Date of acceptance Grade

Instructor

Software Plagiarism Detection Using N-grams

Kristian Wahlroos

M.Sc. Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, November 1, 2018

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Kristian Wahlroos

Työn nimi — Arbetets titel — Title

Software Plagiarism Detection Using N-grams

Oppiaine — Läroämne — Subject		
Computer Science		
Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	November 1, 2018	81

Tiivistelmä — Referat — Abstract

Plagiarism is an act of copying where one doesn't rightfully credit the original source. The motivations behind plagiarism can vary from completing academic courses to even gaining economical advantage. Plagiarism exists in various domains, where people want to take credit from something they have worked on. These areas can include e.g. literature, art or software, which all have a meaning for an authorship.

In this thesis we conduct a systematic literature review from the topic of source code plagiarism detection methods, then based on the literature propose a new approach to detect plagiarism which combines both similarity detection and authorship identification, introduce our tokenization method for the source code, and lastly evaluate the model by using real life data sets. The goal for our model is to point out possible plagiarism from a collection of documents, which in this thesis is specified as a collection of source code files written by various authors. Our data, which we will use to our statistical methods, consists of three datasets: (1) collection of documents belonging to University of Helsinki's first programming course, (2) collection of documents belonging to University of Helsinki's advanced programming course and (3) submissions for source code re-use competition. Statistical methods in this thesis are inspired by the theory of search engines, which are related to data mining when detecting similarity between documents and machine learning when classifying document with the most likely author in authorship identification.

Results show that our similarity detection model can be used successfully to retrieve documents for further plagiarism inspection, but false positives are quickly introduced even when using a high threshold that controls the minimum allowed level of similarity between documents. We were unable to use the results of authorship identification in our study, as the results with our machine learning model were not high enough to be used sensibly. This was possibly caused by the high similarity between documents, which is due to the restricted tasks and the course setting that teaches a specific programming style during the timespan of the course.

ACM Computing Classification System (CCS):
Information systems → Information retrieval → Retrieval tasks and goals → Near-duplicate and plagiarism detection
Information systems → Information retrieval → Retrieval tasks and goals → Clustering and classification
Information systems → Information systems applications → Data mining
Computing methodologies → Machine learning → Learning paradigms → Supervised learning
Computing methodologies → Machine learning → Learning paradigms → Unsupervised learning

Avainsanat — Nyckelord — Keywords
plagiarism detection; authorship identification; similarity detection

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Massive Open Online Courses (MOOCs) are a popular way to complete undergraduate courses offered by various institutes and universities. For example a course *Circuits and Electronics* led by Massachusetts Institute of Technology and Harvard University, gathered around 155 000 registered students from all over the world to a website called *edX*[1] in 2012 [7]. The structure of *Circuits and Electronics* consisted of two parts which are now common in majority of MOOCs: theory part and graded tasks which are offered weekly during the timespan of a course.

The course *Ohjelmoinnin MOOC* is an online programming course offered by University of Helsinki. It has a two-part structure; introduction and advanced course in Java programming language, where both are mandatory undergraduate-level courses including 14 weeks of total workload. During these weeks students follow the offered course material independently and submit their solutions to various programming tasks that are automatically tested and scored. If the participant is not a student in University of Helsinki, she can apply for a study right after completing the course and taking an exam [56], otherwise the student gains total of ten credits to her degree. As *Ohjelmoinnin MOOC* is based on scoring the submissions and students are free to choose their working hours without any major mandatory attendance, this can increase the likelihood for plagiarism. There are also over hundred students registered and many submissions sent by each student, making it very hard for course staff to manually detect possible cheating.

The word *cheating* here refers to an act of plagiarism and one of the ways to define the verb *plagiarize* is as "to steal and pass off (the ideas or words of another) as one's own"[2], and the person conducting this act is called *a plagiarist*. Source-code plagiarism on other hand, refers to the act of plagiarism that happens between software that is built from various source code documents. This kind of plagiarism can be also defined as *source-code reuse*, which includes the following four facets [11]: (1) copying others work without alterations, (2) copying and changing some parts of the code to fool a human inspector, (3) converting a solution from one language to another and (4) using code-generators to automatically create a solution.

Source-code plagiarism in academia is considered as a serious offence and there often exists a zero tolerance for it [11]. This is usually stressed at the start of courses and can lead to serious consequences ranging from rejecting the students current course registration to even suspension. Dick *et al.* points out that in some courses over 80% of the students were found guilty of cheating if they were given a good enough opportunity for it [15]. Usual forms of cheating methods were found to be related to plagiarism:

[1] `https://www.edx.org/` Accessed 10th April 2018

[2] `https://www.merriam-webster.com` Accessed 10th April 2018

copying solutions from the web, sharing solutions with friends and excessive collaboration between students.

The opinions about cheating motives varies between students and academics [11]. Academics reported that cheating is due to three major factors: external pressure, the ease of sharing solutions and cultural differences. Students on the other hand, gave two major reasons under the study: time pressure and heavy workloads. Given that MOOCs often have time sensitive weekly assignments, an automatic scoring system and the freedom to complete the course wherever students want can increase the likelihood of cheating among students who want to complete the course without effort.

In this thesis we approach the problem of source code plagiarism detection with data mining and machine learning. By data mining we mean an approach that is able to use computers to find interesting patterns from the data, and by machine learning a statistical process which is able to make predictions using previously observed data. For our proposed detection model, we first build two classifiers: identifying suspicious authors based on the similarity of documents and authorship identification that is able to predict the most likely author of a document. Using results from both of these classifiers, we propose a novel approach where the intersection of suspicious authors and the most likely authors of a document is able to reveal possible cases of plagiarism. Suspicious authors are grouped together to reveal clusters of possible plagiarists, whereas authorship identification is used to detect if a writing style of an author matches her previous work. The intersection of these two models should thus minimize the amount of falsely accused people, as the model has two inputs that are used to verify if the author is who she claims to be.

Following three research questions are asked and answered in this study, which are all tied closely to the question *How plagiarism can be automatically detected from source code documents?*

Q1: *What kind of approaches exist to detect source code plagiarism?*

Q2: *What are the possible benefits of using code structure for plagiarism detection?*

Q3: *How can one obtain a model with high plagiarism detection accuracy?*

To answer these questions, we first conduct a systematic literature review in which we establish a categorization for techniques used in plagiarism detection, show what kind of datasets are being used and search the theoretical foundations for our model. Then, we show how documents can be presented and retrieved in large-scale environment, and introduce the benefits of using the code structure within plagiarism detection. Finally, we evaluate the similarity detection and authorship identification individually and combine

the best scoring models to see how false positives are affected, and how they are introduced in the model.

Rest of this thesis is structured as follows. In Section 2 more detailed overview of source code plagiarism is given with a theory of classifiers, Section 3 presents the results of systematic literature review where we focus on data and methods applied in research, in Section 4 our method of using the result of two classifiers and the used real-life data sets are presented, Section 5 presents the results by comparing our method to two popular baselines. Section 6 discusses the results by answering the previous research questions, discusses the shortcomings with our proposal and presents possible problems when automatic system is used to accuse students from plagiarism.

2 Background

In this section we define the problem of plagiarism detection more formally, describe possible plagiarism strategies and give an overview to the similarity detection and authorship identification. We approach these latter two problems by first defining them, then showing how they tie closely to the domain of information retrieval, and finally give two real-life models. The first model is a probabilistic model able to predict the author based on the authors previous work, and the second model is a clustering algorithm able to group similar documents together. We start first by defining the problem of plagiarism detection.

Plagiarism detection. Given a set of documents $D = \{d_1, d_2, ..., d_n\}$ called as the corpus and a set of authors $A = \{a_1, a_2, ..., a_k\}$ who are writers of these documents, define a function f that is able to classify which documents are possibly written by more than one author, and who the possible suspects from the set of authors A are.

The above formalization gives an overview of the problem that is studied in this thesis. Some aspects about the general problem have been simplified for this study, as for example we don't try to reveal the *direction* of plagiarism, we use solely the data gathered from submissions and we only consider authors inside a predefined set. This means that we try only to detect if possible plagiarism can be observed from the the collection of documents submitted by students.

