as such. The FPR is known as the specificity and is the fraction of negatives that is predicted as such. A common quantitative final performance measure is the area under the ROC curve (AUC). It can be shown that the AUC of a classifier is equivalent to the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative instance [24].
2.2 Source code authorship identification

Thusfar we considered the process of authorship identification without focussing on source code. Fortunately, programming language concepts are based on many linguistic concepts from natural languages and as a result the authorship identification process is not very different. Features can be generated from a character based language representation. One can also consider more abstract language representation levels. The first phase in most compilers is lexical analysis. In this step the source code is converted from a sequence of characters into a sequence of tokens. These tokens form a lexical source code representation. In subsequent compilation steps, the tokens are converted into a syntax tree by following a grammar, which defines how tokens can form a syntactically correct program. As is the case in ordinary linguistics, computer languages have a grammar with a corresponding syntax and semantics. Hence, just as with natural languages, we can generate character, lexical, syntactic and domain specific features.

In literature, several closed set source code identification studies are found. However, we have not found authors that addressed the verification or open set identification problem. Among the source code authorship identification studies in literature, Burrows compared the different techniques most thoroughly [12]. He presented a comparative study that includes a technique developed by himself [10] and a reimplementation of most of the existing techniques. Burrows observed that the technique developed by himself and a technique by Frantzeskou [29] outperformed the other approaches. The techniques of Frantzeskou and Burrows deviate on some essential aspects from most other contributions. First, they are similarity based. Secondly, the features are extracted from a low level language representation (character and lexical n-gram features respectively). The similarity based approaches will be described in Section 2.2.1. Most other techniques found in literature are feature vector based and employ machine learning classification algorithms. These will be described and compared to the Frantzeskou and Burrows method in Sections 2.2.2 and 2.2.3. Many of the feature vector based approaches use domain specific features, however, also character based document representations are used.

2.2.1 Similarity based approaches

The Frantzeskou approach

Frantzeskou [29, 28] developed a Common n-gram (CNG) based identification technique. This technique is presented as the SCAP approach. This approach is profile and similarity based and uses character n-grams features\(^3\). The first step in the approach is the construction of an author profile for each candidate author. Figure 2.3 systematically shows how a profile is constructed. First all the sample files of an author are concatenated into one large file. The next step is the extraction of the \(L\) most frequent n-grams from the

\(^3\)In the approach byte level n-grams are used as features, which are in essence identical to character n-grams, but also include all non-printing characters such as carriage returns.
2. Preliminaries and Related Work

Source files
author c
Concatenate
Count frequencies of character
level n-grams
Keep the L most frequent n-grams
Scap_c = \{(x_1, f_1), (x_2, f_2), \ldots, (x_L, f_L)\}

Figure 2.3: Construction of a Scap profile for each candidate author c.

concatenated file. A profile is then defined as the ordered set of pairs consisting of the L most frequent character n-grams x_i with the corresponding normalized frequencies f_i.

The classification method is similarity based. Code with unknown authorship is classified by comparing the similarity of the candidate author profiles with the profile of the unlabeled code (the latter profile is derived in the same way). Frantzeskou evaluated two different choices for the similarity measure. The first measure is the relative distance measure, which was taken from [42]:

\[
RD = \sum_{g \in \text{profile}} \left( \frac{1}{2} \cdot \frac{f_1(g) - f_2(g)}{f_1(g) + f_2(g)} \right)^2,
\]

where \( f_1(g) \) is the normalized frequency of an n-gram g in the author profile, and \( f_2(g) \) the normalized frequencies of g in the program profile. A drawback of this measure is that if a limited number of training samples are available per author, the profile length may be smaller than L, and hence become positively biased towards the authors with a small profile. That is why a second similarity measure is introduced that uses simplified profiles, without relative frequencies: \( SP = \{x_1, x_2, \ldots, x_L\} \). The similarity measure, is then defined as the size of the intersection between the author and program profile:

\[
SPI = |SP_A \cap SP_P|,
\]

where \( SP_A \) and \( SP_B \) are the simplified profiles of the author and the program respectively. Frantzeskou examined that the SPI similarity performed consistently better or equal compared to the RD measure for different choices of n and L.

The Burrows approach

Burrows [10] proposed an instance based authorship identification method with features on the lexical language level. The method is similarity based and uses n-grams of the types of lexical token as features. Figure 2.4 depicts an overview of the approach. First, the programs of an author are subject to a lexical analysis in which the documents are tokenized. Based on the type of the tokens, a subset of the tokens is then selected to model the training documents. The token types, such as operators, keywords, identifiers
and whitespace can be denoted by two alphanumeric characters, e.g. ‘op’ for an operator and ‘wh’ for whitespace. Every document is subsequently represented by a contiguous sequence of token types corresponding to the selected tokens. The final model is then formed by extracting all the $n$-grams of token types from the document.

The similarity measure for classification is adopted from the field of information retrieval. Analogous to information retrieval, the modeled training documents are indexed in a search engine. The resulting index then allows to find the labeled training documents that are most relevant to the given query, that is formed by the token type $n$-gram representation of the unlabeled document. The most popular index structure, which is also used in this approach, is the inverted file, which is a dictionary of terms [58] (in this case the terms are token $n$-grams). The index records for each distinct term the term frequency $tf$, which represents its total number of occurrences in the data. Additionally, an inverted list is formed as a set of tuples $(d, idf_d)$ where $d$ is the document identifier and $idf_d$ is the inverse document frequency. The inverse document frequency is the frequency of the term within the document $d$. Once the index has been constructed, classification can be done by finding the most relevant document with respect to the unlabeled document that is used as query. The common notion of similarity measures in information retrieval is that the more often the query terms are in the document, the more relevant the document is. Secondly, the relevance of a document increases with the rarity of the term across the whole corpus. These conditions can be met by assigning
2. Preliminaries and Related Work

to each term a weight which is proportional to its importance both within the document and in the entire document collection. As such, the measure can roughly be described by

\[ w_{i,d} = tf_{i,d} \times idf_i \]

where \( w_{i,d} \) is the weight of document \( d \) of term \( i \), \( tf_{i,d} \) is the term frequency and \( idf_i \) is the inverse document frequency. Often, weighting also compensates for differences in document length, since larger documents tend to have a higher term frequency as they tend to repeat the same words multiple times. In the work of Burrows, five different similarity measures are compared. Burrows found that the Okapi similarity measure was most effective \([11]\). This measure involves the term frequency, inverse document length, and inverse document frequency and is defined as

\[
Okapi(q,d) = \sum_{t \in Q} w_t \cdot \frac{(k_1 + 1)tf_{t,d}}{K + tf_{t,d}} \cdot \frac{(k_3 + 1)tf_{t,q}}{k_3 + tf_{t,q}}
\]

\[ w_t = \ln \left( \frac{N - idf_t + 0.5}{idf_t + 0.5} \right), \quad K = k_1 \left( 1 - b \right) + \frac{b \cdot l_d}{l_{avg}} \]

where \( Q \) is the query document, \( D_d \) is the document, \( t \) is the term and \( N \) is the number of documents in the collection, \( l_d \) is the document length and \( l_{avg} \) is the average document length. Burrows further investigated the length of varying query document length. More importantly, the role of different token types (sixty-three in total) were examined. The best classification accuracy is achieved by selecting operators, keywords and whitespace characters features together.

2.2.2 Feature vector based approaches

The feature vector based approaches can be divided in terms of the syntactic complexity of the document representation where the features are extracted from. Most approaches use domain specific features. However, also some character based approaches can be found in literature. In general, the feature vector based approaches follow the classification approach as outlined in Figure 2.1 of Section 2.1, where the data sampling is instance based and the classifier is an off-the-shelf machine learning algorithm. Therefore, these approaches are sometimes referred to as machine learning approaches.

Text based features

Some feature vector based author identification techniques in source code use features from a low level language representation. A representative example is the character \( n \)-gram based approach, where each dimension of the feature vectors correspond to the frequency of a particular \( n \)-gram \([15]\). Also Kothari \([52, 70]\) used character based features. Kothari compared the performance of two types of features, namely character based stylometric features and character \( n \)-grams. The stylometric features refer to features such as the distribution of line sizes, the distribution of leading spaces and the distribution of words per line. It was shown that classification with character 4-grams
produced better results than classification with the stylometric features. More recently, character distribution frequencies and character based stylometric features were used for the identification of HTML code associated with malware [82].

**Domain specific features**

Most of the instance based approaches found in literature use language dependent domain specific features. One of the earliest source code authorship identification contributions considered the layout of the code [64]. Later, Krsul and Spafford identified a large number of other domain specific features which can be useful in source code authorship identification [53]. They identified layout, style and structural features respectively.

- **Layout features** are related to typographic aspects such as the indentation style.
- **Style features** concern style related aspects such as naming conventions and the degree of usage of global variables. These are less susceptible to be changed automatically by pretty printers and code formatters than layout metrics.
- **Structural features** measure the structural decomposition of the code and are more dependent on experience of the programmer, such as mean lines of code per function and software quality measures.

Several works in source code authorship identification use domain specific features and a machine learning classification algorithm, e.g. [22, 55, 52, 19, 53, 57]. In [19] 56 of the features identified by Krsul and Spafford were evaluated to classify Java source code by means of discriminant analysis. In some works domain specific features are combined with features generated from a character based document representation, such as the mean number of characters per line [57] and the number of bits in the zipped (compressed) source code [22]. Also [55] used a wide range of text based and domain specific features, such as the number of words on a line and the number of membership operators in a single logical identifier in Java code. In some cases, code metric tooling is employed for feature generation. Code metrics incorporate all kind of quantitative measures of code, such as code maintainability, coupling and cohesion. An example of a feature generated by a code tool is the cyclomatic complexity [57]. Others employed a lint static analyzer to gather information on the number of warnings in the code and used a dynamic analysis tool to determine the testability of code [35]. More complex features have also been examined in binary code authorship identification, such as subgraphs in the control flow graph (CFG) [67]. In [45], more subjective features are suggested that are identified by an expert’s judgements, such as the degree to which comments match the actual source code’s behavior.

**2.2.3 Comparison of the existing approaches**

Above we regarded a number of approaches to source code authorship identification. Since most works use different training set corpora and are based on different program-
ming languages, a reasonable comparison between the techniques is difficult. Nevertheless, we now attempt to compare the approaches in terms of obtained performance.

Classification technique

One of the first comparisons was done by Franzeskou [29], who showed the effectiveness of his common \( n \)-gram method by validating on the same dataset as the domain specific approach in [57]. As already mentioned earlier, the most prominent comparison was done by Burrows [12]. Burrows implemented most of the existing techniques and compared them with four different collections with code from student assignments and open source code. Burrows showed that the similarity based methods of himself and Franzeskou perform better than the feature vector based methods that employ a machine learning classifier\(^4\). In an independent study in [77], the accuracy of the similarity based approaches of Burrows and Franzeskou were further compared with each other. However, the authors asked themselves whether comparing the accuries of the two methods is fair. The Burrows method inherently anonymizes the data as it uses lexical types as tokens and thus neglects values of literals and comments. Therefore they anonymized all data by removing comments and string literals and found that in this case both approaches performed approximately equally well.

Document representation

Burrows seems to attribute the difference in accuracy between the techniques in his study mainly to the similarity based vs. machine learning classification paradigm. Since the similarity based approaches both use \( n \)-grams from a low level language representation and the feature vector based approaches mostly use domain specific features from a high level language representation, it is however questionable whether the difference in performance only traces back to the classification paradigm: it may also relate to the used features. Research results indeed indicate that the latter may be the case. First, in the comparative study of Burrows, the performance of the similarity based approaches were followed by a machine learning approach by Kothari, that used character \( n \)-gram features [52]. Also latter results in literature suggest that the similarity based approaches are not generally better. In [15] is shown that compared to the SCAP approach, a feature vector based approach with character \( n \)-grams performs approximately equally well. One main conclusion of that study was that the machine learning approach was more accurate in balanced training sets. Such results suggest that it are mainly the approaches that use \( n \)-gram features that achieve a high accuracy. Of course, this does not imply that the classification paradigm has no importance.

In reviewing the literature, few studies were found that sought to determine which high level program characteristics are of most significance for authorship analysis. Juola noted that text formatting and layout features lend itself in particular to source code,

\(^{4}\)Burrows refers in his study to the similarity based methods of himself and Franzeskou as information retrieval approaches. In contrast, according to our definition, only the approach of Burrows is an information retrieval approach (see Section 2.1.2).
2.3 Conclusion

because of the importance many authors place on the style in the source code [38]. Indeed, literature shows that layout is one of the most relevant indications of authorship in source code. This is the case in either domain specific and character based methods. In the domain specific approach of [19] was observed that for Java, the layout features played a more important role than the style and structure metrics. Similar conclusions followed from analysis of the character $n$-gram based SCAP method on Java source code [27, 26]. Since character $n$-grams are perfectly able to capture the formatting and layout, this is perhaps one of the reasons that character $n$-grams perform well in source code authorship identification. The JavaScript language is highly influenced by Java. Therefore the layout is expected to play an important role in the classification.

2.3 Conclusion

In this chapter we first explored the authorship analysis problem in general. We then focussed on existing source code authorship analysis techniques in literature. All works we encountered focus on the authorship identification problem with a closed set of programmers and no studies are found that investigate the authorship verification problem. The employed classification techniques can broadly be divided in similarity based techniques that employ a similarity metric for classification and and feature vector based methods that employ machine learning algorithms. Different language representations are used for feature extraction. In general, the works that use features from a low language abstraction level in the form of $n$-grams, have shown to achieve the highest accuracies. The layout of source code is among the most significant indications of source code authorship.
3

Approach

In this chapter the implemented source code authorship identification and verification approaches are presented. First, we detail the developed authorship identification and verification techniques that use character features in Section 3.1. These techniques are adopted from literature. Then we propose a new domain specific technique in Section 3.2. The domain specific approach is our own work and based on features specific to JavaScript. The most significant aspect is that the approach employs structures from the Abstract Syntax Tree (AST). In all the discussed approaches, the authorship verification problem is modeled as an extrinsic verification model and thus trained with positive and negative samples.

3.1 Character based approaches

We implemented three techniques employing features from a text based language representation. These are approaches from literature and not our own work. However, we embed the approaches in authorship verification, which is different. The implemented text based authorship identification studies include the SCAP approach (Section 3.1.1), machine learning classification with character n-grams (Section 3.1.2) and classification by compression distances (Section 3.1.3).

3.1.1 The Scap approach

The SCAP method is a Common n-gram (CNG) technique that achieved promising results in several studies, and can be considered as state of the art. As such, the results of this method will enable us to use it as base line to compare the performance with the other implemented techniques. The identification process is described in Algorithm 3.1. In this algorithm the function $\text{getProfile}(d)$ returns the $L$ most frequent $n$-grams $x_1, x_2, \ldots, x_L$ from document $d$. An unlabeled document is classified to the author whose profile is most similar to the unlabeled document profile. Since the SCAP method has been described elaborately in Chapter 2 we refer to Section 2.2.1 for further details.
3. Approach

In our study, we set $L$ to 1500 as was originally suggested by Franzeskou [29]. Other studies, however, reported a different optimal choice for $n$ and $L$, such as $n = 14$ in combination with an unlimited profile size [12]. We leave it as future work to tune the values of $n$ and $L$. Since computing the intersection between profiles is computationally heavy, we stuck to the (rather small) proposed value of $L$. This allowed us to validate the method more extensively and to test with a larger number of candidate authors.

Scap and authorship verification

We adapted the algorithm in order to be able to employ it in the context of authorship verification. To accomplish this, we adopted two different approaches. What distinguishes them, is how the outlier class is modeled. First, the outlier class may be represented by one cumulative profile $P_1$, which consists of the most frequent $n$-grams in all concatenated negative documents. Intuitively, an unlabeled document which belongs to the target is likely to have a higher proportion of $n$-grams that match the target profile than a document of an arbitrary other author. This approach is shown in Figure 3.1(a). In this figure, the decision line is defined in terms of the similarity between the unlabeled profile with respect to the outlier profile $P_1$ and the target profile $P_{\text{target}}$. More precisely, an instance is classified as target iff $d(P_u, P_{\text{target}}) - d(P_u, P_1) < k$, where $k$ is the threshold value and $d$ is the distance function. Note that a document is not necessarily classified as target if its distance to the target class is smaller than the distance to the outlier class. The second approach to model the outlier class is by using multiple outlier profiles $P_i$, each corresponding to a different author $i$. This approach is shown in Figure 3.1(b). In this case an instance is classified as target iff $d(P_u, P_{\text{target}}) - d(P_u, P_i) < k \forall i$.

