View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Murray State University

Murray State's Digital Commons

Faculty & Staff Research and Creative Activity

8-2019

ASAP: A Source Code Authorship Program

Matthew F. Tennyson PhD

Murray State University, mtennyson@murraystate.edu

Follow this and additional works at: https://digitalcommons.murraystate.edu/faculty

Part of the Computer Engineering Commons

Recommended Citation

This is a preprint manuscript of an article published by Springer Berlin Heidelberg. The Version of Record of this manuscript has been published and is available in Volume 21, Issue 4 of International Journal on Software Tools for Technology Transfer (ISSN: 1433-2787), March 20, 2019. https://doi.org/10.1007/s10009-019-00517-3

This Peer Reviewed/Refereed Publication is brought to you for free and open access by Murray State's Digital Commons. It has been accepted for inclusion in Faculty & Staff Research and Creative Activity by an authorized administrator of Murray State's Digital Commons. For more information, please contact msu.digitalcommons@murraystate.edu.

Noname manuscript No.
(will be inserted by the editor)

ASAP: A Source Code Authorship Program

Matthew F. Tennyson

Received: date / Accepted: date

Abstract Source code authorship attribution is the task of determining who wrote a computer program, based on its source code, usually when the author is either unknown or under dispute. Areas where this can be applied include software forensics, cases of software copyright infringement, and detecting plagiarism. Numerous methods of source code authorship attribution have been proposed and studied. However, there are no known easily-accessible and user-friendly programs that perform this task. Instead, researchers typically develop software in an ad hoc manner for use in their studies, and the software is rarely made publicly available. In this paper, we present a software tool called ASAP (**A** **S**ource Code **A**uthorship **P**rogram), which is suitable to be used by either the layperson or the expert. An author can be attributed to individual documents one at a time, or complex authorship attribution experiments can easily be performed on large datasets. In this paper, the interface and implementation of the ASAP tool is presented, and the tool is validated by using it to replicate previously-published authorship attribution experiments.

Keywords authorship attribution · source code · software forensics · plagiarism detection · software copyright infringement · similarity search · information retrieval · machine learning

M. Tennyson
Murray State University
Murray, KY, USA 42071
Tel.: 270-809-6217
E-mail: mtennyson@murraystate.edu

1 INTRODUCTION AND BACKGROUND

Authorship attribution is simply "the task of deciding who wrote a document" [1]. It is applied often to natural language documents. Phraseology and stylistic features such as word usage, word frequency, word length, blend usage, n-grams, etc., are used to determine the style in which a document is written. Documents of known authorship are used for training, and the training results are then used to attribute authors to documents whose author isn't known. Zhao and Zobel [1] provide a review and comparison of many methods of authorship attribution of natural language documents.

If authorship attribution is simply deciding who wrote a document, then "source code authorship attribution" is the task of deciding who wrote a document containing source code. Numerous methods of source code authorship attribution have been proposed [2–14], and many of these methods have been further studied, improved, and compared [15–20]. A comprehensive review of these methods is omitted here for the sake of brevity, but the reader is encouraged to consult the references as desired or needed. Authorship attribution of object code is also possible, but much more difficult and less reliable. Hendrikse [21] provides a detailed study of different methods of authorship attribution of object code and the effects of obfuscation on such methods.

This paper presents ASAP (A Source Code Authorship Program), which is a tool specifically for source code authorship attribution. It is capable of performing "one-off" authorship tasks, in which the user simply needs to determine the authorship of a single document. It is also capable of performing batch authorship tasks, in which the user has an entire collection of documents to attribute. Finally, it is capable of performing complex authorship attribution experiments, in which the user

identifies a directory that contains the data repository over which a k-fold cross validation or a leave-one-out cross validation experiment can be performed. Each of these use cases will be discussed later in this paper.

There are no other known tools whose sole and specific purpose is to perform source code authorship attribution tasks. The ASAP tool implements two state-of-the-art methods [22] of source code authorship attribution: SCAP [2] and Burrows [3]. There are no other known tools that implement these specific methods. The ASAP tool is meant to be accessible, but powerful, suitable for both the expert and the layperson.

There are tools that perform authorship attribution of natural language documents, such as NeoNeuro (neoneuro.com) and JStylo [23]. These tools analyze documents by searching for and extracting stylistic features like those previously mentioned (word usage, word frequency, etc.) to try to attribute authors to documents of unknown authorship. Unlike ASAP, both of these tools are meant to attribute authors to natural language documents rather than source code. NeoNeuro is a proprietary commercial product whose underlying algorithms are largely unknown. While JStylo is open-source, it is meant purely for attribution of naturual language documents. Its analyses are based on features such as sentence length, word choice, and grammatical structure, most of which would not be applicable in a source code scenario. Furthermore, the end purpose of the JStylo tool is not authorship attribution. Rather, the primary purpose is to *circumvent* authorship attribution. Indeed, the end goal of the authors of the JStylo tool was to protect the anonymity of authors by creating a tool that would provide advice to authors on how to change their writing style to thwart authorship attribution attempts [23].

There are also tools that perform general machine learning and data mining tasks, such as scikit-learn [24] and Weka [25]. These tools provide a vast array of functionality including classification, clustering, regression, and visualization. They implement algorithms such as support vector machines (SVM), k-nearest neighbors (knn), random forests, C4.5 decision trees, spectral clustering, and myriad other machine learning algorithms. These tools can be powerful. However, they are also excessive and inaccessible to the non-expert. The input data must be put into a proprietary format, and described as a finite set of features. The user must not only know and specify which general approach to use (e.g., classification, regression, or clustering), but the specific algorithm must also be specified (e.g., naive Bayes, multi-layer perceptron, or Hoeffding tree). These tools are meant to be very broad in their potential applications, and they are targeted specifically for experts and researchers.

The task of detecting copied code, sometimes referred to as clone code detection, is similar but distinct from source code authorship attribution. Detecting when students copy their programs in a classroom setting, for example, can be approached by determining whether any two programs within a collection of submitted programs are similar to each other, where all of the programs in the collection are supposed to have addressed the same problem. In other words, the programs being compared are supposed to be functionally equivalent or at least functionally very similar to each other. While related, this is different from the problem of source code authorship attribution, which is to determine whether a program is stylistically similar to programs known to have been written by a particular author, where the programs being compared address entirely different problems. The programs being compared, from a functional standpoint, are completely unrelated. There are many tools that are meant to detect copied code in a programming classroom, including JPlag [26] and MOSS [27].

Tools also exist that are meant to detect plagiarism, such as SNITCH [28] and Turnitin (turnitin.com). However, these tools are meant primarily for natural language documents rather than source code. Again, while this is certainly a related problem, it is not equivalent to the problem of authorship attribution. The goal in this other case is not to determine the author of a document. Rather, the goal is to identify snippets of text within a document that are similar to snippets of text that can be found elsewhere in other documents.

Therefore, we believe the ASAP tool is unique. Its scope is limited to authorship attribution of source code. It is suitable for both experts and non-experts. Unlike sophisticated machine learning workbenches, it requires no training or preparation to use. It is user-friendly and accessible. It takes plain source files as input, rather than proprietary data files. No data cleaning, preprocessing, or feature extraction is required prior to performing authorship tasks. It implements two state-of-the-art methods, SCAP and Burrows, that are not implemented in any other off-the-shelf, ready-to-use tool.

Although the ASAP tool currently incorporates only these two methods, it was designed and implemented to be extensible so that additional methods could be added in the future. Because the tool has incorporated the SCAP and Burrows method specifically, these methods will be briefly described here.