To get a better understanding of the details that are relevant to source-code plagiarism, instead of *e.g.* detecting plagiarism from essays, we define some important concepts and terms next. Starting from the definition of source code plagiarism, we show some common strategies of plagiarists and briefly introduce the underlying structure of source code and existing tools to detect plagiarism.

2.1 Source code plagiarism

Source code plagiarism refers to a plagiarism between source code files, which can happen in both academic programming courses as in software industry. Despite our focus on academic courses, both of these domains share a common problematic constraint which makes plagiarism detection often manually impossible. This constraint is simply the time constraint, creating a need for automatic detection tools as course administrators have limited hours to use for one course.

In academia, the underlying motives behind source code plagiarism include following concepts [30]: ambiguity about what is considered as excessive collaboration between students, using other students work to gain grades and minimizing the work needed to complete the course. There are at least three plagiarism behaviors with take-home exams [28]: help-seeking, collaboration and systematic cheating. Indicating that the most common type of plagiarism is accidental and done with other students from the same course.

Students committing plagiarism can have problems to define what they consider as source code plagiarism, and generally three common guidelines can be defined [44]:

1) Refactoring other students work, and submitting it as your own, is plagiarism.

2) If exercise templates are used, possible similarities between documents and templates are not plagiarism.

3) Submitting a direct copy of other students work is plagiarism.

Detecting 2. and 3. are straightforward; code templates can be filtered out from documents so that they contain only students own work and detecting direct copy can be easily found by using string matching techniques. However, the problem arises when students try to hide the plagiarism by mutating the directly copied document.

Plagiarism strategies

Some common source code transformation techniques, often called as *obfuscation strategies*, are targeted mainly towards two types of changes [30]: lexical and structural. Lexical changes such as changing variable names, do not require a deeper understanding of the logic. Structural changes requires understanding the program logic, and includes modifications which change the layout of the source code but keeps the logic same. For example when considering following clause with an operand `if(a == true)`, it can be written equally as `if(a == !false)`, keeping the logic same but mutating the lexical information.

Table 1 shows some of the most common transformation targets when speaking of source code. Given source code from another student, plagiarist can apply transformations on these targets and complicate the task for a human to spot plagiarism, or even confuse naïve methods like string matching techniques. The motivation behind these transformations is that plagiarists want to hide traces and thus, the detection method must be resilient against obfuscation strategies.

Table 1: Common targets for transformations [30]. Lexical changes are superficial and easy to change, whereas structural changes requires understanding of the logic.

Lexical	**Structural**
Comments	Loops
Formatting	Clauses
Naming	Statement order
	Operand order

Transformations targets defined in Table 1 closely relate to another study which characterizes six levels of transformations [18].

Table 2: Six levels of transformation which can be applied to any source code document.

Level of change	**Target**	**Example action**
1	Comments and indentation	Add extra spaces and newlines
2	Identifiers	Rename all variables
3	Declarations	Reorder functions
4	Modules	Merge functions
5	Statements	Use `for` instead of `while`
6	Logic	Change whole expressions

Applying all of these transformations one after another from Table 2, makes the detection of plagiarism very difficult, as the plagiarized document diverges too much from the original document and hides most of the traces that could be used for detection. However, as the textual information changes, plagiarists still try to maintain the same logic between original and copied documents. This means that there still exists some kind of similarity, but this similarity can not be found directly from the textual representation of a source code. Information about the logical structure is thus crucial and accessible, when source code is parsed to a tree format.

Code structure

Source code is structured text, made of keywords and user-defined variables. To write a running program, one must know the rules *i.e.*, the *grammar* of a

language, which is represented as the order in which various keywords and variables must follow each other. A compiler is the core of a programming language and it is used to transform source code meant for humans into machine code which is meant for computers. When grammar rules must be interpreted by the compiler, it uses a *parse tree* that is generated from the source code [29]. This parse tree captures the syntax and semantics of the source. The abstracted version of parse tree is called the *abstract syntax tree*.

Consider for example storing an integer value to a variable. The source code in JavaScript and its abstract syntax tree is visible from the following diagram.

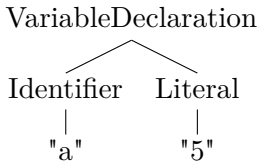


Diagram 1: Example syntax tree for the expression `var a = 5;`. Two leaves are generated, one for the identifier and its value, and another for the literal and its value. Together they form a declaration of a variable.

Pruning the leaves of Diagram 1 leads to a more general expression that captures the logic of the source code, becoming resistant against most of the transformations given in Table 2. This gives the ability to detect similar *structure* rather than similar *text tokens*, where latter is more vulnerable to simple transformations. For example changing the name of the identifier or the value of the literal will not affect the upper tree structure at all.

Tools

Because plagiarism is considered a serious offence, a lot of various detection software has been made to detect it. Novak lists seven of the most well-known tools in a review [41]: *MOSS, JPlag, SPLaT, SIM, Marble, Plaggie* and *Sherlock*. These tools can be classified into five different categories based on methods used: text, token, graph, tree and hybrid which combines one or more approach. The most common way to detect plagiarism follows a five step approach: pre-process documents, tokenize documents, exclude templates, calculate similarities and find suspects using the similarity scores. It's notable that all of these tools listed by Novak are trying to calculate the similarity value between documents.

As an example, JPlag is tool targeted for Java language [45]. It works by utilizing the program structure, transforming the program code into sequence of tokens by traversing the parse tree of a program and using predefined token correspondence to form a token stream which represents the source code. To form similarity score between two programs, a string matching

algorithm called *Greedy String Tiling* [59], is applied and the summed length of all matches are being stored. This results a similarity value between zero and one, where the value one means that two programs are exact copies of themselves. Makers of JPlag suggest that their tool is resilient against most common obfuscations made by plagiarists *e.g.* renaming and reordering of statements.

2.2 Similarity detection

Similarity detection, or code clone detection, focuses directly on finding similar functionality from a set of source code documents. We define it formally as following.

Similarity detection. Given a set of source code documents $D = \{d_1, ..., d_n\}$ called as the corpus, define normalized similarity function $sim : d_i, d_j \rightarrow [0, 1]$ where $1 \leq i, j \leq n$, $sim(d_i, d_j) = sim(d_j, d_i)$, and $sim(d_i, d_i) = 1$. In other words similarity score is equal regardless of the order of two documents and similarity value between clones is one. With an optional threshold $\theta \in [0, 1]$ one can define the limit when two source codes are considered as too similar.

The definition of the similarity function is ultimately based on how the source code document is presented as a data [46]: document consisting of plain text, series of tokens, syntax tree, series of metrics or as a graph. If the document is seen purely as a sequences of characters, one can use naïve methods like string matching techniques to detect fragments of copy and paste, which requires no pre-processing. Other data formats require some kind of transformation or an extraction process.

If one represents the source code simply as metrics, then there exists two major categorization for those metrics [46]: attribute-counting metrics and structure metrics. Attribute-counting refers to high-level features which can be extracted directly from plain source code *e.g.* line counts and the amount of whitespace, whereas structure metrics use the underlying structure of the source code to capture the low-level representation [55].

The core process to detect similarity can be visualized to Figure 1, which follows the general structure seen in state of the art systems [46]. In Figure 1, after the corpus has been defined, pre-process stage takes as an input the unmodified source codes to perform two key tasks: to remove unnecessary segments such as template code and to determine the level of comparison granularity. The granularity one chooses can range from function-level to document-level, depending how accurately the results should pinpoint plagiarism. After the source code is partitioned, it is transformed into intermediate representation which consists of two parts: extraction and normalization. In extraction the data is modified so it is usable in similarity function, and covers things like parsing, tokenization or building control flow from the given code. In normalization one applies techniques which reduces

the variation between documents [46]: comments and whitespace removal, uniforming user-defined identifiers or removing anything not relevant for the detection process. The final result after calculating the pairwise similarities is a collection of documents which are suspect of being too similar to each others.