---

**Input:**
- $C$: set of candidate authors
- $D_c$: training samples for each candidate author $c \in C$
- $u$: unlabeled sample
- $L$: profile size
- $n$: gram length

**Algorithm 3.1: The Scap algorithm.** This is a common $n$-gram (CNG) approach. Classification takes place by determining the most common character $n$-grams between the unlabeled code and the training data of each author.

```plaintext
1: function Scap
2: $P_u \leftarrow \text{getProfile}(u, L, n)$
3: for $c \in C$ do
4: $P_c \leftarrow \text{getProfile}(\text{concat}(D_c), L, n)$
5: $S_c \leftarrow |P_u \cap P_c|$  
6: end for
7: return argmax$_c(S_c)$
8: end function
```

24
3.1. Character based approaches

The threshold value $k$ defines the required confidence that an instance is positive to be classified as such and determines in this way the bias to the target class. By regulating the required confidence that an instance is positive to be classified as such, we aim to be able vary ratio between the precision and recall of the classifier. In the SCAP method, this confidence is completely determined by the value returned by the similarity function. Also the Receiver Operating Characteristics curve (ROC-curve) is obtained by varying the threshold, and tracking the FPR and TPR at several choices for the value of $k$. Clearly, in such a cases the distance metric is of great importance. The $SPI$ similarity is not a proper distance metric and has some other weaknesses. That is why we adopted the Jaccard index instead of the $SPI$ measure for authorship verification. We now explain the rationale behind this decision by taking a closer look at the similarity metric.

A closer look at the similarity metric

In the SCAP approach the similarity between profiles is defined by the $SPI$ measure, which is the cardinality of the intersection between profiles, i.e. the similarity between the unlabeled profile $u$ and the author profile of an author $i$ is $SPI_i = |P_u \cap P_i|$. This metric can be considered as a precision measure. Remember the precision is the fraction of retrieved instances that are relevant. Now the similarity measure $SPI_i' = \frac{SPI_i}{|P_u|}$ expresses the fraction of relevant $n$-grams in the author profiles (and thus is a precision measure) and is monotonic with $SPI$. Consequently it will result in an identical authorship identification process.

In the $SPI$ measure it is important that the author profiles $P_i$ are assumed to have the same length $L$, so that the measure is not biassed towards authors with a larger profile. There may however, not be enough data to model the outlier classes with $L$ $n$-grams, such that this assumption cannot be made. This is depicted in Figure 3.1(c), where the two author profiles $P_1$ and $P_2$ have different sizes. In this case, not only the precision is relevant but also the recall. The precision $p_i$ is the fraction of matching
3. Approach

$n$-grams in the unlabeled profile, and the recall $r_i$ is the fraction of matched instances in author profile $i$:

$$p_i = \frac{|P_u \cap P_i|}{|P_u|}, \quad r_i = \frac{|P_u \cap P_i|}{|P_i|}.$$

In Figure 3.1(c) the precision $p_1$ and $p_2$ is approximately equal, while the $r_2 > r_1$. Both the precision and recall are of importance, so we define the new similarity measure as the harmonic mean of precision and recall, the $F_1$-score. This measure addresses the problem with author profiles with unbalanced sizes. Now, the obtained $F_1$ score is equal to the Sørensen–Dice coefficient. This metric, $\text{Dice}(P_u, P_i) = \frac{2|P_u \cap P_i|}{|P_u| + |P_i|}$, is a commonly used similarity measures in information retrieval [14] and taxonomy and bio-associational studies [16].

$$F_1 = 2 \cdot \frac{p_i \cdot r_i}{p_i + r_i} = \frac{2}{\frac{1}{p_i} + \frac{1}{r_i}} = \frac{2|P_u \cap P_i|}{|P_u| + |P_i|} = \text{Dice}(P_u, P_i)$$

Note that the Dice coefficient is monotonic with the SPI measure, when $|P_i| = L$ for all $i$. Consequently, using the Dice coefficient will not lead to a different classification, if all author profiles are of length $L$. It can however be shown that the according distance measure $1 - \text{Dice}(P_u, P_i)$, is not proper distance function as it does not satisfy triangle inequality [14]. The Jaccard index $J(P_u, P_i) = \frac{|P_u \cap P_i|}{|P_u \cup P_i|}$ is not very different from the Dice coefficient\(^1\), but is a proper distance metric. Furthermore, a nice property is that $0 < J < 1$. The confidence $C$ that an unlabeled document is a target object can then be expressed by:

$$C = \frac{1}{2} + \frac{J(P_u, P_{\text{target}}) - J(P_u, P_i)}{2},$$

where $P_i$ is the most similar outlier author profile. The unlabeled document is then predicted to belong to the target class if $C > k$. Different choices for the threshold $k$ will result in a different ratio between the precision and recall.

3.1.2 The $n$-gram based approach

The $n$-gram approach is feature vector based. Together with the similarity based SCAP method, this method has shown to be quite effective in literature [15]. Each document is described by a vector $x_i \in \mathbb{R}^m$, where each dimension corresponds to the relative frequency of the occurrences of an $n$-gram in the document. A problem in $n$-gram based techniques is the curse of dimensionality. The ASCII table includes 95 different printable characters, which will results in $95^8 \approx 6.6 \cdot 10^{15}$ character 8-grams. Obviously, this dimensionality cannot be handled well. What can be done is to decrease $n$ or to select a subset of $n$-grams. Most techniques in literature limit the number of dimensions by selecting only the most frequent character $n$-grams that are present in the sample data collection. We also adopted this strategy. Not all the selected $n$-grams are necssarily

\(^1\)Note that it can be written as $J(P_u, P_i) = \frac{|P_u \cap P_i|}{|P_u \cup P_i|} = \frac{|P_u \cap P_i|}{|P_u| + |P_i| - |P_u \cap P_i|}$.
3.1. Character based approaches

observed in a particular code sample. Consequently, the feature vector may be sparse. Therefore we chose a larger number of \( n \)-grams than was done in the Scap method and utilized the 7000 most frequent \( n \)-grams. For classification we follow the instance based paradigm and classify by means of a Support Vector Machine (SVM). Since classification by an SVM is much faster than classification by the SPI measure in the Scap method, this dimensionality was computationally feasible. The SVM classifier was chosen, since SVMs are among the best text classification algorithms to date [1]. SVMs have many properties that make that they perform well on text classification. For instance, they can handle a high dimensional input space and they are robust when document vectors are sparse [37]. The SVM is a binary classification algorithm. Authorship identification with more than 2 authors requires a multiclass classifier, which can be constructed by combining several binary classifiers [36]. In authorship verification the two classes are modeled with the positive and negative data. By estimating the posterior probabilities of a class prediction, the threshold can be varied.

The Support Vector Machine

We now briefly give a theoretical background behind the SVM classifier [25]. Consider the binary classification problem in a two dimensional space in Figure 3.2. Each training sample \( x_i \) is labeled by class label \( y_i \), such that \( y_i = +1 \) for positive samples and \( y_i = -1 \) for negative samples. An SVM separates the positive samples from the negative examples by defining a hyperplane between the instances. This decision boundary can be described by \( w \cdot x + b = 0 \), where \( w \) is (by definition) normal to this hyperplane. In Figure 3.2(a), the line is intuitively a good choice for the decision boundary, since it is far away from the data of both classes. An unlabeled document \( u \) is then predicted positive if it falls in the region above the decision line, described by \( w \cdot u + b > 0 \), and negative otherwise. The question remains how the direction and location of the decision line can be determined.

We first assume that the data points of the classes are linear separable such that

\[
\begin{align*}
  x_i \cdot w + b & \geq +1 \quad \text{for} \quad y_i = +1 \\
  x_i \cdot w + b & \leq -1 \quad \text{for} \quad y_i = -1.
\end{align*}
\]

Thanks to the definition of the two label values, these two equations can then conveniently be combined into

\[
y_i (x_i \cdot w + b) \geq 1 \quad \forall i.
\]

This constraint is however not enough to pick a particular value for \( w \) and \( b \), since there are many possible orientations of the decision boundary. Hence, another constraint needs to be defined, that boils down to selecting \( w \) and \( b \) such so that the decision line is as far as possible away from the closest members (the support vectors) of both classes. This notion can be formalized by defining two additional hyperplanes \( H_1 \) and \( H_2 \) perpendicular to the decision boundary on which the closest members are. By rescaling \( w \) and \( b \) appropriately, these planes can be described by:

\[
\begin{align*}
  x_i \cdot w + b & = +1 \quad \text{for} \quad H_1, \\
  x_i \cdot w + b & = -1 \quad \text{for} \quad H_2,
\end{align*}
\]

The whole region enclosed by these two hyperplanes is the so called margin. If we now define a new hyperplane \( H \) that is exactly in the middle of the margin, we have:

\[
x_i \cdot w + b = 0 \quad \text{for} \quad H
\]

This new hyperplane is called the optimal hyperplane, since it has the maximum distance to the closest members. A support vector is a training sample that lies exactly on the hyperplane.

The optimal hyperplane can be found by solving the optimization problem:

\[
\min_{w,b} \frac{1}{2} w^T w + C \sum_i \xi_i
\]

subject to:

\[
\begin{align*}
  y_i (x_i \cdot w + b) & \geq 1 - \xi_i \quad \forall i \\
  \xi_i & \geq 0 \quad \forall i
\end{align*}
\]

where \( \xi_i \) is a non-negative slack variable that allows a training sample to lie on the wrong side of the hyperplane. The parameter \( C \) controls the trade-off between maximizing the margin and minimizing the classification errors. SVMs are very robust to outliers, since the hyperplane is not affected by training samples that lie far away from the decision boundary. The classification of a new sample \( x \) is then done by projecting it onto the optimal hyperplane and checking whether it lies on the correct side of the decision boundary.
which can again be combined into a single formula:

\[ y_i(x_i \cdot w + b) = 1 \forall i. \]

With this equation we are now able to calculate the distance between the two hyperplanes. Consider two points \( x_+ \) and \( x_- \) on the margins. The length of the difference between these vectors is \( x_+ - x_- \). Since we know that \( w \) is perpendicular to the hyperplanes, the width of the margin can now be expressed by the dot product between the unit normal vector and the difference vector, \( (x_+ - x_-) \cdot \frac{w}{||w||} \). By using the defined formula for the margin hyperplanes, we can rewrite it to \( x_+ \cdot w = 1 - b \) and \( -x_- \cdot w = 1 + b \), so that the width of the hyperplane can be expressed as \( \frac{2}{||w||} \). This is exactly the value we want to optimize to find the maximum-margin hyperplane:

\[
\max \frac{2}{||w||} \quad \text{s.t.} \quad y_i(x_i \cdot w + b) \geq 1 \forall i.
\]

It can be shown that this optimization problem can be solved by expressing it as the equivalent quadratic programming problem

\[
\min \frac{1}{2}||w||^2 \quad \text{s.t.} \quad y_i(x_i \cdot w + b) \geq 1 \forall i.
\]

We will not go through the mathematical details of this. By using the so called kernel trick, the SVM can also be efficiently used in non-linear classification problems. In this case a kernel function is used to map the data to a higher dimensional space. The data in the higher dimensional space may be linearly separable, even if this is not the case in the original lower dimensional space. However, it can still not be guaranteed that all data is linearly separable. To make the algorithm work for non-linearly separable datasets as well, we reformulate the minimization problem by introducing a cost function, to pay a cost for points that fall in the margin (and thus have a distance to the decision boundary smaller than 1):

\[
\min \frac{1}{2}||w||^2 + C \sum_i \xi_i \quad \text{s.t.} \quad y_i(x_i \cdot w + b) \geq 1 - \xi_i \forall i, \quad \xi_i \geq 0 \forall i.
\]
The soft margin parameter $C$ controls the balance between two goals: maximizing the width of the hyperplane and ensuring that most samples have functional margin at least 1. By tuning this parameter a tradeoff can be made between the bias and variance of the SVM. The strengths of an SVM are that they scale well to high dimension data and they have nice theoretical properties: unlike neural networks, an SVM does not suffer from local optima. Furthermore the SVM can be controlled by tuning margin parameter $C$. In our approach we will use a linear kernel function.

### 3.1.3 Compression based approach

Compression based approaches employ text compression distances between documents. The basic idea is that documents that have similar characteristics should be compressed with each other more easily than documents with dissimilar characteristics. This idea is not new and has been used in various domains, such as text categorization and language identification [76]. In our work we employed the compression based technique that was reported in [54]. The compression distance metric that will be used is:

$$CDM(d_1, d_2) = \frac{C(d_1d_2)}{C(d_1) + C(d_2)}$$

where $C(d)$ is the size of the compressed document $d$ and $d_1d_2$ is the concatenation of $d_1$ and $d_2$. If the two documents are similar then $CDM$ approaches 0.5, while it approaches 1 if they are completely unrelated.

For classification, the nearest neighbour classification method could be applied, where the unlabeled document is classified to the author whose source code the best compresses jointly. However, we follow the approach of [54] that reduces the risk of overfitting. In this approach, for an unlabeled document a vector is created with the distances of this document to all the training texts. Then an off-the-shelf classification algorithm is capable to attribute the unseen document to one of the authors. However, in case the training set is large, the computational burden can become too heavy, since the feature vectors will become as long as the number of documents in the training set. To overcome this problem, a subset of documents of each author in the training set can be selected. The selected samples are indicated with prototypes of the training set. Figure 3.3 depicts the construction of such a vector. In this example two prototypes are selected per author, so that a 6-dimension feature vector $x \in \mathbb{R}^6$ is created for the unlabeled document. Subsequently, the distance of the training documents to the prototypes are determined in a similar manner. An SVM classifier is trained by the feature vectors corresponding to the training documents. Algorithm 3.2 describes the explained approach in more detail.

The text compression algorithm to compute the $CMD$ metric is performed by a variant of the Prediction by Partial Matching (PPM) algorithm, namely PPMd [18]. PPM is a lossless compression algorithm, which assumes that every next symbol is dependent on previous symbols. The key issue is to let the alphabet be the alphabet of sequences and not alphabet of single symbols [40]. The algorithm maintains frequencies for past character sequences, and compresses the text by predicting the next symbol.
3. Approach

Figure 3.3: Feature vector construction in the compression based approach in a 3 class classification problem and two prototypes per class.

---

**Input:**
- $C$: set of candidate authors
- $D_c$: training samples for each candidate author $c \in C$
- $d_u$: unlabeled sample

1. Select prototypes $P = (p_1, p_2, \ldots, p_r)$ from the training samples in $D_c$
2. for each $c \in C$ do
3. for each $d_t \in D_c$ do
4. $x^{(c)}_t \leftarrow (x_1, x_2, \ldots, x_r)$ where $x_i = CDM(p_i, d_t)$
5. end for
6. end for
7. $x^{(u)} \leftarrow (x_1, x_2, \ldots, x_r)$ where $x_i = CDM(p_i, d_u)$
8. Train an SVM with $x^{(c)}_t$ with all $t$ and $c$
9. Predict class membership $c \in C$ of $x^{(u)}$ with the trained classifier

---

Algorithm 3.2: Compression based authorship identification. This method has either aspects of similarity and feature vector based techniques, and can thus be regarded as a hybrid approach.

---

using these frequencies. Thus each symbol is encoded according to the context provided by the preceding symbols [76].