1.1 SCAP Method

In the SCAP method [2] of source code authorship attribution, a profile is created for every author that is a candidate to have written the program of unknown authorship, which will henceforth be known as the query program. The author profile is known as the Source Code Author Profile (SCAP). To determine the author of a query program, the program is compared using a similarity measure known as SPI to all of the available author profiles. The author whose profile is most similar to the query program is attributed to be its author.

In the SCAP method, both the programs and the author profiles are represented as n-grams. The n-grams are encoded at the byte level, which means that every byte contained in the source file is included in the n-grams, even hidden control characters. The scheme is completely language-agnostic; nothing in the SCAP method relies on features of the programming language used to write the source code.

An author profile is created by concatenating together all of the programs written by that author. The concatenated programs are represented as byte-level n-grams. The frequency of each n-gram is stored in a table. It is this table of frequencies that becomes the profile for that author. Only the L-most frequently occurring n-grams are retained in the table, so that L is referred to as the profile length.

The Simplified Profile Intersection (SPI) is the similarity measure used to compare a query program to the author profiles. The SPI is simply the number of n-grams that an author profile and a program have in common:

$$|P_A \cap P_P| \tag{1}$$

where P_A represents the author profile and P_P represents the program profile (i.e., the set of n-grams that occur in that program). So, it is ultimately the author who often uses the n-grams that appear in the query program that is attributed to be its author.

1.2 Burrows Method

The general approach used in the Burrows method [3] is quite similar to the one used in the SCAP method. To determine the author of a program, that program is considered to be a query. The query program is compared using a similarity measure to all of the programs in the dataset. The author of the most-similar program is considered the author of the query program. So, in essence, it is the author who wrote the program that is most similar to the query program that is attributed to be the author. Note that this approach can be clearly distinguished from the SCAP approach in that author profiles are not used at all. The query program is compared to each program in the dataset one at a time. The author of the program that is found to be most similar is then attributed to be the author of the query program.

The Burrows method also uses n-grams to represent programs. However, those n-grams are not byte-level. Rather, they are token-based and very much language-specific. Features that are considered significant are selected for each programming language. Programming language features such as keywords, identifiers, whitespace, literal values, and operators are used. Programs are scanned, only those tokens deemed as significant are retained, and those tokens are used to create n-grams.

The Burrows method uses the Okapi BM25 similarity metric [29]. This metric was selected after it was determined to be the most effective through empirical testing. The metric, used in some search engines, is meant to calculate the likelihood that a document is relevant to the information need expressed in the query. The metric is calculated as follows [3, 29]:

$$Okapi(Q, D_d) = \sum_{t \in Q} w_t \frac{(k_1+1) f_{d,t}}{K + f_{d,t}} \frac{(k_3+1) f_{q,t}}{k_3 + f_{q,t}} \tag{2}$$

$$w_t = \ln \frac{N - f_t + 0.5}{f_t + 0.5} \tag{3}$$

$$K = k_1 \Big((1-b) + \frac{b W_d}{W_D} \Big) \tag{4}$$

where Q is the query document, D_d is the document that the query is being compared to, t is a term in the query that also appears in the document, N is the number of total documents in the collection, W_d is the document length, W_D is the average document length in the collection, f_t is the frequency of the term within the collection, $f_{d,t}$ is the frequency of the term within the document, and $f_{q,t}$ is the frequency of the term within the query. The recommended values for the parameters are $k_1 = 1.2$, $k_3 = 1000$, and $b = 0.75$. In the Burrows method, the recommended values for k_1 and b are used, while k_3 is set to 10^{10}.

2 ASAP USER INTERFACE

The ASAP tool is organized into two parts: a front-end and a back-end. The front-end is a GUI that is meant to be user-friendly and accessible. The back-end is a command-line interface that performs all of the actual processing. The GUI is meant to provide an easy-to-use interface for all of the fundamental authorship

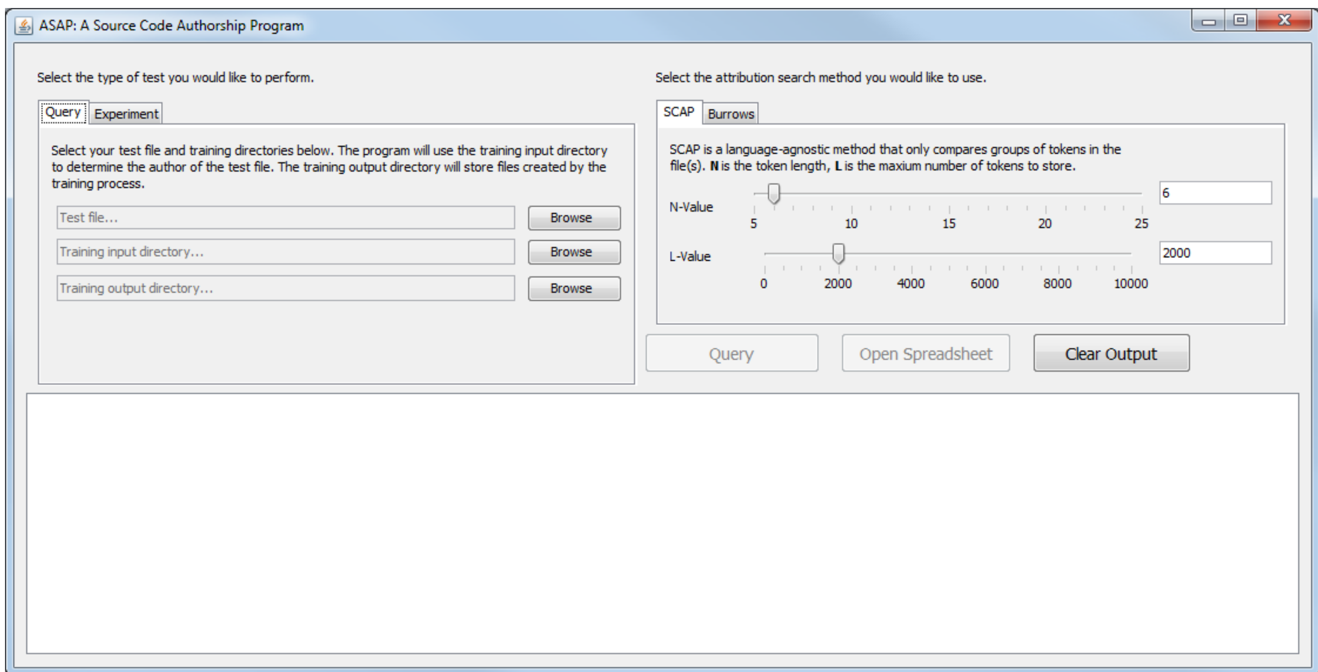

Fig. 1 The ASAP GUI

attribution activities that a user might want to perform. However, if the GUI is found to be insufficient for whatever reason, then the GUI can be bypassed and the command-line interface can be called directly. This could be useful, for example, if the user wishes to create a batch script in order to easily and repeatedly process a sequence of commands that might be specific to that user's purpose.

This section will discuss the ASAP tool from a user's perspective, and will be organized into three parts. The first part will discuss the front-end GUI. The second part will discuss the back-end command-line interface. The third part will discuss installation issues. A subsequent section will discuss the tool's software from a design and implementation perspective.

2.1 The Front-End

The ASAP GUI is shown in Fig. 1. The GUI is partitioned into three main areas. The first area, in the upper-left-hand position of the window, allows the user to select the type of query or experiment to execute. The second area, in the upper-right-hand position of the window, allows the user to pick which method of authorship attribution to execute (either SCAP or Burrows) along with any parameters that are specific to whichever method is picked. The third area, in the bottom of the window, is a text area where the output is displayed.