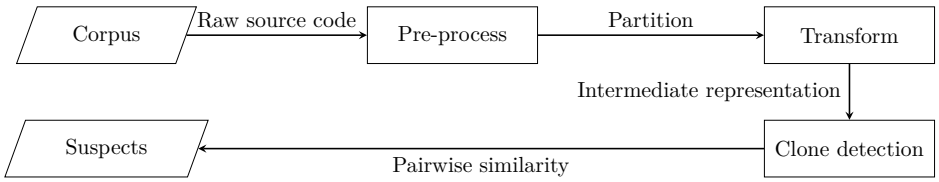


Figure 1: Similarity detection process for source code documents. Documents are transformed into intermediate representation so similarity scores can be calculated.

The key issue regarding to similarity detection are false-positives that can be handled by manual verification after the detected documents are gathered [46,55]. This is often a mandatory step as the detection tools simply try to find similarity between the documents, but this similarity can be pure coincidence for example when the solution space for a given task is highly limited.

2.3 Authorship identification

Authorship identification deals with the issue of trying to name the author of a document given some previous work of the author. This problem can be seen as a classification task [33], and thus, we define authorship identification formally as following.

Authorship identification. Given a set of documents D, a set of authors A and a function $f : D \to A$ that identifies the writer by assigning every source code document $d \in D$ to one author $a \in A$. Estimate f with $\hat{f}$, a classifier that treats every document as a feature vector $\mathbf{x}$ where $x_i \in \mathbb{R}$, and every known class as a vector $\mathbf{y}$ where $y_i \in \{0, 1\}$. The binary value represents boolean value if the ith author is the predicted author for a given document, which means that if dimension of the vector $\mathbf{y}$ is $\mathbb{R}^n$, then there are n authors $|A| = n$. The predicted author $\hat{y}$ can be thus expressed with $\hat{f}(\mathbf{x}) = \hat{\mathbf{y}}, \hat{y} \in \hat{\mathbf{y}}$.

This classifier should be able to discriminate between writing styles of different authors, which is heavily restricted by the grammar of the chosen programming language. Writing style can commonly refer to everything that is controllable by the author *e.g.* how one names variables or uses spacing.

One way to represent writing style in programming is by using software metrics [33]. Software metrics can be put into roughly three categories:

layout being fragile metrics which are easily transformed by the IDE, style which are non-fragile metrics related to layout and lastly structure which can capture experience and ability of the programmer. Because source code can be also thought as a text written with a specific language, natural language processing (NLP) techniques can be applied. Targets of these NLP techniques can be categorized into five-level stylistic features called *stylometrics features* [52]. Categorization for stylometric features is visualized in Table 3, where semantic features are the most difficult to form as they require understanding deeper meaning of the written source code.

Table 3: Five-levels of stylometrics features. More external information is required on each level.

Category	**Feature examples**
Character	Character subsequences, types, compression
Lexical	Token statistics, word sequences
Syntactic	Errors, expression usage, keywords, parse tree
Application-specific	Indentation, language-specific constructs
Semantic	Synonyms, functional dependencies

In natural language authorship analysis, statistical methods are often being used [52]. More specifically machine learning methods are used to find reoccurring patterns that are able to distinguish between writing styles. The training of these statistical models can happen in two ways: via profile-based or via instance-based approach. In profile-based approach, all documents that are presented as observed data per author are concatenated into one file. In instance-based learning, each text is used as an individual data point. If we know that each document belongs to one author, then the problem can be thought as an instance of *a multiclass classification* [52], where there exist many possible classes for a single observed data point, but only one of them can be correct.

2.4 Information retrieval

Before we can apply any statistical models to our problem, we need a way of properly represent the collection of documents. One way is to think the problem of plagiarism detection as retrieving certain kinds of documents from a collection. Therefore we next introduce some topics from the theory of *information retrieval*.

Information retrieval (IR) is a collection of strategies of finding documents from large collections [36]. These documents are often represented as unstructured text and possible methods covers topic like: clustering documents to find similar documents, classifying documents based on their content, ranking text for query search and building search engines. In this thesis as we mainly focus on finding similarities between documents and how to

classify the author, we disregard some of the query-based focus of IR and use techniques that are relevant to plagiarism study *i.e.*, how document can be represented for statistical models and how distance between two documents can be calculated.

2.4.1 Document representation

As we require some form of numerical way to represent one document, we use following IR-related concepts to express the documents in our corpus: *vector space model* which captures algebraically the representation of the document and a *weighting scheme* which gives more importance to specific terms appearing in documents.

Vector space model One form of vector space model is called *a binary term-document incidence matrix* [36], which represents documents as columns and terms as the rows of a matrix. Terms are gathered by tokenization procedure which divides single document into units that are often *words* of the document, but can also be adjacent characters.

Let $\mathbf{M}_{n\times k}$ be a matrix having n terms and k documents, then the value of $\mathbf{M}_{d,t}$ *i.e.*, term t appearing in document d, is 1 if it appears at all and zero otherwise. Table 4 shows example matrix build from programs in Appendix A.

Table 4: Example of a binary term-document incidence matrix for three example programs in Appendix A.

Term \ **Document**	A	B	C
`public`	1	1	1
`sum`	0	1	0
`double`	1	1	0
⋮	⋮	⋮	⋮
`b`	1	1	1

Taking a transpose of the values in Table 4 gives a document-term matrix, where one row represents a document having n dimensions, reserving one dimension for each term occurrence. For example the representation of document A is the vector $\mathbf{a} = [1, 0, 1, \cdots, 1]$. This is referred as the *bag of words* model, because it treats every word as independent event, losing the information about ordering of the words [36].

A binary term-document incidence matrix is however very naïve, giving the same value for a term regardless of the times a term appears in a document. This can be improved by using a method called *term weighting*, which is able to assign non-binary values for terms.

Term weighting schemes One simple scheme is called *term frequency*, which is the number of times term t appears in document d denoted by $tf_{t,d}$ [36]. Let $f_{t,d}$ denote the raw frequency count, then term frequency can be given as $tf_{t,d} = f_{t,d}$ and normalized by dividing the raw frequency with the total frequency over every term in document

$$tf_{t,d} = \frac{f_{t,d}}{\sum\limits_{t' \in d} f_{t',d}} \tag{2.1}$$

To scale down the most frequently appearing terms, one can use *inverse document frequency* which boosts the weights of rare occurring terms [36]. Inverse document frequency is defined as

$$idf_t = \log \frac{N}{df_t} \tag{2.2}$$

Where N is the total amount of documents and df_t is the count of documents that contains the term t.

By taking the product of the term frequency and inverse document frequency, we get a weight for each term appearing in a document called term frequency–inverse document frequency (TF-IDF):

$$tf\text{-}idf_{t,d} = tf_{t,d} \cdot idf_t \tag{2.3}$$

By using TF-IDF weighting scheme, we are able to discriminate between documents despite their length and diminish the problem of frequently appearing terms. In other words, TF-IDF is able to grow or decrease the importance of individual terms making it an important concept, as in programming a number of terms appear frequently because the language is very structured and defined by a finite amount of keywords.

2.4.2 Document similarity

As we have a way of expressing a document as a numerical vector using a weighted vector space model, we are interested in being able to calculate a similarity value between two documents, which is a crucial part for similarity detection. One way of doing this is by using *cosine similarity*.

Cosine similarity Cosine similarity measures the similarity between two documents by calculating the cosine of the angle between the document vector representations [36]. Let $\mathbf{x}, \mathbf{y}$ be these vectors for documents d_1, d_2, then

$$sim(d_1, d_2) = \cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\|_2 \|\mathbf{y}\|_2} = \frac{\sum\limits_{i=1}^{n} x_i y_i}{\sqrt{\sum\limits_{i=1}^{n} x_i^2} \sqrt{\sum\limits_{i=1}^{n} y_i^2}} \tag{2.4}$$

The dot-product is normalized in Equation 2.4 with Euclidean norm and because weights derived from TF-IDF are non-negative, the cosine similarity gets values between zero and one [36]. Values closer to one indicate high content similarity, whereas values closer to zero indicate dissimilarity. The opposite of cosine similarity is *cosine distance*, which can be calculated by subtracting cosine similarity from one *i.e.*, $d = 1 - \cos(\theta)$.