3.2 Domain specific approach

The approaches discussed so far, use character based features. In this section we present a feature vector based approach that that employs features specific to JavaScript, and classifies with an SVM. The domain specific features used in literature can be divided into layout, style and structural features respectively\(^2\). We also extracted these features, however, we propose to generate a much larger feature space, by parsing the code into an AST prior to feature extraction. Most important of this representation is that enables an easy way to capture structural patterns in the tree. By our knowledge no prior studies in authorship analysis have considered such structures in the parse tree. The

\(^2\)See Section 2.2.2.
3.2. Domain specific approach

![Diagram of CST and AST](image)

**Grammar:** Exp → Num
Exp → Exp “+” Exp

**Source:** 2 + 3 + 5

**CST:**
- Exp
  - Exp
    - Num
    - Exp
      - Num
      - Exp
        - Num
        - +
        - Num
        - +
        - 5

**AST:**
- Plus
  - 2
  - 3
  - 5

Figure 3.4: The difference between the Abstract Syntax Tree and Concrete Syntax Tree. The grammar describes the language for the addition of numbers. Note that the grammar is ambiguous as it is possible to derive two different parse trees from the source code. Hence, in practice additional rewrite rules are necessary, which will result in an even bigger CST (while retaining the same AST).

extraction the domain specific JavaScript features was carried out by a tool we developed\(^3\) in JavaScript by using Node.js. The parse tree of JavaScript programs is obtained by the Esprima ECMAScript parser\(^4\), which generates a Mozilla compatible AST. Before we further detail the feature extraction and selection process, we describe the AST program representation.

### 3.2.1 AST program representation

Parse trees are usually employed in compilers and language tools for code analysis and allow to select detailed structural features from the program. Crucial is the difference between the Concrete Syntax Tree (CST) and Abstract Syntax Tree (AST) [78]. Figure 3.4 illustrates the difference between AST and CST in more detail. The CST represents the concrete syntax of the source language defined by the concrete grammar for parsing the source code. This tree includes many of the punctuation tokens from the syntax and extra nonterminal symbols for technical purposes such as elimination of ambiguity and left recursion. The AST is the result of simplifying the CST down to the real content and meaning of the code. Therefore, it may reveal variation among authors and is the preferred tree to be used in authorship analysis. For JavaScript program analysis, we adhere to the Abstract JavaScript Syntax specification as described in Appendix A. An example of a parse tree obtained by this grammar is depicted in Figure 3.5. In the remainder of this section we describe the rationale behind the different types of domain specific features which are generated from the parse tree. The different features are described exhaustively in Appendix B.

### 3.2.2 Domain specific features

As stated above, we extracted *layout, style* and *structural* features from the parse tree. We will now discuss how we extracted these feature types from the AST.

\(^3\)https://github.com/wilcowisse/v.js
\(^4\)http://esprima.org
3. Approach

Programming layout features deal specifically with the layout of the program, such as indentation and spacing. A problem in the AST program representation is that this kind of information is disregarded, since it is inessential for program analysis during compilation or interpretation. That is why we performed a number of refactorings on the AST to re-add this kind of layout information from the source code to the tree. This is done by introducing additional nodes of the type Layout, BlockComment and LineComment as children to the nodes in the AST, as is shown in Figure 3.6. For instance, in a member expression the object and the property are separated by a dot. The author is free to insert spacing before and behind this punctuation mark. In the example in Figure 3.6, two additional child nodes will be appended to the MemberExpression-node. We describe the layout slots for this node type as object 1 property, where the numbers represent the locations in the source code the layout nodes correspond to. Similarly, layout nodes are added to many other programming constructs. These are elaborated in Appendix A. We grouped different layout slots with roughly the same meaning, such as the layout slots before closing parenthesis and the layout slots before different operators in binary expressions. For each defined layout feature we defined 12 histogram intervals. These intervals record whether zero, one, two or many spaces, tabs and carriage returns are used in each layout slot. Layout with comments are described by the introduced comment nodes. Using these nodes we identify the ratio between line comments and block comments. Furthermore, we track the length of the comments and the node types of the parent nodes, which is an indication of the preferred location to put comments in the source code.
3.2. Domain specific approach

Figure 3.6: Adding layout nodes to the AST of Figure 3.5.

Style features
Programming style features deal with characteristics of the program that are not easily being changed by pretty printers, such as variable length. The following list describes the style features we extracted from the AST.

- **String patterns.** The nodes in the AST convey much textual information, such as the name of identifiers. The frequency of matches of regular expression form the histogram groups for each feature. For instance, various naming conventions may be adopted in the code, such as CamelCase and Pascal Case. These are captured by the first character of identifiers related to function identifiers and objects. Furthermore type information can be checked. Although JavaScript is a loosely typed language we can match quotes in literal values (i.e. strings) and numeric values.

- **String length.** A similar approach is used to match the length of strings in the AST nodes. This enables to identify features such as the length of identifiers and the length of literal values.

- **Number of children** Some node types have a list of child nodes. The length of different kind of these lists are recorded. Think of the number of parameters in function declarations and the number of elements in the in initialization of an array.

Structure features
The program structure is captured by program structure features. The structural decomposition of the tree, is defined with two types of structural features, which are listed below. Figure 3.7 depicts these feature types.

- **Descendant node count.** We identify the complexity of programming constructs by counting the number of descendent nodes of the child nodes belonging to particular node types. The size of child nodes determines the complexity programming
3. Approach

instructions in many cases, such as the complexity of the parameters and arguments. For instance, the following JavaScript fragments clearly shows an increased complexity of passed function arguments in comparison to `foo.bar(a,1)`, see also Figure 3.7(a).

```javascript
foo.bar(a, function(a,b)
  ...
});
```

Similarly, many other node types hold such information, such as the size of return arguments and the size of binary expressions.

- **Node n-grams.** With node n-grams we aim to capture various structural characteristics. They capture the used rewrite rules in the JavaScript abstract grammar, as described in Appendix A. First, we track the occurrence frequency of uni-node-grams. This is in essence the frequency distribution of individual node types, i.e. the expression and statement types. The variation between node types may for instance indicate the preference for `While`, `For`, `ForIn` and `DoWhile` loops. Furthermore, the frequency of constructs such as `TryStatements`, `NewExpressions` and `MemberExpressions` may be a strong indicator of defensive programming or an object-oriented or functional programming style. Next to unigrams we constructed node 2-grams and node 3-gram features. A node 2-gram is represented by two node types connected by an attribute. This boils down to the determination of the frequency of every unique pair (A →attribute B), where node A is the parent of node B in the parse tree. Similarly, a node 3-gram is represented by three nodes connected by two attributes. Refering to the parse tree in Figure 3.5, examples of 2-node-grams are (MemberExpression →object Identifier), (MemberExpression →property Identifier) and (CallExpression →callee MemberExpression). Node 2-grams should be able to determine structural characteristics in the parse tree. For instance, the constructs that are returned from a function, parameter
3.2. Domain specific approach

types, the kind of update expressions in a for loop and many other characteristics. Considering the code fragment `foo.bar("a",1)` of Figure 3.5, another author may prefer to call the function like `(function(){ ... })("a",1);` This will be captured by a higher frequency of the (CallExpression $\rightarrow$ callee FunctionExpression) bigram. Node 3-grams contain more contextual information than node 2-grams. An example is the following fragment, where functions are defined in an array that is passed as function argument. They are described by the node 3-gram (CallExpression $\rightarrow$ arguments ArrayExpression $\rightarrow$ elements FunctionExpression). The use of such structures may strongly vary per author.

```javascript
bar([
    function(a,b){...},
    function(a,b){...}
]);
```

3.2.3 Refactorings to the AST

To extract structural patterns, only the node types in the tree are considered. As a result, information that is enclosed as attributes of the nodes is possibly neglected. Therefore, we performed some refactorings in the parse tree, by representing some attribute values as child nodes. Another refactoring is that we introduced some new node types. The reason is that some nodes types had multiple concrete representations in the source code. For instance, a MemberExpression can be either a member accessed by an identifier (i.e. `foo.bar`, where `bar` is an identifier), or a member accessed by an expression (i.e. `foo[bar]`, where `bar` is an expression). Also the use of parenthesis in expressions are implicit in the tree structure. Such syntactical differences have a different meaning in the source code and also contain different layout slots. The refactorings performed on the AST are detailed in Appendix A.

3.2.4 Feature extraction

For every defined feature we maintain a histogram distribution of the relative frequency of occurrences, as was done in several other studies, e.g. [41, 55, 70]. In this approach the range of the feature values is splitted into a number of bins. Then for each bin, the number of occurrences (matches) in the sample data are counted. The approach is described in Algorithm 3.3. For instance, we define multiple intervals (i.e. the histogram distribution) for the number of function arguments (i.e. the feature). We then track the number of occurrences of arguments in all function definitions in the source code. The different features and histogram values for each feature are listed in Appendix B. After the occurrences have been counted, we flatten $X$ to a $i \cdot j \times 1$ vector $x$ and normalize its values. We evaluate three different types of normalization:

- Normalization of the vector $x$, such that the sum of its elements becomes 1.
- Feature wise normalization, such that the sum of occurrences in the histogram groups for each different feature becomes 1 (unless the feature is not observed).
3. Approach

Input:

\[ F \leftarrow \text{a set of features.} \]
\[ P \leftarrow \text{a set of AST program root nodes belonging to the author.} \]
\[ X \leftarrow \text{an } i \times j \text{ empty matrix which contains for each feature } f_i \in F \text{ a set of } j \text{ histogram intervals.} \]

1: visit\((p_i)\) for each \(p_i \in P\)
2: normalize\((X)\)
3: function visit\((n : \text{node})\)
4: \hspace{1em} for each \(f_i \in F\) do
5: \hspace{2em} if \(f_i\) matches \(n\) then
6: \hspace{3em} determine feature value and increment \(X_{i,j}\) accordingly.
7: \hspace{1em} end if
8: \hspace{1em} end for
9: visit\((n.chiildren)\)
10: end function

Algorithm 3.3: Domain specific feature generation.

<table>
<thead>
<tr>
<th>Node 1-gram</th>
<th>Count</th>
<th>Node 2-gram</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>DebuggerStatement</td>
<td>7</td>
<td>IfStatement \rightarrow test ArrayExpression</td>
<td>0</td>
</tr>
<tr>
<td>LabeledStatement</td>
<td>39</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>WithStatement</td>
<td>79</td>
<td>IfStatement \rightarrow test ThisExpression</td>
<td>1</td>
</tr>
<tr>
<td>DoWhileStatement</td>
<td>110</td>
<td>ForStatement \rightarrow update Literal</td>
<td>1</td>
</tr>
<tr>
<td>ContinueStatement</td>
<td>685</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>ExpressionStatement</td>
<td>245615</td>
<td>WhileStatement \rightarrow body, ContinueStatement</td>
<td>2</td>
</tr>
<tr>
<td>CallExpression</td>
<td>306206</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>Literal</td>
<td>395096</td>
<td>CallExpression \rightarrow arguments, Literal</td>
<td>132337</td>
</tr>
<tr>
<td>MemberExpression</td>
<td>471884</td>
<td>CallExpression \rightarrow callee, MemberExpression</td>
<td>205447</td>
</tr>
<tr>
<td>Identifier</td>
<td>1607897</td>
<td>MemberExpression \rightarrow object, Identifier</td>
<td>297063</td>
</tr>
</tbody>
</table>

Table 3.1: Some of the least and most frequent node 1 and 2-grams in the dataset, with their absolute number of observations in ascending order. For the details about the dataset we refer to Section 4.2.

- Binary discretization, such that each histogram value becomes 1 if it is observed at least once and 0 otherwise.

3.2.5 Feature selection

The resulting JavaScript AST syntax has 38 different statement and expression types. Consequently, the number of unique 1, 2 and 3-grams becomes computationally heavy. Therefore we removed a large number of node \(n\)-grams. First, the abstract grammar defines for the child attributes whether an expression or statement is expected, which eliminates a large number of possible node 2 and 3-grams. Also, not all node \(n\)-grams that can be derived from the abstract syntax are valid in JavaScript. For instance, the code `new 1+1()` can be parsed without errors, since the callee in a NewExpression can
be of any type of expression. In practice, this code will result in a runtime error, and consequently the node 2-gram (CallExpression → callee, BinaryExpression) is not expected to be observed in JavaScript code collections. Moreover, some structures are valid, but were expected to be observed very rarely in the code, such as \texttt{while(...)break}, with the associated bigram (WhileStatement → body, BreakStatement). To eliminate such infrequent $n$-grams we empirically evaluated what were the most frequent node $n$-grams in all the JavaScript code in our dataset. This selection process was carried out recursively as follows. First, we selected the node $n$-grams that were observed 18 times or more, starting at $n = 1$. Then the node $n$-grams that occurred 4374 times or more were extended to node $n + 1$-grams. The same process then was performed for node $n + 1$ grams. The numbers 18 and 4374 originate from the dataset: 10% of the 182 authors which owned together 4374 repositories. To get an idea of the selection process, Table 3.1 lists the most frequent and least frequent 1 and 2-grams found in the dataset. After the node gram selected procedure we ended up with 1015 node gram features. For the other features we then selected the numerical histogram intervals as they have been documented in Appendix B. These intervals were chosen by intuition while counting the absolute number of occurrences in the entire dataset, such that empty or over-full bins were avoided.

## 3.3 Conclusion

In this chapter we presented the approaches that will be used in authorship identification and verification of JavaScript source code. We presented three existing approaches that employ low level character-based features. The SCAP approach is similarity based, while the other ones are feature vector based. Furthermore, the similarity measure used in the SCAP method was adapted in order to be employed in authorship verification. The domain specific approach differs from the character based approaches in the sense that it extracts features from the AST and is thus language specific to JavaScript. Table 3.2 summarizes the techniques.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Document representation</th>
<th>Classification</th>
<th>Document sampling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scap</td>
<td>Character ($n$-gram)</td>
<td>Similarity based, SPI</td>
<td>Profile based</td>
</tr>
<tr>
<td>$n$-gram based</td>
<td>Character ($n$-gram)</td>
<td>Feature vector based, SVM</td>
<td>Instance based</td>
</tr>
<tr>
<td>Compression based</td>
<td>Character (plain text)</td>
<td>Hybrid, SVM</td>
<td>Instance based</td>
</tr>
<tr>
<td>Domain specific</td>
<td>AST</td>
<td>Feature vector based, SVM</td>
<td>Instance based</td>
</tr>
</tbody>
</table>

Table 3.2: The techniques under investigation in this study.
In this chapter we describe the experimental setup and the methodology by which the research question will be answered. We use source code from GitHub repositories to validate the authorship identification and verification techniques. Remember that the task of authorship identification is to predict the most likely author in a set of authors, while in authorship verification task is to predict whether a document was written by an author or not. In Section 4.1 we describe the validation procedure. Then we address the construction of the JavaScript source code collection in Section 4.2.

4.1 Validation procedure

To validate the proposed authorship analysis techniques, they were implemented in Matlab. For classification we adopted PRTools [20]. PRTools is a Matlab toolbox for pattern recognition that defines many basic pattern recognition concepts and routines. The evaluated feature vector based approaches employ an SVM for classification. Although PRTools provides a Matlab SVM implementation, we chose libsvm [13] for speed considerations. Libsvm is a C++ implementation of the SVM classifier. Fortunately, this library has a Matlab interface and is compatible with PRTools. To compute one-class classification related concepts such as ROC-curves and AUC-values, we employed dd_tools [75], which is a one-class classification toolbox extending the PRTools toolbox. Next, we separately describe the validation procedure for authorship identification and verification.