2.1.1 Use Case #1: Single document query

If the user wishes to attribute an author to a single document, then the "Query" tab should be selected, as shown in Fig. 1. In this case, there are three additional selections to be made: the query document, the training directory, and the output directory.

The query document is the source file whose author is presumably unknown. It is this file to which an author will be attributed.

The training directory should contain samples from all of the candidate authors. Inside the training directory, there should be a folder for each candidate author, and samples from each author should be placed inside those folders respectively. For example, say we have three candidate authors: Alice, Bob, and Carol. The training directory should respectively have three folders called Alice, Bob, and Carol; and all of the known work from each author should be stored inside each of those respective folders.

The output directory should be an empty folder where the output files will be stored. Any intermediate files that are generated, as necessitated by the attribution method, will also be stored in this folder. For example, in the SCAP method, a profile for each author will be created. Those profiles will be stored inside the specified output folder.

Once the query selections have been made, the user will click the Query button in order to finally execute the query. The tool will generate a command-line state-

ment based on the selections made by the user, which will be sent to the back-end to be executed. As the query executes, a verbose read-out of the results will be displayed in the text area at the bottom of the window. The final line of output will indicate which author was attributed to have written the query document. Resulting data will also be saved to a spreadsheet. By clicking the Open Spreadsheet button, the spreadsheet will open allowing the user to view it.

The output from an example query is shown in Fig. 2. In this example, a document known to have been written by AuthorA is selected as the query document. There are three candidate authors: AuthorA, AuthorB, and AuthorC. The SCAP method is used to perform the query. The output text area shows the training phase, where the author profiles are created. Then it shows output from the query phase, where the query document is compared to each author profile. Finally, it indicates that AuthorA is indeed the matching author.

After the query has completed execution, the Open Spreadsheet button will become active. If the user clicks this button, then a spreadsheet will open containing data that was collected from the query. The spreadsheet generated by the example query is shown in Fig. 3. Each candidate author is listed, along with the author's similarity score. (Remember that the similarity score indicates how similar the query document is to the respective author's profile.) In this case, the query document was most similar to AuthorA's profile, therefore AuthorA was attributed to be its author.

2.1.2 Use Case #2: Default split experiment

If the user wishes to attribute an author to many documents all at once or if the user wishes to perform an authorship attribution experiment where the data has already been segmented into training data and testing data, then the "default split" experiment should be chosen. For this option, the test directory and training data must be specified. Both directories should be organized in the same manner as described in the previous subsection. That is, each directory should contain folders that correspond to the candidate authors. Each candidate author's folder will contain samples written by that author. The samples contained inside the specified training directory will be used for training, while the samples contained inside the specified test directory will be used for testing. Once those selections have been made, the user will click the Experiment button in order to finally execute the experiment.

Once the Experiment button has been clicked, the experiment will proceed. Training will first be performed using the specified training data. Then each document in the test folder will be attributed and checked for correctness. The overall accuracy will be reported both as a ratio (e.g. "3 out of 3 files were correctly attributed") and as a percentage. Experiment data will also be collected and stored in a spreadsheet, which can be viewed by clicking the Open Spreadsheet button. (Note that if this option is being used to attribute an author to several documents at once rather than to perform an experiment, then the reported accuracy can be ignored. Each document will be attributed an author, which is all that will be relevant to the user.)

An example execution of a "default split" experiment is shown in Fig. 4. In the output text area, the user can follow the progress of the experiment as it proceeds. Fig. 5 shows the corresponding spreadsheet that is generated. Each test document is listed, showing the actual author of the document along with the attributed author. Finally, the attribution accuracy is given.

2.1.3 Use Case #3: K-fold cross validation experiment

In a k-fold cross validation experiment, only a single dataset is specified. The dataset folder should be organized as usual: each candidate author should have a folder, where that author's samples are stored. A k value is also specified. The k value specifies the number of segments (or "folds") that the data should be split into. Each of the k segments is then used, in turn, as the test data while the remaining folds are used for training. For example, say we choose three folds, then the data will be randomly split into three segments. The first segment will be used for testing while the remaining two folds will be used for training. Then the second segment will be used for testing with the remaining two folds used for training. Finally the third segment will be used for testing with the other two used for training. The accuracy will be measured as a percentage of documents correctly attributed across all the folds. The point of cross validation is to maximize the number of test cases, while avoiding overfitting, in order to estimate the general accuracy of the method.

After the dataset folder and the k value are selected, the user will click the Experiment button, and the experiment will proceed. As usual, the output text area will show the results as the experiment proceeds. The accuracy will be reported for each individual segment, and finally the overall accuracy will be displayed. The data will also be collected into a spreadsheet, which can be opened by clicking the Open Spreadsheet button.

Fig. 6 and Fig. 7 show spreadsheet data from an example execution, where there are five candidate authors and three folds. The first tab of the spreadsheet