2.4.3 Retrieval metrics

Retrieving candidate documents to detect possible plagiarism creates a need to measure how well the retrieval process is performing. For evaluating the retrieval method, three important metrics have been defined [36]: precision, recall and F_1-score. To describe these metrics, we use a confusion matrix which has four fields: true positive indicating a correctly retrieved relevant item (TP), true negative indicating correctly rejected irrelevant item (TN), false negative indicating relevant item which was incorrectly rejected (FN) and false positive indicating an irrelevant item which was incorrectly retrieved (FP). These four fields are visualized in Table 5.

Table 5: Confusion matrix which can be used to visualize the error in retrieval [36].

	Relevant	**Irrelevant**
Retrieved	TP	FP
Rejected	FN	TN

Precision, recall and F_1-score can all be defined by using the individual cells of the confusion matrix. Both precision and recall count the rate of true positives to falsely retrieved documents, and balancing between these values requires knowledge about the domain where they are being applied to. If precision is high, the model is able to correctly retrieve greater portion of correct positive cases. If recall is high, the model is able to retrieve high portion of relevant documents. Formally, precision and recall are defined as

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{|\text{relevant} \cap \text{retrieved}|}{|\text{retrieved}|} \tag{2.5}$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{|\text{relevant} \cap \text{retrieved}|}{|\text{relevant}|} \tag{2.6}$$

F_1-score combines precision and recall, giving an average value between them. It's defined as

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.7}$$

All of the above metrics help to evaluate how well the retrieval process is performing, but they require some form of process where one decides if a document is relevant or not. This relevancy with respect to data point, is often decided by the class or label that is attached to it.

2.5 Document classification

Following [36] we formulate the document classification problem as $\gamma : \mathbb{X} \to \mathbb{C}$, approximating the function γ that maps data $\mathbf{x} \in \mathbb{X}$ to a class $c \in \mathbb{C}$. For example *a binary classifier* would choose between a positive and a negative class $\mathbb{C} = \{+, -\}$, whereas *a multiclass classifier* chooses between multiple classes $\mathbb{C} = \{c_1, c_2, \cdots, c_n\}$ for a given document.

To be able to classify documents algorithmically into some predefined classes, the classifier must learn a way to predict outputs from inputs [26]. That is, given some d-dimensional data $\mathbf{x} \in \mathbb{R}^d$, the classifier γ must predict the response variable y which represents the class. To make the prediction in binary case, the classifier is supported with observed data as a training data, represented in matrix format $\mathbf{X}_{n \times d}$ and predefined response variables in a column vector $\mathbf{y}_{n \times 1}$ [26]. This so called *training* of the model refers to the classifier using some a part of the data to tune its internal parameters. When a data point outside the training set is given, the classifier is able to give prediction for it based on the data it has seen already. This kind of setting is also called as *supervised learning* as there exist some data that guides the process [26].

If we know the response variable y of each observed data point represented as vector $\mathbf{x}$, we want to have similar data to be predicted with the same value. The prediction the algorithm gives can be noted as $\hat{y}$ for $\mathbf{x}$, and because this value is just a prediction, evaluation is needed for the algorithm to be able to change its learning into the right direction. This evaluation happens by penalizing wrong predictions with a loss function $L : \hat{\theta} \times \theta \to \mathbb{R}$ [26], where $\hat{\theta}$ is the prediction that the classifier gives and θ the value wanted the prediction to be. For example a loss function able to penalize categorical predictions, is called *0-1 loss* and formulated as $L(\hat{y}, y) = I(\hat{y} \neq y)$, where I is the indicator function [26].

Naïve Bayes Naïve Bayes is a probabilistic model, which is often used as a baseline model in text classification [37]. It applies the *Bayes' theorem* to estimate the parameters of the classifier *i.e.*, conditional probabilities with respect to data. Bayes' theorem is generally given for events A, B as

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)P(B \mid A)}{P(B)} \tag{2.8}$$

Where $P(B)$ can be expressed by the law of total probability

$$P(B) = P(B \mid A)P(A) + P(B \mid \neg A)P(\neg A) \tag{2.9}$$

When the area of interest is classification, we denote the probabilities of events A, B as the *prior* and *likelihood*. Prior being in this case the probability of a class appearing in data $P(y = c),\ c \in \mathbb{C}$ and likelihood the likeliness of a document belonging to a class $P(\mathbf{x} \mid y = c)$. Therefore, Equation 2.8 can be rewritten as [37, 63]

$$P(y \mid \mathbf{x}) = \frac{P(y)P(\mathbf{x} \mid y)}{P(\mathbf{x})} = \frac{P(y)P(\mathbf{x} \mid y)}{\sum_{c \in \mathbb{C}} P(y = c)P(\mathbf{x} \mid y = c)} \tag{2.10}$$

In Equation 2.10, the denominator remains constant because $\mathbf{x}$ is kept unchanged as it's the sum over every known class, and therefore the Equation 2.10 is proportional to the product between prior and likelihood [36].

$$P(y \mid \mathbf{x}) \propto P(y)P(\mathbf{x} \mid y) \tag{2.11}$$

Because the underlying real distribution is unknown, prior and likelihood have to be estimated from the training data. Estimated prior can be calculated from the relative frequency *i.e.*, number of samples belonging to the class c divided by the total number of observations [36].

$$\hat{P}(c) = \frac{\#c}{|\mathbb{X}|} \tag{2.12}$$

To calculate the estimated likelihood $\hat{P}(\mathbf{x} \mid y)$, one uses the assumption that features represented in feature vector $\mathbf{x}$ are conditionally independent with respect to each other. This assumption simplifies the likelihood by applying the chain rule [36].

$$\hat{P}(\mathbf{x} \mid y) = \hat{P}(x_1, x_2, \cdots, x_n \mid y) = \prod_i^n \hat{P}(x_i \mid y) \tag{2.13}$$

To assign a single data point into a class, the most probable class is chosen [36, 37, 63]. This is referred to also as the *maximum a posteriori* (MAP), and the final class assignment *i.e.*, the result of the classifier γ, is expressible as

$$\hat{y} = c_{\text{map}} = \underset{c \in \mathbb{C}}{\operatorname{argmax}}\, \hat{P}(c) \prod_i^n \hat{P}(x_i \mid c) \tag{2.14}$$

This means that the most likely class for a data point is the class which maximizes the posterior, which again is proportional to calculating product between prior and posterior.

A variant of Naive Bayes called *Multinomial Naïve Bayes*, is able to form the likelihood by assuming underlying multinomial distribution [37]. Given the problem of document classification and the feature vector $\mathbf{x}$ represented as term frequencies of vocabulary $\mathbb{V}$, the conditional probability of Equation 2.14 can be given in similar way as term-frequency function in Equation 2.1.

We use *Laplace smoothing* to eliminate the problem with terms appearing zero times [36], which can happen regularly because the vocabulary $\mathbb{V}$ is only formed from the training data. This smoothed version of the conditional probability using frequencies is given as [36]

$$\hat{P}(x \mid c) = \hat{P}(t \mid c) = \frac{f_{t,c} + 1}{\sum_{t' \in \mathbb{V}} (f_{t',c} + 1)} \tag{2.15}$$

In Equation 2.15 $f_{t,c}$ is the frequency of term $t \in \mathbb{V}$ appearing in class $c \in \mathbb{C}$, so the conditional probability of a data point given a class is estimable from the smoothed relative frequency of the term t that the point x represents.

It has been shown that TF-IDF weighting scheme improves the classification results even as TF-IDF weights are non-discrete like raw term frequencies are [31]. This result means that all documents can be efficiently represented also as vectors of TF-IDF weights for the Multinomial Naive Bayes.