4.1.1 Authorship identification

The process for validating the identification techniques is described in Algorithm 4.4. We measured the accuracy of the techniques with a varying number of training samples per author and a varying number of candidate authors. In this algorithm we repeatedly select a random subset of authors from the whole dataset. To reduce variance in the test, 10 iterations are performed using a different set of authors. Per iteration, the trained classifier is tested with at least \( n_{test} \) samples per author. Because some authors had
4. Experimental Setup

For authorship verification we used the same datasets as were used for authorship identification. One author is selected as target author which is modeled by a varying number of training samples. Since we solved the authorship verification as a binary class classification problem, we modeled also the outlier class by selecting (a possible heterogeneous) set of samples from the other authors in the dataset. To reduce the expected variance in the test results, the outlier class is validated with 5-fold cross validation. The validation approach is detailed in Algorithm 4.5. Significant is that the outlier samples used for

a very large number of samples, we limited the number of test samples per author to 16 in practice. Limiting the maximum number of test samples per author prevented an imbalanced testing set and a too high computational burden. In the algorithm, the function logspace\((a, b, n)\) generates \(n\) logarithmically spaced integers between \(a\) and \(b\) with no repetitions. We express the performance of the classifier by the accuracy, since most other works used this measure, which makes it easy to compare them. The accuracy is defined as

\[
a = \frac{TP + TN}{P + N},
\]

where \(TP\) and \(TN\) are the number of true positives and true negatives, and \(P\) and \(N\) are the total number of positive and negative samples [24].

To prevent topic biases, we defined a code sample to be an entire JavaScript repository. This ensures that the same repository is not used either for training and validation. As a result, the evaluation results will be more representative for varying coding style between projects, since classification will less likely be based on characteristics of a particular project, such as variable names specific to a project.

4.1.2 Authorship verification

For authorship verification we used the same datasets as were used for authorship identification. One author is selected as target author which is modeled by a varying number of training samples. Since we solved the authorship verification as a binary class classification problem, we modeled also the outlier class by selecting (a possible heterogeneous) set of samples from the other authors in the dataset. To reduce the expected variance in the test results, the outlier class is validated with 5-fold cross validation. The validation approach is detailed in Algorithm 4.5. Significant is that the outlier samples used for

Input:

- \(C\): candidate authors in the dataset
- \(D_c\): training samples for each candidate author \(c \in C\)

1: for \(n_{author} \leftarrow \text{logspace}(2, 75, 10)\) do
2:  for \(n_{train} \leftarrow \text{logspace}(1, 25, 6)\) do
3:     for \(i \leftarrow 1 \ldots 10\) do
4:         Select a random \(C' \subseteq C\), such that \(|C'| = n_{author}\) and \(|D_c| > n_{train} + n_{test} \forall c \in C'\).
5:         Train with \(n_{train}\) randomly selected samples from \(D_c\) for each \(c \in C'\).
6:         Test with \(t\) remaining samples in \(D_c\) for each \(c \in C'\) such that \(n_{test} \leq t \leq n_{max}\)
7:         Track accuracy \(p_i\) of the instances in the test set.
8:     end for
9:     \(p = \text{mean}_i(p_i)\) \quad \triangleright \text{Performance with } n_{author} \text{ authors and } n_{train} \text{ training samples.}
10: end for
11: end for

Algorithm 4.4: Validation procedure for authorship identification techniques.
testing are selected from different authors than the outlier samples for training. In this way we modeled the case that the negative test samples belong to an author we never observed before. Computing the AUC values and coordinates of the ROC curve is carried out by the dd_tools Matlab toolbox.

4.2 Source code collections

A labeled set of JavaScript code is needed in order to validate the proposed authorship analysis techniques. Unfortunately, no such JavaScript datasets are readily available. Therefore, we developed a code collection tool for mining source code from GitHub. GitHub is a web based service for hosting repositories of the distributed Version Control System Git\(^1\). It extends Git by an online interface and several collaboration features, such as forking, wikis and pull-requests. It has become the largest code host in the world, with more than 5M developers collaborating across 10M repositories \[32\]. Many of those repositories are open source software projects, which makes GitHub a very adequate source to mine source code. We constructed two different datasets:

- **Dataset A1 and A2** consists of a number of authors with many repositories developed solely by the authors. Dataset A2 is identical to dataset A1, but in this dataset all the code is modified by a JavaScript minification algorithm. The

\[1\]\(https://github.com/\)

\begin{algorithm}
\begin{algorithmic}[1]
\State **Input:**
\State \(C\): candidate authors in the dataset
\State \(D_c\): training samples for each candidate author \(c \in C\)
\For {\(n_{\text{train}} \leftarrow \logspace(1, 25, 6)\)}
\State Select a set of authors \(C' \subseteq C\) such that \(|D_c| \geq n_{\text{train}} + n_{\text{test}}\) \(\forall c \in C'\)
\For {each \(c \in C'\)}
\State Compose target train set \(D_t \subseteq D_c\) with \(|D_t| = n_{\text{train}}\)
\State Compose target test set \(D_u \leftarrow D_c \setminus D_t\)
\State Divide the remaining authors \(C \setminus \{c\}\) at random into 5 equally sized sets \(C_1 \ldots C_5\)
\State Select for each author a set prototypes \(P_c \subseteq D_c\) with \(|P_c| \leq n_{\text{protos}}\)
\For {\(k \leftarrow 1 \ldots 5\)}
\State Train with target samples \(D_t\) and outlier samples \(\bigcup_{c \in C} P_c\) with \(i \neq k\)
\State Test with target samples \(D_u\) and outlier samples \(P_c\) with \(c \in C_k\)
\State Track estimated labels and posterior probabilities.
\EndFor
\EndFor
\State Compute ROC-curve and the corresponding AUC-value from the tracked information
\EndFor
\end{algorithmic}
\end{algorithm}

Algorithm 4.5: Validation procedure for authorship verification techniques.
aim of the minified code is to investigate the influence of removing all redundant characters from the source code on the classification performance.

- **Dataset B1 and B2** both consist of a cluster of repositories developed together by a team of authors. The aim of this dataset is to validate the authorship identification approaches on code that was developed collaboratively.

The remainder of this section describes the data mining process we followed for establishing these datasets.

### 4.2.1 Git and GitHub authorship concepts

First, the different authorship concepts in Git and GitHub are of interest in order to understand the rationale behind the data collection approach. Different terms such as contributor, collaborator and commiter are used and have a slightly different meaning. Figure 4.1 shows how these authorship concepts relate to each other. A GitHub user account is identified by a unique login name. The single user who has full control of a particular repository, is the *owner* of that repository. The owner can grant read and write access to other users, which are *collaborators* on the repository. A collaborator is able to commit directly to the repository. In addition, the concept *contributor* is important. A GitHub user is a contributor whenever he contributed to a project by committing to a project’s default branch or the gh-pages branch\(^2\), opening an issue or proposing a pull-request. Note that the project’s set of contributors is not necessarily equal to the set of collaborators, since contributions such as pull requests or opening issues can be done by every arbitrary GitHub user (and not only collaborators). Consequently, the code in the repository can be written by authors who are not collaborator of the project. Moreover, a contributor does not necessarily have to be author of code in the project, as he may for instance only have reported an issue. Oddly enough, we will see that in the non-ideal case, code can have been written by an author than can not be linked to a GitHub user. Consequently, it does even not hold that each author in the repository is a contributor. GitHub furthermore introduced the concept of *organizations*. Organizations are GitHub accounts that are designed for business and large open-source projects that need multiple owners and administrators. Within an organization, groups of *members* (teams) can be created and access to repositories can be controlled. The latter concepts are not shown in Figure 4.1.

In Git, a developer is represented by an *identity*, which is formed by the combination of a name and an e-mail address. Every commit includes the identity of the developer. In order to construct the labeled datasets, the identity of the commits is crucial to determine the author of the code. In principle it is possible that a single person uses multiple identities in one repository. This is the case when the developer has multiple checkouts of the project configured with a different name or e-mail address. The *committer* is not necessarily identical to the *author* of the commit. The code is written by the author and can be committed on behalf of the original author by another person. If a Git repository

\(^2\)The gh-pages branch is meant to generate a webpage for the project.
4.2. Source code collections

is hosted on GitHub, the email-address in the commit is used to link the commit to a
GitHub user. A GitHub user can add multiple email addresses to his account, so that
multiple Git identities can be traced back to one Github account. If the identity is
unknown to GitHub, the commits are not linked to any user.

4.2.2 Data collection tool

We developed an application\(^3\) that enables exploratory repository collection on GitHub
by using the GitHub API\(^4\). Figure 4.2 depicts the GUI of the application, which visualizes
the relation between GitHub developers and repositories in a developer network. We
define a developer network as a directed graph, where a directed edge \((u, v)\) is considered
to be a developer \(u\) linked to repository \(v\). The repositories and developers in such a
network displayed with blue and green vertices respectively.

We aimed to be able to determine the GitHub users that definitely contributed code
to a repository, and vice versa, the repositories a GitHub user contributed code to. The
tool determines the contributors of a specified repository by the GitHub API REST call:

GET /repos/:owner/:repo/contributors

However, as was explained, a contributor does not necessarily have authored code to
a project. Therefore the tool may associate users with a repository to which they did
not contribute code. Besides the users belonging to a project, we are interested in all
the repositories associated to a user. Unfortunately, the GitHub API does not offer
functionality to query the repositories a user contributed to. Therefore, the tool is only
able to identify the repositories a user owns or is member of. This is possible by the
REST call:

GET /users/:username/repos

---

\(^3\)https://github.com/wilcowisse/Shibboleth
\(^4\)https://developer.github.com/v3/
4. Experimental Setup

Figure 4.2: Graphical user interface of the source code mining application.

Clearly, looking up users of a project and projects of a user is asymmetric and produces data that should be validated. Therefore, the tool eventually determines the real authors of a repository by cloning a GitHub project to the local hard drive. Then the code is analyzed by employing the Git blame command, which annotates each line in the given file with information that includes the identity of the person who last modified the line. The blame information is used to assign a class label to the code in the cloned repositories, needed to construct labeled datasets. A possible improvement on our approach is not to consider the contributors to repositories, but instead to query all the commits done to a repository:

\texttt{GET /repos/:owner/:repo/commits}

The resulting commits enclose all the authors who committed to the project. Moreover, an advantage is that for each commit it is indicated to which GitHub user the committer and author of the commit are linked. In our approach the Git user is manually linked to a GitHub user by his email-address. This is however not possible, when a GitHub author has not made his email address publicly available or committed by another e-mail address. In such cases we use a heuristic that links the Git username and email-address to a Github account: we evaluate the measure of similarity between the Git and Github usernames and login name by the Jaro-Winkler string similarity metric [17]. In practice a record is linked if the string similarity is above the threshold of 0.9, which roughly implies a nearly exact match.

The application respects the API rate limit of 60 requests per hour. Using basic authentication this limit can be extended to 5000 requests per hour by supplying a
4.2. Source code collections

personal OAuth token. By using caching the number of requests are further limited. The application relies on a number of third party libraries. They include the Google HTTP Client Library used to access the REST interface of the GitHub API. To clone and analyse Git repositories, JGit\(^5\) is employed. Furthermore, the Gephi Toolkit\(^6\) is used to display the relation between repositories and developers.

4.2.3 Mining repositories

The core functionality of the data collection application involves the construction of a dataset from given GitHub repositories. What has not been discussed so far, is how appropriate projects have been discovered. For instance, collections A1 and A2 require projects which have been written by exactly one author. Furthermore, since the validation methods need multiple samples per author, developers are wanted that developed a large number of projects. GitHub does not offer search functionality satisfactory for these needs. Hence, in order to find appropriate repositories we used data from the GHTorrent project\(^3\). The GHTorrent project offers an offline mirror to a data collection that has been gathered by event streams and data from GitHub. We used the relational repository metadata that is offered in a MySQL database\(^7\). Figure 4.3 shows the database tables which are relevant for our work. The database table \texttt{project\_members} refers to users with commit access to the referenced project\(^3\), so we concluded that project members equivalent with GitHub collaborators. Again, this is not exactly the information we would like to have (i.e. we target contributions), however, it is a good approximation of a number of repositories a user contributed to. The used database contained 1.1M JavaScript repositories and users. We decided to remove projects that were forked from other projects, since those forked projects ‘inherited’ all the project members of the upstream repository. As a consequence some popular projects appeared a very large number of times in the database, with the only difference that the project had another owner. After removing forks, 361554 JavaScript repositories remained, which were imported in Gephi\(^5\). Gephi is open source software for the analysis of graph structures and networks. Using Gephi we removed all users that were member of less than 5 projects. This ensured that there were multiple samples per author. The resulting users and repositories are shown in Figure 4.4 (a).

4.2.4 Data selection

Using Gephi, appropriate users and projects were selected from the mined data to construct the eventual datasets. To construct dataset \(A\), we first queried the repositories that had one author (e.g. the repositories without other collaborators). Then we listed the corresponding users in descending order on the number of such repositories they owned. Since the GHTorrent data may be outdated we determined the currently owned

\(^5\)\url{http://www.eclipse.org/jgit} \\
\(^6\)\url{http://gephi.github.io/toolkit/} \\
\(^7\)We used the database dump from of 10-12-2013. The size of the compressed dump is approximately 4.2GB while the uncompressed size is more than 15GB.
4. Experimental Setup

Figure 4.3: GHTorrent MySQL database tables used in our study. Projects are equivalent to Github repositories.

repositories on Github by our data collection tool. Then we validated whether the owner was the only GitHub user who contributed to the owned projects. If that was the case, the repository was included in the dataset.

Dataset $B$ consists of a cluster of repositories developed by the same set of authors. This dataset was constructed by querying $k$-core networks by Gephi in the GHTorrent dataset. A $k$-core network is a subgraph in the original graph where all nodes have degree of at least $k$. Consequently a repository in such a network is developed by at least $k$ authors, and these authors collaborated in least $k$ of these repositories. Figure 4.4 (b) depicts a number of such $k$-core networks from the dataset. By manually inspecting a number of such clusters we selected two of them with the property that most contributors did a significant contribution to the project. Since a $k$-core network is a subset of a larger network, we then identified all the project members and collaborators of the repositories in the core network. A resulting developer network is shown in Figure 4.4 (d).

4.2.5 Data cleaning and preparation

After selecting the data, we created a clean subset of these data. The cloned data of dataset $A$ was 29.1GB, distributed over 826k files. Obviously, only JavaScript files are relevant, and thus other data in the repository were removed. The remaining datasize was 185.6MB. Unfortunately, we encountered that many repositories included source code of JavaScript libraries such as JQuery in their code base. Since the libraries were written by different authors, the libraries would introduce noise in the extracted features to characterize the writing style of the particular author. Thus without removing the libraries the results of the authorship analysis are in question. We first attempted to remove libraries by a predefined blacklist of file names. However, by manually inspecting the data, we found that this was not sufficient, since the file names had some variation, and obviously, we were not aware of all existing JavaScript libraries. Therefore we adopted the following strategy to eliminate libraries from the code base. The first observation we did was that in general libraries were much larger than other JavaScript files. Therefore we removed all files that were larger than 100kB (this already eliminates the JQuery libraries). The second observation was that the code in library files was committed in a single commit. Therefore, we used Git blame to determine whether all lines in a were committed in one single commit, and removed the file if this was the case. We were confident that this approach was satisfactory after manually searching for libraries in the remaining code. After removing libraries, we removed all JavaScript files that
4.2. Source code collections

![Image](4.2-source-code-collections-figure.png)

Figure 4.4: The GHTorrent dataset imported in Gephi. Figure (a) shows the users with 5 or more JavaScript repositories. Figure (b) shows the 200 authors that had the largest number of repositories without other collaborators in this dataset. Figure (c) depicts a number of $k$-core networks in the GHTorrent dataset. One of these $k$-core networks including all the collaborators of each repository is shown in figure (d).

could not be parsed. In dataset $A$ we then removed the authors which had less than 5 repositories. Finally, we removed repositories that were smaller 1kB or larger than 50kB. This was done because there was a high variance in the size of repositories. By removing the smallest and largest samples, the size of the samples are better balanced. Table 4.1 shows the size of dataset $A1$ before and after the described cleaning process.

Dataset $A2$ consists of the same repositories as dataset $A1$, but all the code was
4. Experimental Setup

<table>
<thead>
<tr>
<th></th>
<th>Before cleaning</th>
<th>After cleaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authors</td>
<td>216</td>
<td>182</td>
</tr>
<tr>
<td>Repositories</td>
<td>5846</td>
<td>4374</td>
</tr>
<tr>
<td>Files</td>
<td>15092</td>
<td>10039</td>
</tr>
<tr>
<td>Total datasize</td>
<td>185.6 MB</td>
<td>29.3 MB</td>
</tr>
</tbody>
</table>

Table 4.1: Dataset A1, before and after detecting and removing all invalid JavaScript files, authors and repositories.