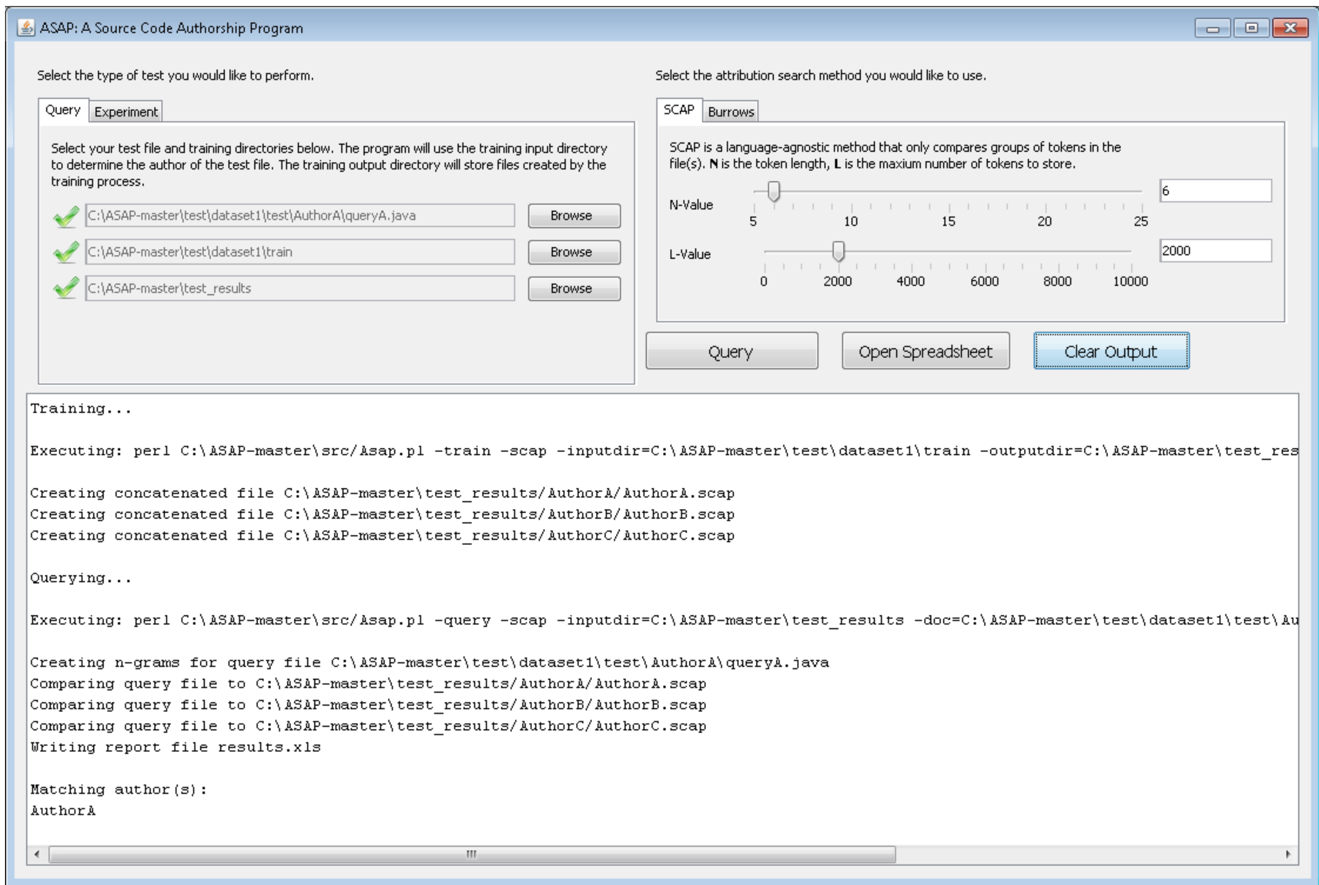

Fig. 2 Output from an example query

	A	B
1	**Author**	**Score**
2	AuthorA	108
3	AuthorB	87
4	AuthorC	54
5		

Fig. 3 Spreadsheet data from an example query

contains the overall aggregate results across all the folds (see Fig. 7), and the test results of each fold is reported in its own tab (see Fig. 6).

In this example, the overall accuracy was 90.6% with 29 file out of 32 correctly attributed. The attribution accuracy of each individual author in the dataset is also reported. (See Fig. 6.) For each fold, each file in the fold is listed, along with the actual author of the document as well as the similarity score for each author. The highest similarity score for each file is highlighted. In this example, ten out of eleven files in the fold were correctly attributed. The single file that was incorrectly attributed barely missed. The highest similarity score was 107, while the correct author had a similarity score of 105. (See Fig. 6.)

2.1.4 Use Case #4: Leave-one-out cross validation experiment

A leave-one-out cross validation experiment is basically a special case of the k-fold cross validation, where the k value is equal to the total number of documents in the dataset. In other words, each file is individually selected (one at a time) for testing while all the remaining files are used for training. This maximizes both the size of the training set for each query as well as the total number of test cases, while still avoiding overfitting.

In this type of experiment, the user only has to specify the dataset folder. The results are reported in the output text area as the experiment proceeds with the overall accuracy finally reported at the end. The results are also compiled in a spreadsheet, which can be opened by clicking the Open Spreadsheet button. The first tab of the spreadsheet will contain the overall aggregate results across the entire dataset, and the test results of each individual author will be reported in its own tab. The format of the spreadsheet is not dissimilar to that

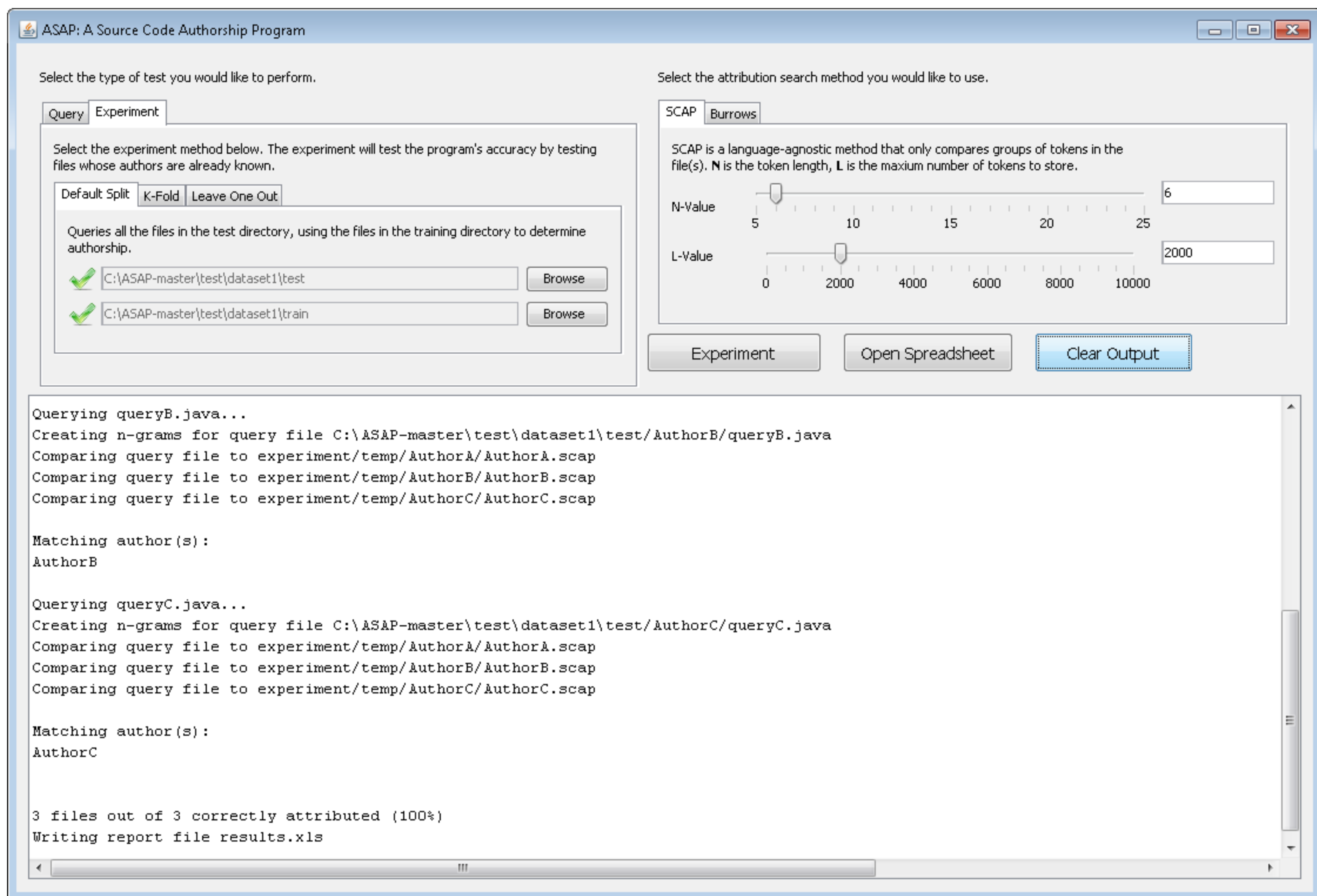

Fig. 4 An example execution of a "default split" experiment

	A	B	C	D
1	File Name	True Author	Attributed Author	Correctly Attributed?
2	queryA.java	AuthorA	AuthorA	TRUE
3	queryB.java	AuthorB	AuthorB	TRUE
4	queryC.java	AuthorC	AuthorC	TRUE
5				100%

Fig. 5 Spreadsheet data from an example "default split" experiment

of the k-fold cross validation, an example of which is shown in Fig. 6 and Fig. 7.

2.1.5 The built-in authorship attribution methods

As previously noted, the ASAP tool contains two built-in methods of authorship attribution: SCAP and Burrows. (See Sect. 1.1 and Sect. 1.2 for an overview of these methods.) In the upper-right-hand position of the GUI, the user is allowed to pick which method of authorship attribution to use (either SCAP or Burrows) along with any parameters that are specific to the selected method. The user selects the method by simply clicking on the corresponding tab, and whichever tab is active when the query is executed will be the method used.