2.6 Document clustering

Document clustering is a process that is used to group a set of documents into n clusters, so that their similarities within each cluster is maximized *i.e.*, documents belonging to the same group are as similar as possible. This is relatively easy task for a human to do manually for a small set of documents, but in order to perform this task automatically in large scale typically uses *unsupervised learning.*

Unsupervised learning can divide the observed data *i.e.*, documents, into subgroups called *clusters* [26]. The main difference to supervised learning (classification) is that when the data is represented as a sequence $X = (\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n)$, where $\mathbf{x}_i \in \mathbb{R}^d$ is d-dimensional feature vector which represents the ith document, we do not have the sequence of response variables $\mathbf{y} = (y_1, y_2, \cdots, y_n)$ to guide the process. Thus there is no clear loss function that is dependent from the true classes of the data, which leads to the situation where distribution of the data determines the classes [36]. The performance of the unsupervised model can be therefore very subjective, requiring some kind of prior domain knowledge [26].

As an example, the Figure 2 visualizes two-dimensional data generated from three separate distributions. Because we know how the data was generated in Figure 2, we are able by prior knowledge and by visually to divide the space exactly into three regions. However, if the data would be more uniformly distributed and one could not say the exact amount of regions, then this task requires more knowledge about how two data points are able to have similar location. When considering for example plagiarism between documents, we are highly interested of cases where there is too much similarity between two or more documents.

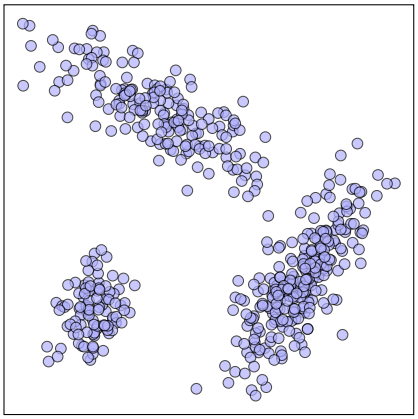


Figure 2: Data points centered around three distributions with respect to the means and variances. Three separable clusters are visible.

The normalized similarity value $s \in [0, 1]$, or respectively distance value $d = 1 - s$, is defined before the clustering algorithm is executed [26], and it ultimately controls what kind of cluster are being formed. Distances between data points can be precomputed into matrix of documents $\mathbf{M}_{d \times d}$, where $\mathbf{M}_{i,j}$ is the similarity, or distance value between two documents d_i and d_j.

We next give a brief formalization for the problem of clustering and then introduce two different unsupervised clustering algorithms: *K-means clustering* and *DBSCAN*.

Document clustering. Given a set of datapoints $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$, $\mathbf{x}_i \in \mathbb{R}^d$ which represents the documents, define assignment $\gamma : X \to \{1, \cdots, k\}$ where k is the total amount of clusters [36]. The set of clusters can be notated by $\Omega = \{\omega_1, \omega_2, \cdots, \omega_k\}$ and each document belongs to one cluster $\forall d \in \omega$.

K-means clustering K-means clustering requires a predefined number of clusters, the parameter k, to be predefined and it assumes there exists a *centroid i.e.*, a mean point, for every cluster. These *cluster centroids* are notated as $C = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \cdots, \boldsymbol{\mu}_k\}$, $\boldsymbol{\mu}_i \in \mathbb{R}^d$ [36]. To assign a data point to a cluster, one calculates the *squared Euclidean distance* from a point to a centroid $\|\mathbf{x}_i - \boldsymbol{\mu}\|^2$, and minimizes this distance. In other words this means, that the data point is assigned to the same cluster as its nearest centroid. The K-means algorithm works iteratively by updating the cluster assignments for each point and calculating new centroids until the algorithm converges. Convergence can be decided in multiple ways and one of those ways is, that no new assignments have been done when all data points are being iterated.

Algorithm 1 shows the pseudocode for K-means. In it, centroids are first chosen randomly from the set of data points with InitCentroids-function. Then iteratively, until there are no further updates to centroids C, k clusters

are first initialized as empty sets. Next cluster assignments are calculated from the set of data points with respect to Euclidean distance to the nearest cluster. After every loop, new centroids are calculated by taking the mean of assigned data points per cluster in UpdateCentroids. The return value of the K-means will be k centroids, representing the middle points of a cluster, and cluster assignments Ω indicating which data point belongs to which cluster.

Algorithm 1 K-means algorithm [36]

Require: Set of datapoints X
Require: Amount of clusters k
  procedure K-means(X, k)
    $C \leftarrow$ InitCentroids(X, k)
    while stop criterion has not been met **do**
      for $i = 1$ to k **do**
        $\omega_i \leftarrow \{\}$
      end for
      for $j = 1$ to $|X|$ **do**
        $l \leftarrow \text{argmin}_l \|\mathbf{x}_j - \boldsymbol{\mu}_l\|^2$
        $\omega_l \leftarrow \omega_l \cup \mathbf{x}_j$
      end for
      $C \leftarrow$ UpdateCentroids(Ω)
    end while
    return C, Ω
  end procedure

The drawback with K-means clustering is that one must specify the parameter k before the clustering [26]. When for example detecting similar documents, there is no indication beforehand that how many documents should be grouped together, and therefore pre-estimating number of clusters can be very difficult. To overcome this issue, one can utilize the density of the data points rather than the direct distance between them.

Visualization of the clustering result using K-means for the same data as in Figure 2 is seen in Figure 3. From it, we see how three different clusters emerge as the value of parameter k is 3. There are however many ways a human could interpret the results, and especially the assignment of data points which are located between multiple clusters.

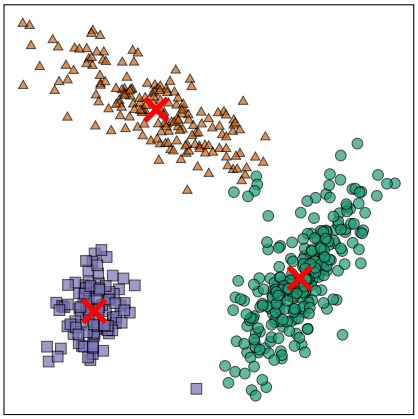


Figure 3: Result of K-means clustering after converge. Crosses indicate the cluster centroids, other colored markers cluster assignments. Parameter k is set to 3, meaning that three different clusters will be discovered by the algorithm.

DBSCAN Density-based spatial clustering of applications with noise (DBSCAN) can produce clustering by using only the density information, label some data points as noise, produce arbitrary sized clusters and use any distance function [17]. It requires two parameters. Epsilon ε which controls the neighbour search radius, and $MinPts$ defines the minimum number of points needed to form a cluster.

To form a cluster, point q must be reachable from point p *i.e.*, there must exist a path from p to q that fulfills both ε and $MinPts$ parameters. To form this path, some points are labeled as core points satisfying parameters simultaneously, and some as border points which have at least one core point in its ε-range. If a data points is neither above, it is labeled as noise.

The pseudocode for DBSCAN is given in Algorithm 2, where DiscoverNeighbours is a recursive function that finds the neighbourhood space by forming the radius based on the distance function, and retrieves all reachable points restricted by the ε-range. The algorithm is able to form the amount of clusters itself and requires no pre-defined amount of clusters. Using parameters $\varepsilon = 0.5, MinPts = 15$ and setting distance function as Euclidean distance, DBSCAN learns more denser clusters than K-means and is able to label some data as noise. This noise is visible in Figure 4 as black cross markers.

Algorithm 2 DBSCAN algorithm [17, 48]

Require: Set of datapoints X
Require: Distance radius ε
Require: Minimum neighbour count $MinPts$
Require: Distance function $dist : X \times X \to \mathbb{R}$
  procedure DBSCAN$(X, \varepsilon, MinPts, dist)$
    $k \leftarrow 0$
    for $i = 1$ to $|X|$ **do**
      $N \leftarrow$ DiscoverNeighbours$(X, dist, \mathbf{x}_i, \varepsilon, MinPts)$
      if $\mathbf{x}_i$ is a core point **then**
        $k \leftarrow k + 1$
        $\omega_k \leftarrow N \cup \mathbf{x}_i$
      else
        $\mathbf{x}_i$ is noise
      end if
    end for
    return $\{\omega_1, \omega_2, \cdots, \omega_k\}$
  end procedure

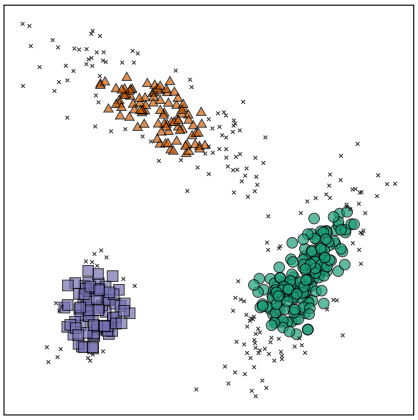


Figure 4: Result of DBSCAN by setting parameters $\varepsilon = 0.5, MinPts = 15$. Black crosses refers to noise as these points are too far away from core points which forms the three clusters.