Figure 4.5: Result of minifying data set A1. Figure (a) depicts the degree of compression per JavaScript file. Figure (b) depicts the empirical cumulative distribution of the repository size of dataset A2. In this graph, the dashed line corresponds to dataset A1.

Minification is done by using the Esmangle\(^8\) and Escodegen\(^9\) projects, with the `compact=true` option. Figure 4.5 shows the percentage of compression per file, and how this affected the eventual size of the samples. In dataset B we removed the edges between authors and repositories that represented a contribution of less than 1kB. Also the authors that contributed to only 1 project, and the projects with only one author were removed. The amount of code in dataset B was limited, so we manually inspected the code for JavaScript libraries. Appendix C details the graphs of the resulting authors with the corresponding size of their contributions.

\(^8\)https://github.com/Constellation/esmangle
\(^9\)https://github.com/Constellation/escodegen
5

Results

In this chapter we present the experimental results of the four different authorship analysis techniques that have been introduced in Chapter 3. The fundamental difference between the techniques under investigation can be summarized as follows:

• Broadly speaking, there are two basic classification approaches in authorship analysis: similarity and feature vector based approaches. The SCAP method is a similarity based common \( n \)-gram (CNG) technique, while the other techniques are feature vector based techniques and employ an SVM classifier.

• Next, the syntactical complexity of the used features is different. In our study we evaluate one language specific approach that uses layout, style and structure features from the AST. The other techniques use language independent character-based features. These include character \( n \)-grams and text compression features.

The evaluation procedures have been described in Section 4. A different evaluation procedure was designed to investigate the authorship identification and verification task. Correspondingly, we describe the experimental results of these two distinct authorship analysis tasks in Section 5.1 and Section 5.2.

5.1 Authorship identification

The task of authorship identification is to predict the most likely author in a set of candidate authors. We first evaluated the identification techniques on dataset A1. The samples in this dataset consist of GitHub repositories each developed by a single author. Figure 5.1 (a) and (b) provide the results obtained by validating the identification techniques by training with \( n = 7 \) and \( n = 25 \) samples per author on this dataset\(^1\). The results show that the authorship identification techniques are quite effective. In a

\(^1\)In Appendix D, the results for other values of \( n \) are presented.
Figure 5.1: The authorship identification techniques trained with 7 and 25 samples respectively, with a varying number of candidate authors. Note that in figure (b) the maximum number of authors on the x-axis is 34 in contrast to 75 authors in figure (a). This is due to the validation procedure, which selects at random a set of authors from the corpus, such that the number of samples for each author is equal to or larger than the number of required train and test samples. Table 5.1 shows that an identification problem with 50 or 75 candidate authors cannot be trained with 25 samples per author.
Table 5.1: Number of authors in the dataset that have \( n \) or more training samples. This is the result after reserving 5 samples per author for validation.

5.1.1 Comparison of the classification techniques

We now zoom in on the accuracy of the different classification techniques presented in Figure 5.1. The identification techniques are evaluated on various problem complexities by regulating the number of candidate authors and the amount of available training data per author. It is straightforward that the accuracy substantially decreases with a larger number of candidate authors. More candidate authors make it difficult to discriminate between the coding characteristics of the developers. Figure 5.1(a) shows that the accuracy declines sharply when the number of candidate authors is limited, while the decline is less fast with a larger number of candidate authors. Apparently, a part of the features in the feature space is not sufficiently unique across the authors. Those features can only be exploited when the number of candidate authors is small. With a larger number of training samples, classification is based on the features that are to a larger extent unique for each author. In that case the decline is more gradual.

Besides the number of candidate authors, the number of training samples influences the classification accuracy. A smaller number of training data allows the learning algorithm to generalize from the training data less well. An important observation in Figure 5.1 is that it roughly indicates that with a larger number of training samples, the accuracy of the machine learning classification approaches increase proportionally more than the similarity based approach (i.e. SCAP). To investigate this relationship further, we compared the influence of the number of training samples on the SCAP technique and the machine learning technique with the combination of domain specific and \( n \)-gram features. This is shown in Figure 5.2. With 7 training samples, the methods approximately perform equally well. With a small training set size, the SCAP method is substantially better. However, SCAP performs slightly worse with a large number of training samples. The most likely cause for these results is that the data sampling in the feature vector based approaches is instance based, which means that each training document is considered as a distinct unit that contributes separately to the classification model. Therefore, multiple training samples per class are required for creating a reliable model. In contrast, SCAP is profile based, which means that the differences between training documents by the same author are disregarded and that the training documents are treated cumulatively per author. Consequently, a profile can be constructed with a limited number of training samples. Stamatatos [73] pointed out that segmenting long samples into multiple parts may improve the instance based approaches. Although data segmentation may be a good way to improve the accuracy, we did not investigate it. First, the documents should be parsable for the domain specific approach. This makes it difficult to split the files at arbitrary positions. Secondly, to prevent topic biases, we strictly separated repositories in the training and test data so that no code of a project
5. Results

Figure 5.2: Figure (a) shows the accuracy of the SCAP approach (dashed line) compared to the machine learning approach with \( n \)-gram and domain specific features (solid line). In figure (b) the two approaches are compared with a varying number of train samples. The results were obtained by a set of 15 candidate authors and a training set size of 13 samples per author. To decrease the variance, we tested with a larger number of validation iterations than the results in the previous figure (100 instead of 10).

had been seen before. In the current validation procedure splitting the repositories in multiple parts would violate this condition.

5.1.2 Comparison of the feature types

Having analyzed the different classification techniques, we now focus on the different feature types that were used. The techniques in our study that use character \( n \)-gram features include machine learning classification with \( n \)-grams, and similarity based classification with \( n \)-grams (i.e. the SCAP method). As explained in the literature review in Chapter 2, character \( n \)-grams are one of the most effective feature types in authorship identification. Figure 5.1 shows that also in our case the accuracy of the \( n \)-gram based techniques achieved the best performance.

Next to the \( n \)-gram based techniques, the compression based approach is character based. However, in the experiments it turned out that the compression based method is the worst performing identification technique. By inspecting the compression distances between samples, we encountered that the differences of the values were very small. This may be due to the limited amount of the content of the document that is characteristic for the coding style of a developer. Consequently, the difference in compression distances that originates from coding style may be concealed by noise introduced by the other content in the document. For example, frequently used words or sentences in comments
5.1. Authorship identification

Figure 5.3: Results of testing different types of domain specific features. We did this by excluding feature from the feature space. We tested with 15 candidate authors and a training set size of 13. Omitting all features corresponds to a random guess, $1/15 \approx 0.067$, this is indicated with the dashed line in the bottom of the graph.

may determine the compression distance to a great extent. As a result, the contribution of the smaller portion of characters in the document that describes the layout, may be negligible.

To investigate the contribution of the individual domain specific feature types, we tested the domain specific approach by exclusively using one feature type and removing the other feature from the feature space. This is shown in Figure 5.3. In this figure ‘Expression types’ and ‘Statement types’ relate to the corresponding type of node 1-grams. From the graph above we can see that the structural features obtained a high accuracy. This supports our hypothesis that structures in the AST are effective in distinguishing developer styles. Another important observation is that the accuracy of using only layout features is higher than the accuracy of all the remaining features together (the layout excluded). The significance of the layout features is also shown in Figure 5.4. This figure shows in descending order the accuracy of all $2^{12} = 4096$ combinations of domain specific feature types listed in the bar chart in Figure 5.3 (excluding the first and third bar). All the combinations that include layout features, correspond to the first half of the graph, before the accuracy drops in the middle of the graph.

Having discussed the different feature types, we now briefly look at the normalization of the feature values. As was discussed in Chapter 3, we maintain for every feature a histogram distribution of the relative frequency of occurrences. We observed that the domain specific and n-gram based approaches achieved a better performance with binary normalization compared to representing features values by the relative frequency of occurrences. Therefore, the presented results use binary feature normalization. It is
5. Results

Figure 5.4: Accuracy of all combinations of domain specific feature types.

<table>
<thead>
<tr>
<th>Technique</th>
<th>A1</th>
<th>A2</th>
<th>Δ(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scap</td>
<td>0.77</td>
<td>0.52</td>
<td>33</td>
</tr>
<tr>
<td>Domain specific</td>
<td>0.76</td>
<td>0.51</td>
<td>33</td>
</tr>
<tr>
<td>Compression based</td>
<td>0.66</td>
<td>0.54</td>
<td>18</td>
</tr>
<tr>
<td>n-gram based</td>
<td>0.79</td>
<td>0.63</td>
<td>20</td>
</tr>
<tr>
<td>Combined</td>
<td>0.84</td>
<td>0.60</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 5.2: Relative difference in accuracy between dataset A1 and A2.

Figure 5.5: Authorship identification accuracies on dataset A1 (solid lines) and A2 (dashes lines). In these experiments 13 training samples per author were used. Table 5.2 presents the deviation in accuracy between the two datasets when using 13 training samples per author, and a set of 15 candidate authors.

difficult to explain this result, but it may be related to the dependency of each bin-value on other bins in the histogram. In the case feature-wise normalization is used, the value of each bin is relative to the values of other bins, which may have undesired effects. For instance, missing observations may cause the relative frequency of other bins to become higher. Conversely, bins with a very high number of occurrences may cause the value of bins with more scarcely observed patterns to become too small to be significant. Clearly, the absence of an observation or a high variance of a bin may introduce noise in this way. If binary normalization is used, the value of a bin indicates whether a feature is observed or not in the sample. The values are thus independent of other bin-values in this case and may consequently be more consistent across the samples. A possible drawback of binary normalization is that relevant information from the input data may be disregarded. For example, the ratio between the use of different sorts of control flow statements is ignored. Furthermore, more vector values will become 1 with an increased document size. This suggests that more sophisticated normalization may improve the classification performance. For instance, several methods exist to deal with missing feature values by estimating them from the data that are present [68].
5.1.3 The influence of source code minification

The techniques discussed so far were validated on dataset A1. This dataset contains the original source code that is developed by a single author. We addressed two problems which may deteriorate the authorship analysis techniques, namely source code minification and collaboration within projects. First, we focus on the effect of source code minification. Code minification is the process of removing all redundant characters from source code, for instance by stripping the layout and shortening variable names. To validate this dataset, we removed all layout related features from the feature space in the domain specific approach. Figure 5.5 provides the corresponding classification accuracies tested with 13 training samples, compared to dataset A1. Table 5.2 describes the relative performance reduction with an instance of 13 training samples and 15 authors. The accuracy of the authorship identification was considerably worse when validating with the minified JavaScript code in dataset A2.

The techniques that had the highest accuracies on the original code also had the highest accuracies on minified code. Surprisingly, the compression based technique seems to be the least influenced by the minification of the code, and obtained at a small number of candidate authors even a better performance than the Scap approach. This is remarkable, since the Scap method is substantially better with the original source code. The observation may be linked to the minification procedure, which makes the code more uniform by removing layout and by renaming variables. As a consequence, minification produces many re-occurring character patterns in the code, which the compression algorithm may take advantage of: the compression takes place by identifying repeated character sequences. Aspects such as the number of parameters and other coding style aspects may become better visible due to the uniformity of the code. The latter can also be exploited by the n-gram based techniques. Among the most frequent n-grams are ‘ion(a,b)’, ‘};});};’ and ‘var b=th’. Such patterns are not mixed with style characteristics and hence carry much contextual information in the code such as the number of parameters and nesting depth, etc.

5.1.4 Authorship identification within projects

As well as code minification, the collaboration of multiple authors within projects may deteriorate the authorship analysis techniques. To test this, the same authorship identification techniques were then evaluated on dataset B1 and B2, which both consist of a cluster of repositories developed together by a team of authors. Unfortunately, none of the techniques were effective on these datasets, as the accuracies did not significantly surpass the random-guess baseline. Due to time constraints we did not further investigate the cause of the negative results. First, it may be the case that it is simply too difficult to detect authors if they closely worked together. However, it may also be the case that the way the sample data is collected is not accurate. We used Git blame information to identify the lines of code written by each author. The blame information indicates who was the last one who edited the line. As a consequence, the line of code

\[\text{The minifier tries to shorten the code by renaming variables in a consistent way.}\]
may often have been introduced by another author, and later be edited by the person we considered as author. In such cases it is hard to define who is the actual author of the concerning code.

5.2 Authorship verification

In Section 5.1 we analyzed the results of the authorship identification techniques. The results of authorship verification are discussed in this section. The authorship verification task involves determining whether two documents have been written by the same author, without necessarily identifying the author. Figure 5.6 shows the ROC-curves of the verification techniques, in case the target user is trained with 7 samples. An ROC-curve describes the compromises between the true positive rate (TPR) and the false positive rate (FPR) when the decision threshold is varied [60]. Obviously, the TPR should be high, while the FPR should remain low. A better classification performance is indicated by an ROC curve that is higher to the left in the graph. By choosing a threshold value, the classifier can operate at any desired point that lies on the ROC curve. Since it is hard to compare the differences between ROC-curves, often the Area Under the Curve value (AUC-value) is taken as indication of performance. The AUC-value can be interpreted as the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance [24]. Random guessing produces a diagonal line between (0,0) and (1,1), therefore a realistic classifier should always have an AUC higher than 0.5. Figure 5.7 shows the AUC values of the authorship analysis techniques for a varying number of training samples. The corresponding ROC curves can be found in Appendix D.

5.2.1 Comparison of the classification results

We first consider the techniques applied on the original source code in dataset A1, presented in Figure 5.7 (a). The performance results of the authorship verification techniques have a number of aspects in common with the authorship identification results. First, classification with an SVM using \( n \)-gram features was the most successful technique. Furthermore, the performance of the domain specific approach was lower, but combining the domain specific feature space with the \( n \)-gram feature space resulted in a higher performance. The compression based method is also in authorship verification the least well performing technique.

The authorship verification and identification results also deviate on some essential points. In contrast to authorship identification, the Scap approach performs considerably worse compared to the machine learning approaches. As was pointed out in the previous section, the feature vector based approaches are better able to identify the authorship of unlabeled samples if much training samples are available. Now, in authorship verification, an abundant amount of code samples were available to model the outlier class, so that the outlier class can be modeled pretty well. Therefore, the performance of the instance based methods may be attributed to the well modeled outlier class. Fur-
5.2. Authorship verification

thermore, the outlier class is very diverse and heterogeneous, as it represents ‘the rest of the world’. The SVM classifier may be better able to discriminate between the two classes than the similarity measure that is used in the similarity based approaches.

We adopted two different approaches to use the similarity based Scap method in a verification context. In the first approach the outlier class is represented by a single cumulative profile of most frequent $n$-grams and the distance between the unlabeled code and the cumulative profile is used to predict the class label. In the second approach the outlier class is modeled by using multiple outlier profiles, each corresponding to a different author. Then, the minimal distance of a piece of unlabeled code to an author is taken to predict the class label. As Figure 5.7 shows, the difference between the two evaluated representations of the outlier class in the similarity based approach are minimal.

5.2.2 The influence of source code minification

The effect of code minification was also tested in the verification problem context. Figure 5.6 (b) and Figure 5.7 (b) show the verification results on the dataset with minified source code. As was the case with authorship identification, the compression based approach performs surprisingly well with minified JavaScript code. Especially with a large number of training samples the compression based technique was quite effective. In the previous section we already argued that this may be attributed to the uniformity of the source code after minification, which the compression algorithm may take advantage of.

It is remarkable that all the feature vector based approaches show a drop down in the
AUC-values at two training samples. This is particularly the case for minified code in dataset A2. These observations are reflected by the ROC curves that show an unexpected shape at two training samples (see also Appendix D). We did not investigate the origin of these unexpected results, but it may relate to the way the posterior probabilities (the probability that a predicted label is correct) are obtained from the SVM classifier. Since the SVM is not based on probability density estimates, the confidences are estimated from the distance of the samples to the decision boundary [21]. This is done by scaling the distances on the [0,1] interval by a sigmoid function, such that the likelihood of the posterior probabilities is maximized\(^3\). The peculiar shape of the ROC curve may be attributed to an inadequate estimation of the confidence intervals.