When a tab is selected, the parameters associated with that method of attribution can be specified. Both the SCAP and Burrows methods utilize n-gram representations of programs. Therefore, for both methods, the value for n must be selected. In SCAP, the n value represents the number of bytes contained in each n-gram. In Burrows, the n value represents the number of tokens contained in each n-gram.

In the SCAP method, the parameter L represents the profile length. Recall that the number of times each n-gram is used by an author is stored in a table, and it is this table that becomes the profile for that author. But only the L-most frequently occurring n-grams are retained in the table, such that L is referred to as the profile length.

In the Burrows method, only features of a programming language that are considered significant are used in the n-gram representation of programs. Features are used such as keywords, identifiers, literal values, and operators. The features that are considered significant are dependent upon the programming language, and determining which features are considered to be significant has been a topic of research [17]. For that reason,

File	Author	Bronson	Carrano	Dale	Deitel	Drozdek	Correct?
ApproxExponential.java	Bronson	250	79	132	140	157	1
ArrayArgument.java	Bronson	105	62	98	107	93	0
BinarySearchTree.java	Carrano	58	347	90	150	144	1
BinaryTree.java	Carrano	57	316	65	94	141	1
ClientForLoop.java	Dale	56	74	536	112	104	1
ColorDemo.java	Dale	70	30	124	72	75	1
WildcardTest.java	Deitel	115	108	156	1149	123	1
ZeroFlagTest.java	Deitel	70	62	70	773	54	1
IOmethods.java	Drozdek	51	76	83	92	216	1
Library.java	Drozdek	105	156	140	187	366	1
Queens.java	Drozdek	84	70	83	93	214	1

TOTAL | Fold 1 | Fold 2 | Fold 3

Fig. 6 Data from a single fold of an example k-fold cross validation experiment

Author	NumFiles	NumCorrect	Percentage
Bronson	5	4	80.0%
Carrano	6	5	83.3%
Dale	7	6	85.7%
Deitel	6	6	100.0%
Drozdek	8	8	100.0%
TOTAL	32	29	90.6%

TOTAL | Fold 1 | Fold 2 | Fold 3

Fig. 7 Aggregate data from an example k-fold cross validation experiment

the user is allowed to select which features to use. The list of features should be stored in a plain text file, and that file is then selected in the GUI. Two feature files are distributed with the ASAP tool: one set of features for the Java programming language and another set of features for C++. These files can be freely modified by the user, or the user is free to select a different file altogether containing an entirely different set of features.

2.2 The Back-End

The ASAP back-end is a command-line interface, implemented in Perl. The main Perl script that is to be executed is *Asap.pl*, so to launch the application the following command must be issued:

```
perl Asap.pl
```

Without any additional flags the program's help text will be displayed. The first line of help text is as follows:

```
Mode not specified (-train|-query|-
    experiment).
```

This means that the ASAP back-end requires a flag indicating the mode of operation: train, query, or experiment. Without this mode-of-operation flag, the program cannot proceed. Additional flag options will vary, depending upon which mode the program is operating in. In other words, the options available in training mode will be different than the options available in query mode, and those options will be different than the options available in experiment mode.

Typically, the user will want to perform a single document query or perform an experiment. In the case of a single document query, training must be performed first. So, to do a single document query, the user will first run in training mode, and then execute the query in query mode. Additional queries can be executed without re-training, so long as the training data does not change. If the training data does indeed change, then the user must re-execute in training mode before any additional queries can be made. In the case of an experiment, training does not have to be done separately. The timing and nature of training will vary based on the parameters of the experiment itself. Therefore, any necessary training is automatically performed as needed while the experiment itself is executing.

2.2.1 Training Mode

The script is executed in Training Mode as follows:

```
perl Asap.pl -train (-scap|-burrows) (-
    inputdir=<directory>) (-outputdir=<
    directory>) [-n=<natural number>] [-
    tokenfile=<file>]
```

The user must select the method of authorship attribution being used:

```
-scap: Use SCAP method
-burrows: Use Burrows method
```

The user must select the input directory containing the training data, and the output directory where any intermediate and output files will be stored:

```
-inputdir=<directory>: Location of training
    data
-outputdir=<directory>: Location to store
    output files
```

The n-gram size and token file location only need to be specified when using the Burrows method only (note that n-grams are not generated during the training phase of the SCAP method, so the n-gram size is not necessary during training when using SCAP):

```
-n=<natural number>: Specifies n-gram size
-tokenfile=<file>: Specifies the file
    containing the list of feature tokens
```

Here is an example of a training command when using the SCAP method:

```
perl Asap.pl -train -scap -inputdir=
    my_input_data -outputdir=train_results
```

Here is an example of a training command when using the Burrows method:

```
perl Asap.pl -train -burrows -inputdir=
    my_input_data -outputdir=train_results -
    n=6 -tokenfile=JavaTokens.txt
```

2.2.2 Query Mode

The script is executed in Query Mode as follows:

```
perl Asap.pl -query (-scap|-burrows) (-
    inputdir=<directory>) (-doc=<file>) [-
    report=<file>] [-n=<natural number>] [-L
    =<natural number>]
```

First, the user must select the method of authorship attribution being used. Next, the user must select the directory containing the training results. (Note that the query input folder will be the same as the training output folder. In other words, the output of the training phase becomes the input to the query phase.) The query document must also be specified:

```
-inputdir=<directory>: Location of files
    created during training
-doc=<file>: Query document whose author is
    to be attributed
```

If an Excel spreadsheet is desired, then the name and location of the spreadsheet must be specified. If no spreadsheet is desired, then this flag should simply be omitted:

```
-report=<file>: Specifies Excel file to
    store results (optional)
```

The n-gram size and profile length only need to be specified when using the SCAP method only (if using Burrows, the n-gram size and token file were specified previously during training and do not need to be repeated for the query):

```
-n=<natural number>: Specifies n-gram size
-L=<natural number>: Specifies profile
    length
```

Here is an example of a query command when using the SCAP method, where a spreadsheet is not created:

```
perl Asap.pl -query -scap -inputdir=
    train_results -doc=example.java -n=6 -L
    =2000
```

Here is an example of a training command when using the Burrows method, where a spreadsheet is created:

```
perl Asap.pl -query -burrows -inputdir=
    train_results -doc=example.java -report=
    results.xls
```

2.2.3 Experiment Mode

The script is executed in Experiment Mode as follows:

```
perl Asap.pl -experiment (-scap|-burrows)
    [...]
```

First, the user must select the method of authorship attribution being used. If performing a **default split** experiment, then the test directory and training directory must be specified:

```
-testdir=<directory>: Location of testing
    data
-trainingdir=<directory>: Location of
    training data
```

If performing a **k-fold cross validation** experiment, then the dataset directory and k value must be specified:

```
-inputdir=<directory>: Location of dataset
-k=<natural number>: Number of folds
```

If performing a **leave-one-out cross validation** experiment, then only the dataset directory must be specified (and the k value should simply be omitted):

```
-inputdir=<directory>: Location of dataset
```

Finally, the parameters required by the authorship attribution method must be specified. The n value must be specified for both SCAP and Burrows. The profile length L must be specified only if using SCAP, while the token file must be specified only if using Burrows.