If data points in Figure 4 would represent documents in Euclidean space, we could interpret noise as documents which are too dissimilar to any other document and requires no further attention. Therefore using DBSCAN allows us to react only to the most densely packed documents.

2.7 Summary

In this section we showed what we mean by the term *source code plagiarism* and what kind of transformations a plagiarist can apply to her source code. We approach the problem of plagiarism detection from two major viewpoints: identifying the most likely authorship of a source code document to verify the author and clustering highly similar documents together to reveal suspects. Because retrieving documents is close to the theory of information retrieval, we explained some important concepts regarding it. Finally, we introduced the equivalent algorithms for authorship identification and similarity detection.

3 Literature Survey

Systematic literature review was conducted to construct an overview of the current state of source code plagiarism research. In the review, we mainly focus on methods to determine what is considered as a well-performing approach to plagiarism detection, especially focusing on authorship identification and similarity detection. Our review consists six steps that follows the structure of systematic review [43]: 1) purpose, 2) details of search, 3) inclusion criteria, 4) exclusion criteria, 5) information extraction and 6) analysis.

The database that was utilized to query research articles is called *Scopus*[3], which is a service containing peer-reviewed scientific literature. It allows users to search scientific articles by matching *e.g.* titles, abstracts or keywords to user-defined query. The service itself maintains links to articles which are published under for example *ACM (Association for Computing Machinery)* and *IEEE (Institute of Electrical and Electronics Engineers)*, both of these being major computer science releases.

Following subsections describe how the review was conducted and what results were found. We first form a categorization between studies to gain overview of the methods that are applied, then we extract statistics about data sets used in studies, and finally show the various methods that are applied to detect plagiarism.

3.1 Survey methodology

Querying Scopus is similar to querying databases in SQL-like languages. The query used is as follows:

```
TITLE-ABS-KEY (("plagiarism" OR "authorship identification")
               AND "source code")
AND  (LIMIT-TO (SUBJAREA,"COMP"))
```

[3] `https://www.scopus.com/` Accessed 2nd February 2018

Above query translates to searching for articles which title, abstract or keywords contains the word *plagiarism* or *authorship identification* and the term *source code*. These keywords were chosen in order to find articles which study the problem of plagiarism finding from source code either in general terms, or by utilizing authorship identification techniques. Finally, the query limits the area of study to computer science publications to find relevant methods for this thesis.

The total number of articles gathered by querying Scopus in the inclusive search part of the literature review was 187, and the date when the query was done was 7th of February 2018. The distribution of studies per year can be seen in the following plot.

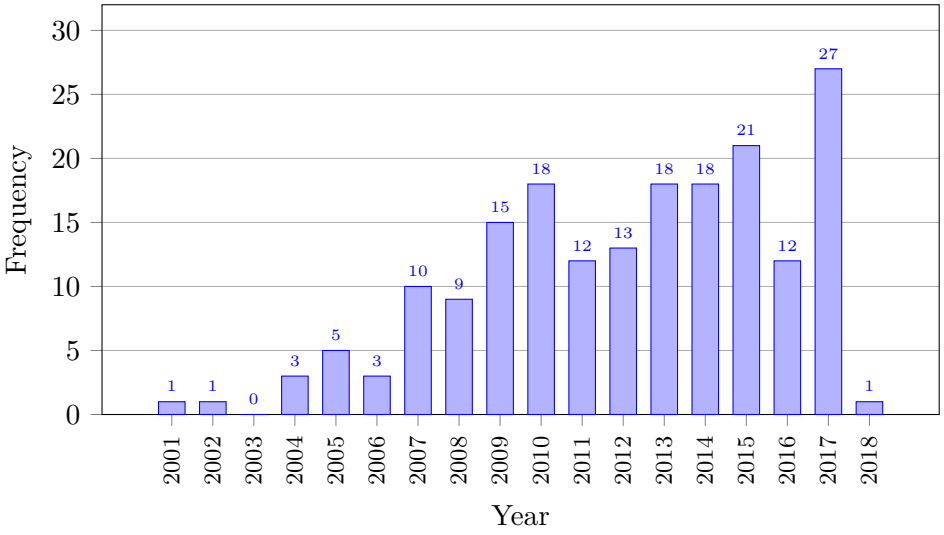


Figure 5: Distribution of articles per year after the inclusion phase.

After the inclusive search, we performed exclusion phase. This was done by filtering out manually all articles that were any of the following types: a review of certain aspect of source code plagiarism *e.g.* student motives behind plagiarism, an improvement to some pre-existing algorithm *e.g.* algorithmic speedups, plugins to online learning management systems, applications where the used method was not explained, studies that used either byte-level information or information gathered during running the program, hashing techniques *e.g.* compression and using the remaining size as a metric, system reviews which did not address the methods and theses. Beside these attributes, articles needed to also test their proposed method in some way and the amount of documents in experiment phase needed to be larger than two. The reason for adding this as a limiting factor was to gather studies that used test sets to evaluate the performance of their model.

The final number of articles after exclusion, and which are inspected more carefully in this systematic literature review is 32. From the set of 32 studies,

we look answers for following questions: *how plagiarism can be detected from source code*, *what are possible features that can be derived from source code* and *how can one identify the author of a given source code*. We start first by categorizing the articles by their themes to see what kind of different approaches there are to deal with the problem of plagiarism detection. After the initial classification, we summarize statistics about the data and explain the used methods, using terminology introduced in Section 2.

3.2 Categorization

We first categorize the articles based on the objective; plagiarism detection and authorship identification. However, as authorship identifying can also work as a way to detect plagiarism by verifying the author, we use the following high-level split visible in Table 6.

Table 6: Two high-level categories that both are closely connected to the domain of plagiarism detection.

Similarity detection	**Authorship identification**
[1, 8, 12, 13]	[3, 9, 53, 54]
[19, 23, 27, 61]	[4, 10, 16, 32]
[14, 22, 40, 42]	[5, 20, 21, 34]
[38, 39, 51, 64]	[6, 60, 62]
[24]	
Number of articles	**Number of articles**
17	15

Even though the articles are quite evenly divided in Table 6, these high-level groups do not provide an overview of various methods being used. Thus we continue to categorize both groups into smaller categorization, which reveals differences between them.

Similarity detection in itself can be further divided into at least two general categories based on the current tools [41]: attribute and structure. Then naturally as authorship identification uses features derived directly from the source code too, we can use the same categorization to authorship identification studies. However, there are more finer categorizations that define the studies better based on the features they use, and thus we propose the following five categories: *attribute counting (AC)*, *segment matching (SM)*, *n-gram (NG-STRU)*, *tree-based (TREE-STRU)* and lastly *hybrid approaches (HYB-STRU)*. These categories are similar to categories identified from other plagiarism detection studies by Ali *et al.* [2] and can be summarized briefly as follows.

Attribute counting (AC) Utilizing countable statistics, often referred as *metrics*, that are gathered from source codes. This includes any features derived directly from source code like amount of words per line, number of lines per source code and number of keywords. For example the first two programs in Appendix A represented as metrics is seen below.

Program	Line count	Empty spaces	Semicolon count
Listing 1	10	1	5
Listing 2	8	1	3

Table 7: First two programs in Appendix A; A and B, represented as software metrics.

Segment matching (SM) Considers two source codes as two strings and counts maximum match between them i.e. the longest common subsequence. These problems are also known as string matching problems, where one of the most famous algorithms is *Greedy String Tiling* introduced first by Wise [59]. As an example, statements `int a = 2;` and `int b = 2;` matching sequences are "`int`" and "`= 2;`". We also count all string similarity measures like string edit distances to this category.