\(^3\)In PRTools this normalization is performed by the \texttt{cnormc.m} function.
The experimental results have been presented in the previous chapter. In this chapter we discuss the main findings and implications obtained by the experiments. First we compare the authorship analysis approaches in terms of the classification technique and the feature choice in Sections 6.1 and 6.2 respectively. Then we discuss the effectivity of authorship analysis techniques applied to authorship verification in Section 6.3. We also address the investigated complications in authorship analysis in Section 6.4. Subsequently, the overall effectivity of the techniques are summarized in Section 6.5. Finally, we point out the limitations and threads of validity of the current work.

6.1 Comparing classification techniques

The investigated authorship analysis techniques include one similarity based technique which employs a distance metric for classification, and three feature vector based techniques that employ an SVM for classification. The authorship identification results confirm the high performance of the similarity based technique by common n-grams that has been reported in literature [28]. We found that the similarity based technique can especially handle cases where a very limited amount of training data per author is available, while the feature vector based techniques were more effective with a large number of training samples. Different aspects may explain these observations. First, the machine learning techniques are instance based which means that each document is treated separately and a sufficient amount of documents is needed to create a representative model of the classes. Secondly, the observations may be inherent to the difference between similarity based and feature vector based classification. In the feature vector based approaches, the classes are modeled in the same feature space, where each dimension represents a single measured pattern in the source code. With a small number of training samples, we perceived that the representation of the feature vectors was sparse, because many of the features were not observed. In contrast, the similarity based approaches classify by means of similarity function that quantifies the degree of
shared information between the instances. This information can also be extracted with
a limited number of training samples, i.e. we are not tied to a predefined set of n-grams
to be observed in a document. Therefore, in the similarity based techniques a more
adequate set of information can be extracted if only a small amount of training data is
available.

In contrast to authorship identification, the feature vector based approaches turned
out to be substantially better than the similarity based techniques. This may be at-
tributed to the abundant amount of training samples that were available to model the
outlier class. Like the classes in authorship identification could be modeled well with
many training samples, the instance based techniques may create a very accurate model
of the outlier class with the large number of available training samples. The profiles
used in the similarity based method, in contrast, disregard the differences between the
repositories in the outlier class. Since the outlier class is very heterogeneous, it may be
not possible to create an accurate model in a profile based representation.

6.2 Comparing the effectivity of features types

One of the observations is that the approaches that use character n-gram features had
the best performance. A possible explanation of the good performance n-gram features
is their ability to implicitly capture much stylometric characteristics. Character n-grams
may capture many little nuances of style, which are not considered in the domain spe-
cific feature space. Therefore we presume a good classification is mainly based on small
textual differences in the code, such as frequently occurring variable and function names,
text in the comments, and small stylistic variations in the layout. This may also be an
explanation for the observation that the compression based technique is the least well
performing approach. The PPM compression algorithm compresses the text by predict-
ing the next symbol by identifying repeated character sequences in the original text. If
the author specific style information mainly lies in small stylistic variation, only a small
number of author specific symbol sequences may contribute to a good text compres-
sion. Consequently, the proportion of authorship relevant content that contributes to
the compression distance may be minimal and thus be concealed by noise introduced by
other content of the document. Small deviations such as commonly used variable names
and layout are less well captured by the domain specific document representation, which
may explain its lower accuracy.

One of the main contributions of this work is the investigation of structural features
extracted from the Abstract Syntax Tree (AST). The structural aspects were described
by node n-grams extracted from the parse tree. We showed that node 2 and 3-grams
are effective markers of the coding style of an author. However, our results agree with
the findings of other studies, which show that the layout is a more relevant indication
of authorship in source code than style and structure features [27, 19]. Furthermore,
the performance of the domain specific techniques have shown to be less effective than
the n-gram based techniques. These findings confirm the high accuracy of character n-
gram based approaches in comparison to domain specific approaches that were found in
literature as was discussed in Chapter 2. However, the results do not imply the domain specific approach is not of interest. First, we showed that the combination of n-gram and domain specific features are complementary, since they can be combined in such a way that they enhance each other. Secondly, because the layout is one of the most important markers of style, it is at the same time also the most susceptible to be changed to circumvent authorship analysis techniques [53]. In such cases, the domain specific approach may better allow to constrain the features that are used in the classification process. Additionally, some works pointed out that character n-gram based methods may be less robust in cross-topic and cross-genre conditions, since n-grams may be closely associated to a particular domain. Thus if the training data is not representative for an author, the n-gram model may capture particular topics and may perform poorly when the training and test corpora are on a different topic [74]. Further research should be done to investigate the effect of such situations in source code authorship analysis. Clearly, it is not unlikely that the domain specific approach is the preferred technique in such situations.

6.3 Applying authorship verification

The previous studies on authorship analysis of source code focussed on problems with small and closed candidate sets. However, no attention has been paid to authorship verification of source code, which is a one-versus-the-rest-of-the-world problem. Although some results indicate that verification is considerably more difficult than authorship identification [2], we obtained quite high levels of performance in the verification problem. We followed the extrinsic verification paradigm where the one-class problem is transformed to a multi-class classification problem. In the similarity based paradigm we adopted two strategies to model the outlier class, first by representing the outlier class by one single profile, and secondly by representing the outlier class by distinct profiles belonging to different authors. The difference between these two approaches turned out to be minimal. The first approach is quite similar to a recently proposed profile based approach for authorship verification in [65]. The second is more or less comparable to the impostors method proposed in [51]. However, in the latter work the adopted similarity metric is different, since the average of the distance to the outlier profiles is taken instead of the minimal distance. The mentioned similarity based authorship verification methods in natural language have shown to be very effective, e.g. the winning submission of the PAN 2013 contest was a modification of the impostors method [39]. However, the findings of the current study do not support that this is also the case in source code. As was explained above, the feature vector based approaches achieved performances that were substantially higher than the performance of the similarity based approach.

6.4 Addressing authorship analysis complications

We addressed two problems which are related to external impact on coding style, namely the impact of code minification and the impact code collaboration. Code minification has
6. Discussion

<table>
<thead>
<tr>
<th></th>
<th>Original code</th>
<th>Minified code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Few samples</strong></td>
<td>Similarity based, Scap</td>
<td>Similarity based, Scap</td>
</tr>
<tr>
<td><strong>Many samples</strong></td>
<td>Feature vector based, $n$-gram*</td>
<td>Feature vector based, $n$-gram*</td>
</tr>
</tbody>
</table>

Table 6.1: The authorship analysis techniques that obtained the highest accuracies in authorship identification. Techniques with an asterisk indicate that the accuracy can be improved by combining the technique with domain specific features.

<table>
<thead>
<tr>
<th></th>
<th>Original code</th>
<th>Minified code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Few samples</strong></td>
<td>Feature vector based, $n$-gram*</td>
<td>Compression based**</td>
</tr>
<tr>
<td><strong>Many samples</strong></td>
<td>Feature vector based, $n$-gram*</td>
<td>Compression based**</td>
</tr>
</tbody>
</table>

Table 6.2: The authorship analysis techniques that obtained the highest AUC-values in authorship verification. Techniques with an asterisk indicate that the accuracy can be improved by combining the technique with domain specific features. Two asterisks indicate that if feature vector based classification with $n$-grams and domain specific features are combined, they outperform the compression based technique.

shown to significantly deteriorate the performance of the methods, but the techniques remain quite effective. In authorship identification the $n$-gram based techniques performed the best. In authorship verification, especially the compression based techniques are robust to minified code. Also in identification, the performance of the compression base technique is the least influenced by the minification procedure. This is likely due to the large number of repeated character sequences in minified code that are the result of stripping the layout and renaming variables in a predefined way. Consequently, the proportion of authorship relevant content in minified code that contributes to a good compression may be larger than in the code that has not been not minified. Unfortunately, we were not able to verify that the identification techniques can be applied when authors collaborated in the same project. The reason for these results is an important issue for further research.

6.5 Overall effectivity of the techniques

We now have discussed a number of aspects of the experimental results. The extent to which the different authorship analysis techniques are effective depend on the problem context. In the current authorship identification results, it is mainly the number of training samples that determine the choice for a particular authorship analysis technique. In authorship verification it is mainly the minification of the code that determines the choice for a particular authorship analysis technique. These observations are summarized in Table 6.1 and Table 6.2.
6.6 Limitations and threats to validity

We made a number of assumptions to guarantee the feasibility of the approach. This may cause the results to not generalize well to source code found in a forensic investigation or in industrial practice. This section describes some threats to the validity of our experiments.

6.6.1 Data representativeness

First, there may be a selection bias in the construction of the datasets. To construct dataset A we selected those authors from the GHTorrent database, which owned a large number of JavaScript repositories. Consequently, the dataset may be mainly representative for active and experienced JavaScript developers that are active on GitHub. In addition, we assumed that it is a priori known which parts of the code were developed by a single author. If there is a number of developers of a piece of code, the whole team of developers should be identified. This has not been addressed in this study, and by our knowledge no other authors addressed this problem.

We also do not know to what extent the external influence on coding style plays a role in the classification accuracy. In practice, there are various complications which may deteriorate authorship analysis techniques:

- **Reuse of code.** Among the most serious problems is the reuse of code. The popularity of question and answer websites such as Stack Overflow\(^1\) makes that a vast amount of solutions for programming solutions can readily be copied. Reuse of programming examples is not generally seen as a serious ethical offense as is the case in plagiarism of written texts, but has been considered as a key notion in software development [4, 7]. The source code that was gathered for our experiments is existing open source code, and is in this sense representative for this problem, as some parts may have been copied. However, we do not know the influence of copied code precisely.

- **Evolution of style and adversarial attacks.** Another problem is a varying programming style of an individual programmer over time, between languages and between projects. Although a number of individual characteristics have shown to be correlated to programmers [35], the results reported in [62] indicate that typographical style can change over the lifetime of a program. Also, an adversary may intentionally change the programming style to circumvent authorship analysis techniques. This can be an *obfuscation attack* which is the attempt to write in such a way that the style will not be recognized, or an *imitation attack* which is the attempt to write in a style that is similar to another author. It has been shown that such adversarial attacks successfully can be applied in written texts [8].

- **External constraints.** The developers may also be influenced by external constraints such as naming conventions, or the code may be pretty printed by IDEs. Moreover,

\(^1\)http://stackoverflow.com
the used framework or library may also have significant influence on aspects such as the complexity of the code, the number of functions and whether the developer uses an imperative or a functional programming paradigm. Additionally, a JavaScript library such as Prototype\(^2\) enables the JavaScript developer to define classes and inheritance which are not part of the genuine JavaScript language. It is not unlikely that such frameworks highly change the programming style of the programmer.

To mitigate the last two complications we defined the samples in the training and validation data to be entire repositories. As a result the projects used for training and validation are strictly separated, such that they may be written in a different context or domain. However, further investigations should give insight into the severity of the mentioned complications.

Subsequently, we should note that the minification of JavaScript code strips a lot of authorship information from the code, but that code minification as it was applied in our investigation is not representative for an adversarial attack or for code formatting conventions imposed by an IDE. In these cases, not all the code is modified in the same way, like we did. For instance, an author may use different IDEs, which format the code in different ways. Equivalently, an adversary may intentionally use different ways to change the layout of the code. Therefore, to investigate such attempts, the techniques should be trained and evaluated with distinct coding formats and different naming conventions used by the same author.

6.6.2 Limitations of the techniques and the validation procedure

The evaluated techniques have some limitations. The character based approaches that were adopted from literature need to be tuned in a better way. First, the PPM compression algorithm used in the compression based technique is especially successful at the compression of natural language text [40]. In source code, other compression algorithms may be more appropriate. Next, we have not considered tuning the optimal \(n\)-gram length in the \(n\)-gram based approaches. Also we did not investigate the optimal profile length \((L)\) in the SCAP technique. Some authors suggest that the truncation of the author profiles, as it was originally proposed, can be safely ignored [12]. However, we have our doubts whether this is generally advisable. Other research pointed out that in cross-topic attribution low frequency \(n\)-grams should be avoided, since they are closely related to the context of the training data [74]. This may also be also the case in source code.

We now consider some limitations of the domain specific authorship technique which is based on an AST language representation. The major drawback of this approach relates to the way the feature space was established. Therefore we suggest some ways to improve this technique:

- In the domain specific approach, the feature values are splitted into a number of histogram intervals. Literature shows that the particular choice for bin intervals

\[^2\text{http://prototypejs.org}\]
6.6. Limitations and threats to validity

can improve the classification [70]. Therefore, optimizing the discretization of the interval ranges may be a way to improve the performance. Furthermore, in the current feature representation, the boundaries of the histogram intervals are non-overlapping. This may conceal the relation between the feature values. For instance, consider the case that candidate authors $a_1$ and $a_2$ use variable names of character length 1–2 and 10–12 respectively. Then, from the current feature representation it can not be derived that a third author that used variable length 3–4 is most similar to author $a_1$, since the bins are considered as distinct features by the SVM classifier.

- The experimental results showed the high impact of layout related features. However, we only counted the number of spaces, tabs and carriage returns at each layout slot. As a result, the order of these characters was not tracked. Describing the layout by character $n$-grams may be a better way, since they also capture the lexical order of the characters.

Finally, the accuracy performance measure that was used to evaluate the performance of the identification problem may be not the adequate one in a forensic investigation. Some studies considered the ranking of authors by likelihood, rather than only identifying the most likely author, e.g. [63]. A ranked result would help to obtain the confidence of the authorship of a document for each individual author in the investigation.
Conclusions and Future Work

The main aim of this thesis has been to investigate different authorship analysis techniques on JavaScript, and to develop a domain specific technique that is based on the AST program representation. In this chapter we summarize the main results obtained in this study and suggest some future research directions.

7.1 Contributions

The findings from this study make several contributions to the current literature:

- Most previous contributions on source code authorship analysis targeted programming compiled languages. In contrast, JavaScript is an interpreted language, which means that no pre-runtime translation takes place and the source code is executed without translating the source code to machine-language. Consequently, source code is more likely available than the compiled languages that were under investigation in other studies. Moreover, JavaScript is widely used on the Web. Hence, suspect code can likely be found in practice.

- We investigated different classification techniques. The present study confirms several findings of previous studies that compared similarity based and machine learning authorship analysis techniques. We showed the effectiveness of n-gram based techniques and compared similarity and feature vector based techniques.

- Authorship identification has been addressed in a number of previous studies. However, little attention has been paid to authorship verification, which we addressed in our study.

- Unlike natural languages, the definition of programming languages is much more formal, and can unambiguously be parsed. Therefore, we proposed to represent the source by an Abstract Syntax Tree prior to feature extraction. A document representation by an AST enables to extract structural features from the parse
tree easily. By our knowledge no studies have considered such structures in the parse tree.

- By means of JavaScript code minification we evaluated how the classification accuracy is affected by removing authorship information from the source code.
- We determined for four different authorship analysis techniques what was the most accurate technique with original and minified code, and with a different amount of training data.

7.2 Research results

The present study was designed to compare four different JavaScript source code authorship analysis techniques. The investigated authorship identification techniques include one similarity based technique, where classification is done by a similarity metric, and three feature vector based techniques that employ an SVM for classification. The similarity based technique is based on character $n$-gram features. The feature vector based techniques consist of a character $n$-gram based technique, a compression based technique, and a domain specific technique which is based on an AST source code representation.

In the introduction we defined five research questions that would bring us closer to an answer on the main research question. We now review each of these questions.

Research Question 1. What is the state of the art in software authorship analysis?

We addressed this research question in Chapter 2 by reviewing the literature. In summary, we explored similarity and feature vector based classification techniques in literature, and investigated different feature types that have been used in the classification process. In general, the works that use features from a low language abstraction level in the form of $n$-grams, have shown to achieve the highest accuracies.

Research Question 2. What is the difference in performance between similarity based and feature vector based techniques, by considering the number of candidate authors and the number of available training samples per author?