Here is an example of a "default split" experiment command using the SCAP method:

```
perl Asap.pl -experiment -scap -testdir=
    my_test_data -trainingdir=
    my_training_data -n=6 -L=2000 -report=
    results.xls -verbose
```

Here is an example of a k-fold cross validation experiment command using the Burrows method:

```
perl Asap.pl -experiment -burrows -inputdir=
    my_dataset -k=3 -n=6 -tokenfile=
    JavaTokens.txt -report=results.xls
      verbose
```

Here is an example of a leave-one-out cross validation experiment command using the SCAP method:

```
perl Asap.pl -experiment -scap -inputdir=
    my_dataset -n=6 -L=2000 -report=results.
    xls -verbose
```

2.3 Installation

The ASAP front-end is Java-based and the back-end is Perl-based. Therefore, the Java Runtime Environment (JRE) and a Perl interpreter must be installed in order to use it. (Note that the ASAP tool itself is operating system independent. As long as both Java and Perl are installed on the system, then the ASAP tool should work.) A JRE is typically installed on most systems by default, so the user will likely not have to install it. If, however, it does indeed need to be installed, then it can be obtained at Oracle's Java Download page:

```
http://www.oracle.com/technetwork/java/
    javase/downloads/
```

If the user has not already installed a Perl interpreter, then one will likely need to be installed. We recommend ActivePerl, which is free to install and use for non-commercial use. ActivePerl can be downloaded at ActiveState's Perl Download page:

```
https://www.activestate.com/activeperl/
    downloads/
```

After Perl itself has been installed, several Perl libraries that are used by ASAP will also need to be installed. To do this, open a command prompt. (In Windows, this can be done by typing "cmd" into the Windows search field.) At the command prompt the following commands will need to be executed:

```
PPM install dmake
cpan App::cpanminus
cpanm Spreadsheet::WriteExcel
cpanm -f Text::Ngrams
cpanm -f FFI::Raw
cpan IPC::Cmd
```

After Java and Perl have been installed, the ASAP tool itself will need to be installed. The installation is trivial. In fact, it is entirely self-contained, so no true installation is required at all. The user merely needs to download the ASAP folder to any location on the target computer. The ASAP folder can be obtained and downloaded from the following GitHub repository:

```
https://github.com/ASAP-Project/ASAP
```

Once the folder has been downloaded, the GUI can be launched by clicking on the *ASAP.jar* file inside the *src* folder. The command-line back-end can be executed by running the *Asap.pl* script, which can also be found inside the *src* folder. Please refer to the previous sections of this paper for detailed information about running both the GUI and the command-line interface.

3 SOFTWARE DESIGN AND IMPLEMENTATION

The ASAP front-end was implemented using the Java programming language, and the ASAP back-end was implemented using Perl. These languages were chosen for several reasons. Primarily, they are both interpreted languages, and the Java interpreter is installed on most systems by default. As interpreted languages, they are system independent. Any system with a Java and Perl interpreter installed can run the ASAP tool. In this section, the software design and implementation of both the front-end and back-end will be discussed.

3.1 Perl Back-end

The back-end of the ASAP tool does essentially all of the work. All of the processing is done by the back-end, while the GUI front-end is just a shell. The GUI provides a user-friendly interface to collect user input. That input is then used to form a command that is sent to the back-end for processing. Whatever output is generated by the back-end is then displayed to the user via the GUI.

As already noted, the back-end is implemented in Perl. Perl was used for a number of reasons. One of the language's major strengths is text processing, which is fundamentally what most authorship attribution tasks consist of. Second, there is a Perl library called "Text:: Ngrams" that does n-gram processing. Finally, previous authorship attribution researchers set a precedent by using Perl in their studies [2,15,18,20].

The source code for the back-end consists of nine total files: the main script called *Asap.pl*, six general-purpose supporting modules (Anonymize, Console, Directory, File, Report, and Validate), and two modules that implement the built-in authorship attribution methods (Burrows and Scap).

The **Anonymize** module strips what are referred to as C++ and C-style comments as well as string literal values from source files. C++ comments begin with a double slash (`//`), while C comments begin with `/*` and

end with `*/`. String literal values are enclosed in double-quotation marks. The **Console** module handles console messaging. The tool has several print settings: verbose, quiet, and debug. In quiet mode, only messages that are deemed "required" are printed. In verbose mode, all messages that would be relevant at all to the user are printed. In debug mode, all verbose messages are printed plus messages that would only be useful to a code developer. The module also provides a mechanism for printing and handling warning and error messages. The **Directory** module reads and deletes folders. The **File** module reads and writes files. The **Report** module provides a mechanism for tracking data from queries and experiments, and reporting on that data in the form of a spreadsheet. Finally, the **Validate** module performs much of the processing necessary for cross validation, such as segmenting the dataset into folds for k-fold cross validation experiments.

The *Asap.pl* file contains the main script. This is the script that is executed when issuing a command at the command-line. It reads the command-line arguments, and issues errors and help messages when the commands are ill-formed. If the commands and parameters are indeed well-formed, then this is the script that calls the appropriate methods in the supporting modules in order to actually perform the respective query or experiment based on the parameters specified.

Finally, the Burrows and Scap modules contain code to implement the Burrows and SCAP methods of authorship attribution, respectively. Each of these modules contain three primary functions: train, query, and experiment. The **train** function takes an input directory and an output directory as parameters in addition to any other parameters that are specific to that particular method of authorship attribution. The **query** function takes the query document as a parameter, as well as the training input directory and the query output directory. (Note that the input directory for the query will always be the output directory that was generated from the training. In other words, a query can't be executed until training has happened first. The output of the training phase will always be the input to the query phase.) Finally, the **experiment** function will take the test directory, training directory, input directory, and k-value as parameters. The type of experiment that will be executed will depend upon which of these parameters are defined. If the test directory and training directory are defined (but not the input directory or k-fold size), then a "default split" experiment will be executed. If the k-fold size and input directory are defined, then a k-fold cross validation will be performed. If the input directory is defined (but not k-fold size), then a leave-one-out cross validation will be performed. In addition to these three functions, both the Burrows and Scap modules contain other supporting functions that are specific to each particular method.

If any additional methods of authorship attribution are added to the ASAP tool in the future, then a module will be created specifically for that new method. It must contain the three functions, as described above (train, query, and experiment). The *Asap.pl* file must also be modified to accept parameters that are necessary to execute that particular method. None of the other code would be touched. In that way, the back-end software can be considered easily extensible.

3.2 Java Front-end

As noted before, most of the processing is done by the back-end, while the GUI front-end is just a shell. The GUI provides a user-friendly interface to collect user input. That input is then used to form a command that is sent to the back-end for processing. Whatever output is generated by the back-end is then displayed to the user via the GUI.

The GUI is implemented in Java. Java was chosen primarily for cross-platform compatibility. No installation by the user is required, because Java is pre-installed on most systems. The code is compiled into a single executable jar file, making it easy to distribute. It is also easy for the user to execute.

The GUI was visually designed using the NetBeans IDE, so the vast majority of the code was auto-generated by the IDE. The main file is $AsapGUI.java$, which literally contains the *main* method in addition to all of the auto-generated code. The Java Swing toolkit was used to create the GUI. The $AsapGUI$ class extends $JFrame$. The $JFrame$ contains 3 components of primary interest: two tabbed panes called *query Experiment Tabbed Pane* and $methodTabbedPane$ (whose data type is $JTabbedPane$), and a text area called *results Text Area* (whose data type is $JTextArea$). The *query Experiment Tabbed Pane* appears in the upper-left-hand area of the GUI and allows the user to select whether to perform a query or experiment, along with the associated parameters. The $methodTabbedPane$ appears in the upper-right-hand area of the GUI and allows the user to select which method of authorship attribution to use, along with the associated parameters. The $resultsTextArea$ is used to display the output of the query or experiment as it is being executed.