***N*-gram (NG-STRU)** Treating the source code as a continuous sequence of characters or words, and splitting it using a sliding window where the window size is the value of n and the window traverses on either character or word level. This forms the vocabulary of the source code which is then transformed into occurrences of particular terms that are present. For example the statement `int a = 2` could be transformed into following word level 2-tuples using $n = 2$ (bigram). The first value of the following tuples is the n-gram extracted and the second value is the frequency: (`int a`, 1), (`a =`, 1), (`= 2`, 1).

Tree-based (AST-STRU) Constructing a tree presentation from the source code. For example Diagram 1 captures the structure of a simple assignment in parse tree, where one could compare *e.g.* nodes of trees, to define a similarity. The generation of a tree presentation usually requires a parser, whereas the inspection of a generated tree can be done with tree traversal methods using recursive functions.

Hybrid (HYB-STRU) Combines the usage of tree structure with n-gram representation. For example it can be a method which traverses abstract syntax tree, prints it and generates n-gram representation from the output.

We form the subcategorization for previously mentioned high-level classes;

similarity detection and authorship identification. Results for similarity detection studies can be seen from the following table, where most of them use structural features, indicated by the STRU-ending. Many articles also tends to use n-grams to represent the source code.

Table 8: Subgroups and sizes of similarity detection studies.

AC	**SM**	**NG-STRU**	**AST-STRU**	**HYB-STRU**
[38]	[8, 64]	[1, 12, 19] [13, 27, 42] [14]	[23, 40, 51] [22]	[24, 39, 61]
#AC	**#SM**	**#STRU**		
1	2	14		

When making the division for authorship studies, we can see that Table 9 is more evenly distributed in contrast to similarity detection articles in Table 8. Studies seem to utilize more countable attributes from source codes, but using n-grams are as popular as they are in similarity detection, which is quite obvious when one considers that these methods are able to capture the writing style of an author from high-level features. For example authors can name the identifiers how they like, introduce comments and use various stylistic techniques when they write source code.

Table 9: Subgroups and sizes of authorship identification studies

AC	**NG-STRU**	**AST-STRU**	**HYB-STRU**
[16, 32, 34] [4, 5, 6]	[9, 10, 21] [20, 53, 54]	[3]	[60, 62]
#AC	**#STRU**		
6	9		

This suggests that utilizing structure is popular in both high-level classes, but quite dominant in similarity detection. Also both groups show high popularity to the usage of n-grams, and authorship identification prefers to represent documents as statistical metrics.

3.3 Descriptive statistics

The data used for evaluating the each approach of the 32 gathered articles is presented next, where the focus is on the similarity detection of the following attributes: number of total documents, is there any synthetic data used and the average number of lines of code (Avg. LOC). For the authorship identification, we focus on features: documents per author, number of possible authors and whether the data is authentic or synthetic. We summarize these findings utilizing the same categorization that was made earlier.

Similarity Detection

Attribute counting study by Moussiades and Vakali [38], uses two real data sets written in C++. Both contain programming assignments and a synthetic set of programs. The first data set contains 24 programs having an average of 247 lines of code per submission and the second set 51 programs having an average of 178 lines per source code. The forged data set uses two modified versions from one program, trying to deliberately confuse state-of-the-art detectors.

Segment matching study by Brixtel et al. used three corpora on their evaluation, which are written in Haskell, Python and C [8]. Haskell corpus has 13 documents averaging 400 lines per each, Python 15 documents averaging 150 lines per each and C 19 documents averaging 250 lines per source code. Study by Zhang and Liu used 12 programs written in C that all reflected different plagiarism strategies [64].

Studies utilizing n-grams are summarized into following table.

Table 10: Data used in similarity detection studies utilizing n-grams.

Feature \ Article	[1]	[12]	[13]	[14]	[19]	[27]	[42]
Documents	191	179	5408	1277	5302	1356	2935
Synthetic	No	No	No	No	No	No	No
Avg. LOC	NA	NA	63.7	NA	NA	NA	NA

It's visible from the Table 10 that there are a lot more documents used in experimentation and surprisingly synthetic data is not used at all. This is due to the usage of student submissions and competition data sets like *Google Code Jam* submissions, which was utilized for example by Flores *et al.* [19].

Table 11: Data used in similarity detection studies utilizing abstract syntax tree

Feature \ Article	[22]	[23]	[40]	[51]
Documents	22 214	NA	121	555
Synthetic	Yes	NA	NA	No
Avg. LOC	20	NA	NA	305.7

One can see from the Table 11 that the study [22] has a large number of documents. This is due to two facts: they reported the size as pairs of documents and they used a generator to form a lot of forged documents from a small set of 10 original submissions. Ganguly and Jones don't explicitly report the statistics of their data set [23], but use a competition test set

called *SOurce COde Re-use* (SOCO). This competition offers a set of C and Java files which contains known cases of cross-lingual plagiarism [47].

Finally, a hybrid study by Xiong *et al.* utilizes 40 assignments gathered from students [61], Muddu *et al.* uses 5054 original files that they mutate to introduce copied code [39] and Ganguly *et al.* uses both train and test set of the SOCO competition, totaling around 12 000 files [24].

Authorship identification

Usage of data in studies dealing with the problem of identifying the author and utilizing attribute counting, are summarized to the following table where we focus on the amount of candidate authors and documents per author reported in studies.

In Table 12, one can see that there are two same data sets used in [5, 6]. This set was collected from *SourceForge*[4] projects and there are around 61 to 377 files per author. Rest of the attribute counting studies prefers to use *e.g.* submissions gathered from students.

Table 12: Data used in authorship studies utilizing attribute counting.

Feature \ Article	[4]	[5]	[6]	[16]	[32]	[34]
Authors	120	10	10	46	8	20
Documents per author	NA	61-377	61-377	NA	3	3
Synthetic	No	No	No	No	No	No

Data sets from the second popular method n-grams used in authorship identification, are summarized into following table.

Table 13: Data used in authorship studies utilizing n-grams

Feature \ Article	[9]	[10]	[20]	[21]	[54]	[53]
Authors	100	100	8	8	30	30
Documents per author	14	14-26	2	2	NA	NA
Synthetic	No	No	No	No	No	No

There exists three different data sets used by three different authors in Table 13: Burrows *et al.* [9,10] used data set gathered from students C programming assignments, Frantzeskou *et al.* [20, 21] used open-source programs written in Java and Tennyson *et al.* [53, 54] used programs written in C++ and Java.

The only study that mainly used abstract syntax tree in their authorship study is by Alsulami *et al.* [3]. They used *Google Code Jam* to gather 700

[4] `https://sourceforge.net/` Accessed 5th February 2018

Python source code files belonging to 70 programmers averaging around 10 submissions per author.

Finally, the data used in two hybrid studies are summarized. Wisse and Veenman used repositories from version control website called *GitHub*[5] [60]. The largest author pool they had while testing was 30. Zhang *et al.* had the data set also gathered from websites like *GitHub* in their study [62]. Their largest data set with respect to the author size, was imbalanced set of 503 programs belonging to 53 authors.

Summary
When looking the data usage of plagiarism study as a whole, one can see that almost all studies use data that is non-synthetic *i.e.*, use real-life data, that can be gathered for example from students course submissions or from competitions like the SOCO dataset. In similarity detection studies the median of the amount of source codes used is 447 and very few studies reported the average lines of code. In authorship identification the median of possible authors in studies is 30 and the documents per author ranges from 2 to as high as 377.

3.4 Methods

In this section we focus on the methods used in these studies. The formal notation used in studies is generalized to match the style presented in Section 2, meaning that a single element of a set and scalars are represented as lower-cased italics, matrices are bold and upper-cased, vectors are bold but lower-cased, tree structures as capitalized T and segments of source codes as capitalized S, which often implies string format. We use words *term, token* and *word* as a synonym for a single sequence of characters, usually divided by spaces.