The approaches that use character $n$-gram features achieved the best performance in either authorship identification and verification. The similarity based technique can especially handle cases where a very limited amount of training data per author is available, while the feature vector based techniques that were based on character $n$-grams and domain specific features were slightly more effective with a large number of training samples. We have not encountered striking differences between the techniques related to the number of candidate authors in the classification problem.

Research Question 3. What is the effectivity of domain specific JavaScript features compared to character based features?
7.3 Main conclusions

The techniques that used character n-gram features achieved a higher performance than the domain specific approach. However, we showed that the combination of n-gram and domain specific features is complementary, resulting in a higher performance. The compression based technique turned out to be the least effective authorship analysis technique. The investigation of the domain specific approach has shown that the layout features were the strongest markers of writing style. Also structures in the parse tree have shown to be effective markers of the coding style of an author.

Research Question 4. What is the effectiveness of authorship analysis techniques in authorship verification compared to authorship identification?

The character n-gram based techniques obtained the highest accuracies in authorship verification as was also the case in authorship identification. In contrast to authorship identification, the feature vector based approaches turned out to be substantially better than the similarity based approach.

Research Question 5. What is the influence of code collaboration and code minification on the classification performance?

Code minification has shown to significantly deteriorate the performance of the authorship analysis methods. Still, the effectiveness of the techniques is reasonable. Especially the compression based technique is robust against code minification. In authorship verification, the compression based technique was the second best technique with minified code, following the combination of domain specific and n-gram based features. The current study was unable to analyse authorship identification within projects. This is an important issue for further research.

7.3 Main conclusions

This work has demonstrated that JavaScript source code of unknown authorship can effectively be attributed to the respective developer. The main research question of our research relates to the effectiveness of the analysis techniques on JavaScript source code:

Main Research Question. To what extent are authorship analysis techniques effective on JavaScript source code?

The results indicate that an adequate coding style signature of JavaScript developers can be constructed if a sufficient amount of sample code is available. The extent to which the different authorship analysis techniques are effective depend on the problem context, and can be summarized as follows:

- In authorship identification, the n-gram based techniques obtain the highest accuracies. If a limited number of training samples are available the similarity based technique is better, while with a large number of training samples the feature vector based technique is better. In the latter case, the domain specific features can help to improve the classification accuracy. With minified code the same techniques are effective.
7. Conclusions and Future Work

- In authorship verification a distinction should be made between original and minified JavaScript code when considering which techniques are most effective. In the original code, the \( n \)-gram based techniques obtained the highest AUC-values. With minified code, the compression based technique was the best. However, if feature types are combined, the combination of character \( n \)-gram and domain specific features achieve an even higher performance.

7.4 Future work

The research presented in this thesis has raised several questions that provide the basis for further research. First, the techniques evaluated in our study should be validated on datasets that are more representative for malicious source code, as our dataset consisted of open source software. Next, the similarity based technique we evaluated is also the only technique that uses a profile based data sampling. The similarity based technique of Burrows [11] that was presented in Section 2.2.1 is instance based. A comparison between the presented techniques and this technique would give more insight in the difference between similarity and feature vector based classification. Moreover, the variability in the validation results should be investigated. To reduce variability, we averaged the results over multiple rounds. Further research is needed to determine the origin of the variability. For instance, what code is hard to identify, and what is the influence of the used frameworks on the classification accuracy? Additionally, in Chapter 6 we indicated a number limitations of the authorship analysis techniques. Addressing these limitations could be a good way to improve the performance of the current techniques. Especially the domain specific technique, that was developed by us, provides several opportunities for future research. Since little work has done in tree structures in the AST of source code, the selected tree structure features were defined in a rather intuitive way. There is a large volume of published studies that consider the classification of tree patterns [43]. Although the current study shows the effectivity of using structures in the parse tree, more thorough structural features could be investigated. Finally, a domain specific authorship identification technique could especially be of interest in a cross language setting. In this case, the syntax definitions of the languages can be quite different, while the underlying structures may remain stable per author over different languages. Further research regarding cross-language authorship analysis in source code would be worthwhile.
Bibliography


71


[46] Sangkyum Kim, Hyungsul Kim, Tim Weninger, Jiawei Han, and Hyun Duk Kim. Authorship classification: a discriminative syntactic tree mining approach. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 455–464. ACM, 2011.


[56] Jonas Lindh Morén. The application of closed frequent subtrees to authorship attribution. 2014.

[57] Stephen G MacDonell and Andrew R Gray. Software forensics applied to the task of discriminating between program authors. 2001.


Bibliography


Abstract Syntax of JavaScript

A.1 Syntax definition

This section describes JavaScript its abstract syntax used for feature generation. The abstract grammar follows the abstact syntax made available in the Mozilla SpiderMonkey parser API. In the used notation, the left hand side is the node-type. The right hand side describes the possible child node types that are related to the node by the node attributes (in cursive). A list of child nodes are indicated by an asterix (*). Furthermore, the possible types of child nodes are separated by a vertical bar (|). An optional child-node is indicated by an empty node, the NullNode.

\[
\begin{align*}
\text{Program} & \rightarrow \text{body: Statement}* \\
\text{BlockStatement} & \rightarrow \text{body: Statement}* \\
\text{ExpressionStatement} & \rightarrow \text{expression: Expression} \\
\text{IfStatement} & \rightarrow \text{test: Expression} \\
& \quad \text{consequent: Statement} \\
& \quad \text{alternate: Statement | NullNode} \\
\text{LabeledStatement} & \rightarrow \text{label: Identifier} \\
& \quad \text{body: Statement} \\
\text{BreakStatement} & \rightarrow \text{label: Identifier | NullNode} \\
\text{ContinueStatement} & \rightarrow \text{label: Identifier | NullNode} \\
\text{WithStatement} & \rightarrow \text{object: Expression} \\
& \quad \text{body: Statement} \\
\text{SwitchStatement} & \rightarrow \text{discriminant: Expression} \\
& \quad \text{cases: SwitchCase*} \\
\text{SwitchCase} & \rightarrow \text{test: Expression | NullNode} \\
& \quad \text{consequent: Statement*}
\end{align*}
\]

1https://developer.mozilla.org/en/SpiderMonkey/Parser_API
A. Abstract Syntax of JavaScript

<table>
<thead>
<tr>
<th>Statement</th>
<th>→ argument: Expression</th>
<th>NullNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReturnStatement</td>
<td>→ argument: Expression</td>
<td></td>
</tr>
<tr>
<td>ThrowStatement</td>
<td>→ block: BlockStatement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>handler: CatchClause</td>
<td>NullNode</td>
</tr>
<tr>
<td></td>
<td>finalizer: BlockStatement</td>
<td>NullNode</td>
</tr>
<tr>
<td>TryStatement</td>
<td>→ param: Identifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body: BlockStatement</td>
<td></td>
</tr>
<tr>
<td>CatchClause</td>
<td>→ test: Expression</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body: Statement</td>
<td></td>
</tr>
<tr>
<td>WhileStatement</td>
<td>→ init: VariableDeclaration</td>
<td>Expression</td>
</tr>
<tr>
<td></td>
<td>test: Expression</td>
<td>NullNode</td>
</tr>
<tr>
<td></td>
<td>update: Expression</td>
<td>NullNode</td>
</tr>
<tr>
<td></td>
<td>body: Statement</td>
<td></td>
</tr>
<tr>
<td>DoWhileStatement</td>
<td>→ left: VariableDeclaration</td>
<td>Expression</td>
</tr>
<tr>
<td></td>
<td>right: Expression</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body: Statement</td>
<td></td>
</tr>
<tr>
<td>ForStatement</td>
<td>→ id: Identifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>params: Identifier*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body: BlockStatement</td>
<td>Expression</td>
</tr>
<tr>
<td>VariableDeclaration</td>
<td>→ declaration: VariableDeclarator*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>kind: &quot;var&quot;</td>
<td>&quot;let&quot;</td>
</tr>
<tr>
<td>VariableDeclarator</td>
<td>→ id: Identifier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>init: Expression</td>
<td>NullNode</td>
</tr>
<tr>
<td>ArrayExpression</td>
<td>→ elements: (Expression</td>
<td>NullNode)*</td>
</tr>
<tr>
<td>ObjectExpression</td>
<td>→ properties: Property*</td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>→ key: Literal</td>
<td>Identifier</td>
</tr>
<tr>
<td></td>
<td>value: Expression</td>
<td>GetSetFunctionExpression</td>
</tr>
<tr>
<td>FunctionExpression</td>
<td>→ id: Identifier</td>
<td>NullNode</td>
</tr>
<tr>
<td></td>
<td>params: Identifier*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body: BlockStatement</td>
<td>Expression</td>
</tr>
<tr>
<td>ArrowFunctionExpression</td>
<td>→ params: Identifier*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>body: BlockStatement</td>
<td>Expression</td>
</tr>
<tr>
<td>SequenceExpression</td>
<td>→ expressions: Expression*</td>
<td></td>
</tr>
<tr>
<td>UnaryExpression</td>
<td>→ operator: UnaryOperator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>prefix: boolean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>argument: Expression</td>
<td></td>
</tr>
<tr>
<td>AssignmentExpression</td>
<td>→ operator: AssignmentOperator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>left: Expression</td>
<td></td>
</tr>
<tr>
<td></td>
<td>right: Expression</td>
<td></td>
</tr>
</tbody>
</table>
A.1. Syntax definition

UpdateExpression → operator: UpdateOperator
prefix: true | false
argument: Expression

LogicalExpression → operator: LogicalOperator
left: Expression
right: Expression

ConditionalExpression → test: Expression
alternate: Expression
consequent: Expression

NewExpression → callee: Expression
arguments: Expression*

CallExpression → callee: Expression
arguments: Expression*

MemberExpression → object: Expression
property: Expression

Identifier → name: string

Literal → value: string | boolean | null | number
raw: string

The following right hand side values are abbreviations and are defined as:

Statement = EmptyStatement | BlockStatement | ExpressionStatement | IfStatement | LabeledStatement | BreakStatement | ContinueStatement |
WithStatement | SwitchStatement | ReturnStatement | ThrowStatement | TryStatement | WhileStatement | DoWhileStatement | ForStatement | ForInStatement | ForOfStatement | LetStatement |
DebuggerStatement | FunctionDeclaration | VariableDeclaration

Expression = ThisExpression | ArrayExpression | ObjectExpression | FunctionExpression |
ArrowExpression | SequenceExpression | UnaryExpression | BinaryExpression | AssignmentExpression | UpdateExpression | LogicalExpression | ConditionalExpression | NewExpression | CallExpression | MemberExpression | (YieldExpression) | (ComprehensionExpression) |
(GeneratorExpression) | (GraphExpression) | (GraphIndexExpression) |
(LetExpression) | Identifier | Literal

UnaryOperator = "-" | "+" | "!" | "~" | "typeof" | "void" | "delete"

BinaryOperator = "==" | ";=" | ";==" | ";===" | ";<" | ";<=" | ";>" | ";>=" | ";<<" |
">>" | ";>>" | ";>>>
"|" | ";+/+" | ";*+/+" | ";/+/+" | ";%+/+" | ";&+/+" | ";^+/+" |
"instanceof" | ";.."

LogicalOperator = ";|||" | ";&&" |
AssignmentOperator = ";++" | ";-"
A. Abstract Syntax of JavaScript

A.2 Refactorings to the AST

The following list shows the most important refactorings performed on the original Mozilla parse tree.

- The MemberExpression node has the boolean attribute `computed`, which represents a member accessed by an identifier or by an expression. In the latter case `computed` is true. We represent computed member expressions by the node type `CMemberExpression`, with associated layout slots.

- Nested parenthesis in expressions do not appear in the original parse tree. We refactored the tree such that an expression in a nested parenthesis is described in a BracketExpression. We also defined layout slots in this node type.

- Empty attributes are described by the `null` value. Instead, we represented this by a node of the type `NullNode`.

- The operator in an UnaryExpression, BinaryExpression, LogicalExpression, AssignmentExpression and UpdateExpression is converted from a textual value (e.g. `*`) to a node with a type of the operator.

- The variable kind (var, const and let) in a VariableDeclaration is represented by a node.

- A property in an object initialization can also be a function acting as getter or setter such as in the following code fragment:

  ```javascript
  var person = {
    firstName: undefined,
    lastName: undefined,
    get fullName(){
      ...
    },
    set fullName(name) {
      ...
    }
  }
  ```

  We represented this by a `GetProperty` or `SetProperty` respectively. Furthermore, the function declaration for the getters and setters deviate from normal function definitions in the source code, so we represented these as as `GetSetFunctionExpression`.

- Multiple catch clauses in a try statement are SpiderMonkey-specific. Therefore we refactored it in order to contain only one catch clause. Also, some statements and expressions are SpiderMonkey-specific, such as the let statement and array comprehension expressions. Furthermore, JavaScript 1.7 introduced destructuring assignment and binding forms, which had not been implemented in Esprima at time of writing of this work. Altogether, the follow node types in the Mozilla-AST are not
considered: LetStatement, ArrowExpression, YieldExpression, ComprehensionExpression, GeneratorExpression, GraphExpression, LetExpression, ObjectPattern and ArrayPattern.

A.3 Layout nodes

The Layout nodes are added as children to the related nodes in the AST. In the following table the introduced nodes are listed. In this table, the subscripted indices indicate locations where the developer may have inserted spacing. We call these locations layout slots. In case a node has a list of child nodes (e.g. between function arguments) with layout in between, a list of layout nodes is generated. This is indicated by a superscript asterix.

<table>
<thead>
<tr>
<th>Node</th>
<th>Layout slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>bodystructor</td>
</tr>
<tr>
<td>BlockStatement</td>
<td>{bodystructor}_2</td>
</tr>
<tr>
<td>IfStatement</td>
<td>if_1(teststructor)_3 consequent</td>
</tr>
<tr>
<td>WithStatement</td>
<td>with_1(objectstructor)_3 body</td>
</tr>
<tr>
<td>TryStatement</td>
<td>try_1 blockistrator handler</td>
</tr>
<tr>
<td>CatchClause</td>
<td>catch_1(paramstructor)_3 body</td>
</tr>
<tr>
<td>WhileStatement</td>
<td>while_1(teststructor)_3 body</td>
</tr>
<tr>
<td>DoWhileStatement</td>
<td>do_1 bodyistrator while_2(teststructor)_3 body</td>
</tr>
<tr>
<td>ForStatement</td>
<td>for_1(initstructor;_5 teststructor;_5 updatstructor)_3 body</td>
</tr>
<tr>
<td>ForInStatement</td>
<td>for_1(leftistrator)_3 in_4 righistrator_5 body</td>
</tr>
<tr>
<td>FunctionDeclaration</td>
<td>function_1(idstructor)_3 {paramsstructor}_4 body</td>
</tr>
<tr>
<td>VariableDeclaration</td>
<td>kindstructor{declarator_structor}_3 arguments</td>
</tr>
<tr>
<td>ObjectExpression</td>
<td>{propertiesstructor}_2</td>
</tr>
<tr>
<td>Property</td>
<td>key_1;_2 value</td>
</tr>
<tr>
<td>ArrayExpression</td>
<td>{elementsstructor}_2</td>
</tr>
<tr>
<td>FunctionExpression</td>
<td>function_1(idstructor)_3 {paramsstructor}_4 body</td>
</tr>
<tr>
<td>UnaryExpression</td>
<td>operator_1 argument</td>
</tr>
<tr>
<td>BinaryExpression</td>
<td>left_1;_2 operator_2 right</td>
</tr>
<tr>
<td>AssignmentExpression</td>
<td>left_1;_2 operator_2 right</td>
</tr>
<tr>
<td>LogicalExpression</td>
<td>left_1;_2 operator_2 right</td>
</tr>
<tr>
<td>ConditionalExpression</td>
<td>testpositor?_2 consequotor_3;_4 alternate</td>
</tr>
<tr>
<td>CallExpression</td>
<td>calleeutor_1 {argumentsstructor}_3</td>
</tr>
<tr>
<td>MemberExpression</td>
<td>objectutor_1;_2 property</td>
</tr>
<tr>
<td>CMemberExpression</td>
<td>objectutor_1;_2 propertystructor_3</td>
</tr>
<tr>
<td>SequenceExpression</td>
<td>expressionsstructor</td>
</tr>
<tr>
<td>SwitchStatement</td>
<td>switchutor_1 {discriminator_structor}_3 {casesstructor}_6</td>
</tr>
</tbody>
</table>
### A. Abstract Syntax of JavaScript

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>BracketExpression</td>
<td>( (\text{value}_{2}) )</td>
</tr>
<tr>
<td>ReturnStatement</td>
<td>( \text{return}<em>{1}\text{argument}</em>{2} )</td>
</tr>
<tr>
<td>ThrowStatement</td>
<td>( \text{throw}<em>{1}\text{argument}</em>{2} )</td>
</tr>
<tr>
<td>UpdateExpression</td>
<td>( \text{operator}_{1}\text{argument} )</td>
</tr>
<tr>
<td>VariableDeclarator</td>
<td>( \text{id}_{1} = \text{init} )</td>
</tr>
<tr>
<td>SwitchCase</td>
<td>( \text{case}<em>{1}\text{test}</em>{2} : \text{consequent} )</td>
</tr>
<tr>
<td>NewExpression</td>
<td>( \text{new}<em>{1}\text{callee}</em>{2}(\text{arguments}_{4}) )</td>
</tr>
</tbody>
</table>
In this appendix the domain specific features obtained from the AST are listed. For every feature we describe which nodes are considered in this feature. Then we examine the attribute value of the selected node and increment the number of observations of the histogram category that applies.