Other classes that are used are listener classes that act as event handlers. These classes listen for relevant events, such as text being entered into a text field, and respond accordingly. The majority of the Java code is

used to simply set up the GUI and event handling, so there is very little of interest to discuss. Once the Query or Experiment button is clicked, the *run Program Button Action Performed* event-handler method is executed. In this method, the values from the pertinent text fields and sliders are read, based on the tabs that are active, and a Perl command is constructed accordingly. After the Perl command is constructed, it is executed using the *exec* method from the *Runtime* class. (The *Runtime* class is a class found in the standard Java library in the *java.lang* package.) The *exec* method is used to issue a system command that runs in a separate process.

When the back-end command is executed, a new thread is launched that listens to the input stream associated with the process that is running that command. As input is read from that input stream, it is immediately displayed in the results text area. In this way, the output that is generated while the Perl command is executing is, in turn, displayed in real-time in the GUI's text area. The class that handles this is called *StreamGobbler*, which extends the *Thread* class. When the Perl command is issued using the *Runtime* class's *exec* method, a *StreamGobbler* object is created and executed as well. So, as a Perl command executes, there are 3 processes/threads that run in parallel: the GUI application, the Perl command, and a thread that reads the output from the Perl command and updates the GUI's text area accordingly.

To add an additional method of authorship attribution to the ASAP GUI, a corresponding *JPanel* component would need to be created that contains whatever GUI components are necessary to allow the user to specify all of the relevant parameters associated with that method (such as text fields, sliders, etc.). That panel would then be added as a tab to the *method Tabbed Pane*. The *run Program Button Action Performed* method would also need to be updated, so that when the Query/Experiment button is clicked, the necessary GUI components would be inspected in order to build a proper Perl command that would then be executed.

4 TOOL VALIDATION

To validate the ASAP tool, experiments were replicated whose results have been previously published [15–17], and the results generated by the tool were compared to the previously-reported results. The studies being replicated were chosen because the dataset used in the studies was consistent and easily attainable, the studies utilized the Burrows and SCAP methods of authorship attribution (which the ASAP tool supports), and the experimental methodologies used were clearly described making them ideal for replication.

The dataset consisted of 7,231 total files. It consisted of files written in C++ and Java, and it consisted of open-source programs and programs that accompany programming textbooks. Therefore, the dataset could be categorized into four segments: (1) open-source programs written in C++, (2) open-source programs written in Java, (3) textbook programs written in C++, and (4) textbook programs written in Java. The dataset consisted of programs written by a total of 30 authors (15 open-source authors and 15 textbook authors). The open-source programs were collected from the Planet Source Code website (planet-source-code.com) using a procedure established by Burrows [18]. The textbook programs were collected from the websites of textbook publishers and authors, as described by Tennyson [15].

There were a total of 12 experiments conducted. All of the experiments utilized a leave-one-out cross validation approach. The Burrows method was used once on each segment of data, which accounts for four of the experiments. The SCAP method was used twice on each segment of data, which accounts for eight of the experiments. The reason the SCAP method was used twice is because it was executed on both an anonymized version of the source files as well as the original, unmodified version of the source files. Many authorship attribution experiments are performed on "anonymized" versions of source files. These anonymized versions have all comments and string literals stripped from them. There are two reasons why this is sometimes done: (1) to hide the identities in cases where human subjects were used to collect the data and (2) to make the experiments more realistic by eliminating names and other explicitly-identifying information that might potentially be found inside comments and string literals. There was no need to execute the Burrows method twice, because it inherently anonymizes the source files. Neither comments nor string literals are used as tokens in the Burrows method, so they are inherently stripped. Using anonymized files with the Burrows method would always yield identical results to using non-anonymized files, so there was no need to execute it twice for each segment of data.

Note that although a total of 12 experiments were conducted, only 6 sets of values are reported. In the original studies being replicated, the results of each individual experiment were not reported. The results were combined by programming language. So, the reported values are as follows: (1) combined results from the open-source and textbook C++ programs using the Burrows method, (2) combined C++ results using the SCAP method, (3) combined anonymized C++ results

Table 1 Results reported from original studies that are being replicated

	Method	NumFiles	Correct	Percent
C++	Burrows	3655	3263	89.3%
	SCAP (Anonymized)	3655	3339	91.4%
	SCAP (Unmodified)	3655	3505	95.9%
Java	Burrows	3576	3163	88.5%
	SCAP (Anonymized)	3576	3242	90.7%
	SCAP (Unmodified)	3576	3367	94.2%
TOTAL	Burrows	7231	6426	88.9%
	SCAP (Anonymized)	7231	6581	91.0%
	SCAP (Unmodified)	7231	6872	95.0%

Table 2 Results of replicated studies

	Method	NumFiles	Correct	Percent	T-Test
C++	Burrows	3655	3264	89.3%	
	SCAP (Anonymized)	3655	3289	90.0%	
	SCAP (Unmodified)	3655	3481	95.2%	0.818
Java	Burrows	3576	3164	88.5%	
	SCAP (Anonymized)	3576	3222	90.1%	
	SCAP (Unmodified)	3576	3349	93.7%	0.886
TOTAL	Burrows	7231	6428	88.9%	
	SCAP (Anonymized)	7231	6511	90.0%	
	SCAP (Unmodified)	7231	6830	94.5%	0.848

using the SCAP method, (4) combined Java results using Burrows, (5) combined Java results using SCAP, and (6) combined anonymized Java results using SCAP.

The results from the original experiments [15–17] are shown in Tab. 1. The results from our replication of those experiments are shown in Tab. 2. With our results, we also show the p-value from a t-test. We are comparing the set of results from each programming language to determine if they are significantly different from the original results. As can be seen, the p-value is well above the typical 0.05 threshold for each group. Therefore, the results are not significantly different.

In addition to showing that the results are not significantly different, we further want to show that the results are indeed statistically similar. For this purpose, we use the cosine similarity metric. We represent the results of the original study as a 6-feature vector containing the six accuracy values reported above. The replicated results are also represented as a 6-feature vector. The resulting cosine similarity is 0.99999, where the maximum possible similarity value is 1, indicating that the replicated results are indeed statistically similar.

While we have shown that the replicated results are statistically similar to the original results, one might notice by manually comparing the SCAP results, that the replicated results have a slightly lower accuracy across all groups. Some variation is expected, especially in the SCAP method. The order that the authors' files are concatenated together will cause some variation. The profile length L will cause even greater variation. For example, let's say that the profile length is set to 100 and that the last n-gram in the profile occurred only once. Typically, there are numerous n-grams that are used only once by an author. However, since the profile length is set to 100, exactly 100 n-grams must be retained. The cut-off becomes arbitrary. Some of the n-grams that appear only once will be retained, while others will not. Also, ties can occur when multiple authors achieve the same maximum similarity score for a particular file, which causes ambiguity. The way in which these ties are handled in an experimental scenario is also ambiguous. We suspect that the consistency with which our results are lower is due to an implementation detail such as this. Perhaps, in the original study, in cases of a tie involving the correct author, it was considered to be a correct attribution. In our study, it was not. That would explain the slight, but consistent, lower results reported from our study.

The smallest segment of data used in the replicated experiments is the collection of anonymized open-source C++ programs, which consists of 521 files. Using either attribution method, it takes less than 10 seconds to perform training and execute a single query on this data. To perform a complete leave-one-out cross validation experiment, it takes less than 2 minutes. (Recall that a leave-one-out experiment requires training to be performed and a query to be executed as many times as there are files in the dataset.) The largest segment of data is the C++ textbook programs, which consists of 3,134 files. Using this data, a single training/query takes around 30 seconds, while a complete leave-one-out experiment takes approximately 15 minutes. Note that this benchmarking was done informally, and not performed in a well-controlled environment. A desktop computer was used with a 64-bit 3.3 GHz Intel i5 processor with 8 GB of RAM running Windows 7. We have no basis to compare these times to the original studies, because the times were not reported in those studies.

5 CONCLUSION AND FUTURE WORK

In this paper, we've presented ASAP (A Source Code Authorship Program), which is a tool that can perform tasks related to authorship attribution of source code. The tool is suitable for the expert or the layperson. An author can be attributed to an individual source file, or complete authorship attribution experiments can easily be performed using k-fold cross validation or leave-one-out cross validation techniques. A user-friendly GUI is provided for common tasks, or the back-end can be called directly at the command-line by the user or through batch scripts, as needed. The tool incorporates two state-of-the-art methods of source code authorship attribution: SCAP and Burrows. The software was designed

so that additional methods can be added as enhancements in the future. The tool was validated by recreating previously-published studies of authorship attribution, where the results generated by the tool were statistically similar to the results reported in the previous studies.

In the future, the tool could be expanded to include additional methods of source code authorship attribution. After additional methods have been added, a polling system could be incorporated. The user could select which methods to include in the poll, and when a query is made each of the selected methods could be used to generate the result. Methods of that attribute authors to other types of documents, such as object code or even natural language documents, could potentially be incorporated. The software could be enhanced to be even more extensible, making it easier to incorporate additional methods. The tool's code could be refactored, and potentially incorporate aspect-oriented programming.

While these and other improvements could certainly be made, we believe the ASAP tool as it currently stands could be a valuable asset to several different userbases. Researchers in areas related to authorship attribution, individuals in software forensics, those involved in cases of software copyright infringement, and those who teach programming classes could all use the tool for their own purposes. We believe the ASAP tool can fill a need in all of these potential user communities.

Acknowledgements I would like to extend a sincere word of thanks to the following current and former students for their software development contributions: Ethan Hill, Jacob Siegers, Justin Sassine, Conor Aberle, Joseph Sorgea, Anirudh Kambatla, Brian Rickard, and Michael Decker.

References

1. Zhao, Y., Zobel, J., Effective and scalable authorship attribution using function words, Proceedings of the Second Asian Information Retrieval Symposium (AIRS), pp. 174-189 (2005)
2. Frantzeskou, G., Stamatatos, E., Gritzalis, S., Katsikas, S., Effective identification of source code authors using byte-level information, Proceedings of the 28th International Conference on Software Engineering (ICSE), pp. 893-896 (2006)
3. Burrows, S., Tahaghoghi, S., Source code authorship attribution using n-grams, Proceedings of the 12th Australasian Document Computing Symposium, pp. 32-39 (2007)
4. Krsul, I., Spafford, E., Authorship analysis: Identifying the author of a program, Computers and Security (COMPSEC), 16(3), pp. 233-257 (1997)
5. MacDonell, S., Gray, A., MacLennan, G., Sallis, P., Software forensics for discriminating between program authors, Proceedings of the 6th International Conference on Neural Information Processing (ICONIP), pp. 66-71 (1999)
6. Ding, H., Samadzadeh, M., Extraction of java program fingerprints for software authorship identification, The Journal of Systems and Software, 72, pp. 49-57 (2004)
7. Lange, R., Mancoridis, S., Using code metric histograms and genetic algorithms to perform author identification for software forensics, Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO), pp. 2082-2089 (2007)
8. Kothari, J., Shevertalov, M., Stehle, E., Mancoridis, S., A probabilistic approach to source code authorship identification, Proceedings of the Fourth International Conference on Information Technology, pp. 243248 (2007)
9. Elenbogen, B., Seliya, N., Detecting outsourced student programming assignments, Journal of Computing Sciences in Colleges, 23(3), pp. 50-57 (2008)
10. Shevertalov, M., Kothari, J., Stehle, E., Mancoridis, S., On the use of discretized source code metrics for author identification, Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE), pp. 69-78 (2009)
11. Wisse W., Veenman C., Scripting DNA: Identifying the JavaScript programmer, Digital Investigation, 15, pp. 6171 (2015)
12. Neme, A., Pulido, J., Muoz, A., Hernndez, S., Dey, T., Stylistics analysis and authorship attribution algorithms based on self-organizing maps, Neurocomputing, 147, pp. 147159 (2015)
13. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., De-anonymizing programmers via code stylometry, Proceedings of the 24th USENIX Security Symposium, pp. 255-270 (2015)
14. Yang, X., Xu, G., Li, Q., Guo, Y., Zhang, M., Authorship attribution of source code by using back propagation neural network based on particle swarm optimization, PLoS ONE, 12(11), (2017)
15. Tennyson, M., Authorship Attribution of Source Code. Nova Southeastern University (2013)
16. Tennyson, M., Mitropoulos, F., Choosing a Profile Length in the SCAP Method of Source Code Authorship Attribution, 2014 Proceedings of the IEEE Southeastcon, pp. 1-6 (2014)
17. Tennyson, M., Mitropoulos, F., Improving the Burrows Method of Source Code Authorship Attribution, Proceedings of the IADIS International Conference on Applied Computing, pp. 39 (2013)
18. Burrows, S., Source Code Authorship Attribution. RMIT, Melbourne, Australia (2010)
19. Burrows, S., Uitdenbogerd, A., Turpin, A., Comparing techniques for authorship attribution of source code, Journal of Software Practice and Experience, 44, pp. 1-32 (2014)
20. Swain, S., Mishra, G., Sindhu, C., Recent Approaches on Authorship Attribution Techniques-An Overview, Proceedings of the International Conference on Electronics, Communication and Aerospace Technology (ICECA), (2017)
21. Hendrikse, S., The effect of code obfuscation on authorship attribution of binary computer files. Nova Southeastern University (2017)
22. Tennyson, M., A Replicated Comparative Study of Source Code Authorship Attribution, Proceedings of the 3rd International Workshop on Replication in Empirical Software Engineering Research (RESER), pp. 7683 (2013)
23. McDonald, A., Afroz, S., Caliskan, A., Stolerman, A., Greenstadt, R., Use Fewer Instances of the Letter "i": Toward Writing Style Anonymization, Proceedings of the International Symposium on Privacy Enhancing Technologies Symposium (PETS), pp. 299-318 (2012)

24. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al., Scikit-learn: Machine Learning in Python, Journal of Machine Learning Research, 12, pp. 2825-2830 (2011)
25. Frank, E., Hall, M., Witten, I. The WEKA Workbench, 4th ed. Morgan Kaufmann (2016)
26. Prechelt, L., Malpohl, G., Philippsen, M., Finding Plagiarisms among a Set of Programs with JPlag, Journal of Universal Computer Science, 8(11), pp. 1016-1038 (2002)
27. Schleimer, S., Wilkerson, D., Aiken, A., Winnowing: local algorithms for document fingerprinting, Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pp. 76-85 (2003)
28. Niezgoda, S., Way, T., SNITCH: A software tool for detecting cut and paste plagiarism, Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE), pp. 51-55 (2006)
29. Robertson, S., Walker, S., Okapi/Keenbow at TREC-8, Proceedings of the 8th Text Retrieval Conference (TREC-8), pp. 151162 (1999)