B.1 Layout

By means of the *value* attribute in the introduced layout nodes, we identified for each feature the frequency of using layout. More precisely, we track for each feature the frequency that no layout was used, or that zero, one, two or many spaces, tabs, or carriage returns were used. We defined the layout features shown in the following table. The numbers refer to the defined layout nodes (see next).

In addition, the ExpressionStatement, DebuggerStatement, ContinueStatement, BreakStatement and LabeledStatement are optionally closed by a semicolon. Therefore we track the use of semicolons and other spacing after these statement types.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Layout between statements</td>
<td>Program.<em>, BlockStatement.</em></td>
</tr>
<tr>
<td>2 Layout behind opening curly bracket ({} in block</td>
<td>BlockStatement.1, SwitchStatement.1</td>
</tr>
<tr>
<td>3 Layout before opening curly bracket ({} in block</td>
<td>BlockStatement.2, SwitchStatement.5</td>
</tr>
<tr>
<td>4 Layout between keyword/id and opening parenthesis</td>
<td>IfStatement.1, WithStatement.1, CatchClause.1, WhileStatement.1, DoWhileStatement.3, ForStatement.1, ForInStatement.1, SwitchStatement.1, FunctionDeclaration.2, FunctionExpression.2, CallExpression.1, NewExpression.2</td>
</tr>
</tbody>
</table>
### B. Domain Specific Features

<table>
<thead>
<tr>
<th>Layout</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Layout before closing parenthesis</td>
</tr>
<tr>
<td>7</td>
<td>Layout between closing parenthesis and body</td>
</tr>
<tr>
<td>8</td>
<td>Layout before semicolon in for-statement</td>
</tr>
<tr>
<td>9</td>
<td>Layout behind semicolon in for-statement</td>
</tr>
<tr>
<td>10</td>
<td>Layout before comma in function parameters</td>
</tr>
<tr>
<td>11</td>
<td>Layout behind comma in function parameters</td>
</tr>
<tr>
<td>12</td>
<td>Layout before comma in function/object arguments</td>
</tr>
<tr>
<td>13</td>
<td>Layout behind comma in function/object arguments</td>
</tr>
<tr>
<td>14</td>
<td>Layout before comma in variable declarations</td>
</tr>
<tr>
<td>15</td>
<td>Layout behind comma in variable declarations</td>
</tr>
<tr>
<td>16</td>
<td>Layout before comma in Object and Array Expressions</td>
</tr>
<tr>
<td>17</td>
<td>Layout behind comma in Object and Array Expressions</td>
</tr>
<tr>
<td>18</td>
<td>Layout before properties c.q. elements in Object and Array Expressions</td>
</tr>
<tr>
<td>19</td>
<td>Layout behind properties c.q. elements in Object and Array Expressions</td>
</tr>
<tr>
<td>20</td>
<td>Layout before operators</td>
</tr>
</tbody>
</table>
B.2. String patterns

For the following features we track the frequency of matching regular expressions in the attribute value of the selected nodes.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
<th>Attribute</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>27 First character in identifiers related to</td>
<td>NewExpression.callee, CallExpression.callee,</td>
<td>name</td>
<td>/^[A-Z]/, /^[a-z]/, /^[^A-Za-z]/</td>
</tr>
<tr>
<td>function identifiers</td>
<td>FunctionExpression.id, FunctionDeclaration.id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28 First character in identifiers related to</td>
<td>WhileStatement.test, IfStatement.test,</td>
<td>name</td>
<td>/^[A-Z]/, /^[a-z]/, /^[^A-Za-z]/</td>
</tr>
<tr>
<td>objects</td>
<td>ForStatement.init, ReturnStatement.argument,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VariableDeclaration.id,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FunctionDeclaration.params,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CallExpression.arguments,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NewExpression.arguments,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FunctionExpression.params and every expression except CallExpression.callee,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NewExpression.callee, FunctionExpression.id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29 Number of capitals in identifiers</td>
<td>Identifier</td>
<td>name</td>
<td>/^[A-Z]/g matching 0,1,2,3,4,5,6,8,9,∞ times</td>
</tr>
<tr>
<td>30 Number of non-letter characters in</td>
<td>Identifier</td>
<td>name</td>
<td>/^[^A-Za-z]/g matching 0,1,2,∞ times</td>
</tr>
<tr>
<td>identifiers</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### B. Domain Specific Features

#### 31 Literal types

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
<th>Attribute</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal raw String with double quotes, string with single quotes, null, number or regex.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### B.3 Comments

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
<th>Attribute</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 The distribution of the parent nodes of block comments</td>
<td>BlockComment parent</td>
<td>ArrayExpression, ObjectExpression, FunctionExpression, CallExpression, MemberExpression, BlockStatement, ExpressionStatement, SwitchStatement, FunctionDeclaration, VariableDeclaration, SwitchCase, Program and ’Otherwise’</td>
<td></td>
</tr>
<tr>
<td>33 The distribution of the parent nodes of line comments</td>
<td>LineComment parent</td>
<td>ArrayExpression, ObjectExpression, FunctionExpression, BinaryExpression, LogicalExpression, ConditionalExpression, NewExpression, CallExpression, MemberExpression, BracketExpression, BlockStatement, ExpressionStatement, IfStatement, SwitchStatement, ReturnStatement, ThrowStatement, FunctionDeclaration, VariableDeclaration, SwitchCase, Program and ’Otherwise’.</td>
<td></td>
</tr>
<tr>
<td>34 The ratio between line and block comments</td>
<td>BlockComment</td>
<td>type</td>
<td>BlockComment, LineComment</td>
</tr>
<tr>
<td>35 The length of block comments</td>
<td>BlockComment value</td>
<td>0, 8, 16, 32, 64, 128, 256, 512, 1024,</td>
<td></td>
</tr>
<tr>
<td>36 The length of line comments</td>
<td>LineComment value</td>
<td>0, 8, 16, 32, 64, 128, 256, 512,</td>
<td></td>
</tr>
</tbody>
</table>

#### B.4 String length

For the following features we track the length of the given attribute.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
<th>Attribute</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>37 Length of identifiers</td>
<td>Identifier name</td>
<td>0, 1, 2, 3, 5, 8, 13, 21, 34, 55,</td>
<td></td>
</tr>
</tbody>
</table>
### B.5 List length

For the following features we track the length of the list in the attribute value of the selected nodes.

<table>
<thead>
<tr>
<th>Description</th>
<th>Selected nodes</th>
<th>Attribute</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>38 Number of parameters</td>
<td><code>FunctionDeclaration.params</code></td>
<td>params</td>
<td>0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ∞</td>
</tr>
<tr>
<td>39 Number of parameters</td>
<td><code>FunctionExpression.params</code></td>
<td>params</td>
<td>0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ∞</td>
</tr>
<tr>
<td>40 Number of statements</td>
<td><code>FunctionDeclaration.body</code> / <code>BlockStatement</code></td>
<td>body</td>
<td>0, 1, 2, 4, 8, 16, 32, 64, 128, ∞</td>
</tr>
<tr>
<td>41 Number of statements</td>
<td><code>FunctionExpression.body</code> / <code>BlockStatement</code></td>
<td>body</td>
<td>0, 1, 2, 4, 8, 16, 32, 64, 128, ∞</td>
</tr>
<tr>
<td>42 Number of statements in</td>
<td><code>WhileStatement.body</code> / <code>DoWhileStatement.body</code></td>
<td>body</td>
<td>0, 1, 2, 4, 8, 16, 32, 64, 128, ∞</td>
</tr>
<tr>
<td>43 Number of arguments in</td>
<td><code>NewExpression.arguments</code></td>
<td>arguments</td>
<td>0, 1, 2, 3, 5, 8, ∞</td>
</tr>
<tr>
<td>44 Number of arguments in</td>
<td><code>VariableDeclarator.id</code></td>
<td>raw</td>
<td>0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ∞</td>
</tr>
<tr>
<td>45 Number of arguments in</td>
<td><code>WhileStatement.test</code> / <code>IfStatement.test</code></td>
<td>raw</td>
<td>0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ∞</td>
</tr>
<tr>
<td>46 Number of arguments in</td>
<td><code>Identifier.name</code></td>
<td>raw</td>
<td>0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ∞</td>
</tr>
</tbody>
</table>

Note: The categories include values ranging from 0 to infinity (∞), indicating that the length can be any non-negative integer.
## B. Domain Specific Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>CallExpression arguments</td>
<td>0, 1, 2, 3, 5, 8, ∞</td>
</tr>
<tr>
<td>48</td>
<td>ArrayExpression elements</td>
<td>0, 1, 2, 4, 8, 16, 32, 64, 128, ∞</td>
</tr>
<tr>
<td>49</td>
<td>ObjectExpression properties</td>
<td>0, 1, 2, 4, 8, 16, 32, 64, 128, ∞</td>
</tr>
<tr>
<td>50</td>
<td>VariableDeclaration declarations</td>
<td>0, 1, 4, ∞</td>
</tr>
</tbody>
</table>

### B.6 Descendant count

For every measurement we use the following distribution: 0, 1, 2, 4, 8, 16, 32, 64, 128, ∞.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Selected nodes</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>51 Test in conditional expressions</td>
<td>IfStatement</td>
<td>WhileStatement</td>
</tr>
<tr>
<td>52 Call argument</td>
<td>CallExpression</td>
<td>arguments</td>
</tr>
<tr>
<td>53 Array element</td>
<td>ArrayExpression</td>
<td>elements</td>
</tr>
<tr>
<td>54 Return argument</td>
<td>ReturnStatement</td>
<td>argument</td>
</tr>
<tr>
<td>55 Property in object expressions</td>
<td>ObjectExpression</td>
<td>properties</td>
</tr>
<tr>
<td>56 Left hand side assignment expression</td>
<td>AssignmentExpression</td>
<td>left</td>
</tr>
<tr>
<td>57 Right hands side assignment expression</td>
<td>AssignmentExpression</td>
<td>right</td>
</tr>
<tr>
<td>58 Object of member expression</td>
<td>MemberExpression</td>
<td>object</td>
</tr>
<tr>
<td>59 Property of member expression</td>
<td>MemberExpression</td>
<td>property</td>
</tr>
<tr>
<td>60 Callee of new expression</td>
<td>NewExpression</td>
<td>callee</td>
</tr>
<tr>
<td>61 Initialization value of variable declarator</td>
<td>VariableDeclarator</td>
<td>init</td>
</tr>
</tbody>
</table>
Dataset Description

In this appendix we describe the datasets used for validating the proposed authorship analysis techniques.

C.1 Dataset A

Table C.1 describes the size of repositories, the file size within these repositories, and the size of the developers, before and after cleaning dataset A. Figure C.1 depicts the empirical cumulative distribution of repository and author sizes in this dataset.

C.2 Dataset B

Dataset B consists of two clusters of developers that collaboratively developed a number of software projects. Figure C.2 details these networks and the contributions of the

<table>
<thead>
<tr>
<th></th>
<th>Min.</th>
<th>Max.</th>
<th>Mean</th>
<th>Median</th>
<th>Std.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Before cleaning</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author size (no. repos)</td>
<td>1</td>
<td>208</td>
<td>27.1</td>
<td>21</td>
<td>27.3</td>
</tr>
<tr>
<td>Repository size (kB)</td>
<td>23 \times 10^{-3}</td>
<td>15.8 \times 10^3</td>
<td>31.7</td>
<td>3.0</td>
<td>299.4</td>
</tr>
<tr>
<td>File size (kB)</td>
<td>7 \times 10^{-3}</td>
<td>9318.1</td>
<td>12.3</td>
<td>1.8</td>
<td>126.5</td>
</tr>
<tr>
<td><strong>After cleaning</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author size (repos)</td>
<td>5</td>
<td>167</td>
<td>24.0</td>
<td>17.5</td>
<td>22.2</td>
</tr>
<tr>
<td>Author size (kB)</td>
<td>10.8</td>
<td>761.5</td>
<td>160.9</td>
<td>138.9</td>
<td>131.4</td>
</tr>
<tr>
<td>Author size (LOC)</td>
<td>339</td>
<td>31347</td>
<td>5867</td>
<td>5020</td>
<td>4878.7</td>
</tr>
<tr>
<td>File size (kB)</td>
<td>13 \times 10^{-3}</td>
<td>47.5</td>
<td>2.9</td>
<td>1.6</td>
<td>4.1</td>
</tr>
<tr>
<td>Repository size (kB)</td>
<td>1.0</td>
<td>49.4</td>
<td>6.7</td>
<td>3.8</td>
<td>7.7</td>
</tr>
<tr>
<td>Repository size (LOC)</td>
<td>7</td>
<td>2614</td>
<td>244.1</td>
<td>148.0</td>
<td>271.0</td>
</tr>
<tr>
<td>Repository size (files)</td>
<td>1</td>
<td>32</td>
<td>2.29</td>
<td>1.00</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Table C.1: The author, repository and file size in dataset A, before and after cleaning the data.
C. Dataset Description

Figure C.1: Figure (a) shows the cumulative density function of the number of files per repository and the size in kB of repositories in dataset A. Figure (b) and (c) show the of author sizes. Dashed lines correspond to the uncleaned dataset.

authors to the projects. The size of the projects are in kB. The edges represent the contribution of an author (a green node) to a project (a blue node) in kB.
C.2. Dataset B

Figure C.2: The size of repositories, authors and contributions in respectively dataset C and D.
This Appendix contains the results of the performed experiments. Figure D.1, D.2, D.3 and D.4 show the accuracy of the evaluated authorship identification techniques on datasets A1 and A2, with a varying number of training examples and number of authors. Figure D.5 shows the accuracy when the features of the domain specific and n-gram based approach were combined. In these figures $n$ refers to the number of training examples. Further, all approaches are compared to the SCAP approach, which is indicated with a dashed line. The ROC-curves of the performed authorship identification tests are shown in Figure D.6 (dataset A1) and Figure D.7 (dataset A2). In these plots, the SCAP method is tested with either one cumulative outlier profile, and with one profile per author.

Figure D.1: Accuracy of the SCAP approach on dataset A1 and A2.
D. Graphs of Experimental Results

Figure D.2: Accuracy of domain specific approach on dataset A1 and A2.

Figure D.3: Accuracy of compression based approach on dataset A1 and A2.
Figure D.4: Accuracy of the $n$-gram based approach on dataset A1 and A2.

Figure D.5: Accuracy of combined result of $n$-grams and domain specific features on dataset A and B with various number of training examples.
Figure D.6: ROC curves dataset A1.
Figure D.7: ROC curves dataset A2.