3.4.1 Similarity detection

The problem of similarity detection is described formally in Section 2.2 and we use that as a general high-level baseline. In similarity detection studies we focus mainly to the actual similarity measure, as it is one of the major focus in following 17 studies.

The attribute counting study by Moussiades and Vakali uses a graph clustering on top of pair-wise similarities calculated using the Jaccard coefficient [38]. Authors use following form of Jaccard coeficcient in their study where A is the indexed set of substitute keywords per source code

$$sim(d_1, d_2) = \frac{|A(d_1) \cap A(d_2)|}{|A(d_1) \cup A(d_2)|} \tag{3.1}$$

[5] `https://github.com/` Accessed 5th February 2018

The indexed set can be built considering language dependent keywords *e.g.* `while, for, false and true` in C++, and marking their position with respect to the occurrences of same keywords previously. However, authors suggest that in order to generalize the set more, substitution keywords should be used. This means that for example all occurrences of `for`- and `while` -loops should be counted together, which helps to protect against plagiarism. The graph clustering algorithm Moussiades and Vakali uses is called *WMajorClust* which works by presenting all pairs of source codes as non-directed graph $G = (V, E)$ where the set of vertices V represents the source codes while the set of edges E are weighted by equation 3.1. We can also express the definition of E by Moussiades and Vakali with following constraints

$$E = \Big\{ \{d_i, d_j, sim(d_i, d_j)\} \mid (d_i, d_j) \in D \times D \wedge sim(d_i, d_j) \geq \theta \Big\} \tag{3.2}$$

In equation 3.2, θ is a user-defined parameter and works as a minimum threshold value that separates non-plagiarized source codes from plagiarized ones *i.e.*, two source codes will not share an edge if their similarity is below θ.

Segment matching study by Brixtel *et al.* presents their algorithm, using three major steps [8]: pre-filtering, segmentation and document distance calculation. Their pre-filtering is to normalize the source code so that every keyword and parameter definition is transformed into a single symbol. In segmentation, the authors split the source code by lines forming set of segments S_k each presenting the partitioned set of a single source code. Similarity calculation happens by first forming distance matrix $\mathbf{M}$ between two source codes d_1, d_2 and then comparing all pairs of segments $(s_i^1, s_j^2) \in S_1 \times S_2$ where $S_k = (s_1^k, ..., s_n^k)$, with *Levenshtein edit distance* [35]. Distance matrix $\mathbf{M}$ is then transformed into noise reduction matrix $\mathbf{H}$ by finding the maximal matching between segmentations. Finally, $\mathbf{H}$ is filtered into a matrix $\mathbf{P}$ by convolution and utilizing a threshold. With the matrix $\mathbf{P}$, distance between two pairs of documents is calculated by Brixtel *et al.* as

$$sim(d_1, d_2) = 1 - \frac{1}{\min(|S_1|, |S_2|)} \sum_{i,j} 1 - \mathbf{P}_{(i,j)} \tag{3.3}$$

Zhang and Liu utilize AST and their core method is mainly constructed from two methods [64]: forming the AST-representation and similarity calculation. Their AST-representation is done by traversing the tree structure and turning it into textual format by printing the nodes, and similarity calculation is computed using *Smith Waterman algorithm* that finds the optimal matching between two strings S_1, S_2 [49]. Zhang et Liu gives the form for similarity calculation between two source codes as

$$sim(d_1, d_2) = \frac{2 \cdot \text{ SLength}(d_1, d_2)}{|S_1| + |S_2|} \tag{3.4}$$

Where SLength is the length of maximal matching string obtained via *Smith Waterman algorithm*, and $|S_k|$ represents the character length of one segment.

N-gram studies take a different approach. Cosma and Joy uses *Latent Semantic Analysis* to find suspicious documents [12]. They first preprocess the documents by removing *e.g.* short terms and comments. Then all documents are transformed into a term-by-file matrix $\mathbf{A}$, which presents each source code as occurrences of each possible unique term. Values of $\mathbf{A}$ are weighted, and then $\mathbf{A}$ is decomposed via *singular value decomposition* [25] into reduced matrices $\mathbf{A} \approx \mathbf{A}_k = \mathbf{U}_k \mathbf{\Sigma}_k \mathbf{V}_k^\intercal$ where $\mathbf{U}$ represents terms by dimension, $\mathbf{\Sigma}$ singular values and $\mathbf{V}$ files by dimensions. The dimensionality reduction is performed for all these matrices by considering only the first 30 columns represented by the subscript k. Finally, the similarity between a query vector $\mathbf{q}$ representing term frequency of document d_i, and document d_j represented as a column $\mathbf{a}_j$ of matrix $\mathbf{A}$ is calculated by using *cosine similarity* [12]

$$sim(\mathbf{q}, d_j) = \frac{\mathbf{a}_j^\intercal \mathbf{q}}{\|\mathbf{a}_j\|_2 \|\mathbf{q}\|_2} = \frac{\mathbf{a}_j \cdot \mathbf{q}}{\sqrt{\sum_i \mathbf{a}_{(j,i)}^2} \sqrt{\sum_i \mathbf{q}_i^2}} \tag{3.5}$$

Acampora and Cosma [1] continues on same style as Cosma and Joy [12], first preprocessing the documents by lowercasing and removing comments, syntactical tokens and short terms. Then using singular value decomposition with weighting, to form three matrices from the corpus of source codes. For the reduced matrix $\mathbf{V}$, they perform a *Fuzzy C-Means* clustering algorithm, which is tuned with *ANFIS* learning algorithm to optimize the hyperparameters of Fuzzy C-means [1]. The process returns a membership degree $\mu_{i,k}$ per document, indicating how close ith document is to kth cluster. Flores *et al.* [19] uses similar preprocessing approach to Cosma and Joy. They first process the documents by lowercasing them and removing repeated character, tabs and spaces. Then transform the documents into 3-grams and weighting them by using a term frequency. Finally, similarity is calculated using cosine similarity where t is one of the 3-grams and tf is the term frequency function [19]. Formally this can be calculated in a same way as in equation 3.5 between two documents as

$$sim(d_i, d_j) = \frac{\sum_{t \in d_i \cap d_j} tf(t, d_i) tf(t, d_j)}{\sqrt{\sum_{t \in d_i} tf(t, d_i)^2 \sum_{t \in d_j} tf(t, d_j)^2}} \tag{3.6}$$

Heblikar *et al.* [27] preprocesses documents by lowercasing, pruning repeated whitespace and removing single symbols. They then normalize the documents

by considering most frequent terms, renaming similar terms under same symbols and ultimately filtering them completely out from the source codes. For detection phase, they use same approach as Flores *et al.* [19] but use both 1-grams and 2-grams with TF-IDF weighting. Also Ramírez de la Cruz *et al.* [13] and Ramírez de la Cruz *et al.* [14] decides to use cosine similarity and Jaccard coefficient. The only major difference being, that Ramírez de la Cruz *et al.* uses additional structural and stylistic features, forming total combination of eight various similarity measurements [13]. Where as Ramírez de la Cruz *et al.* [14] uses cosine similarity with character 3-grams to calculate five different similarities: lexical, stylistic, comments, textual and structural. Lastly, Ohmann and Rahal proposes density-based clustering to form clusters of similar documents [42]. Their similarity approach follows closely to other studies presented above: filtering and normalization as preprocessing, data format as word n-grams and similarity values gained by using cosine similarity. The only major difference contrast to other studies is, that they perform tokenization that transforms source code into predefined set of tokens *e.g.* integer declarations are changed to a string "DN".

Tree-based studies mostly relies on calculating similarity between two tree structures T_i, T_j obtained from the original documents d_i, d_j by parsing them. For example Ng *et al.* first generate a parse tree T from the source code, then decompose the parse tree into subtrees $T' \subseteq T$ with respect to the functionality *e.g.* imports are categorized together [40]. The similarity score is calculated by traversing trees with *depth-first search* and summing the node and token similarities for all subtrees. The function of similarity between two documents, is defined below where $simST$ is a subtree similarity.

$$sim(d_i, d_j) = sim(T_i, T_j) = \frac{\sum\limits_{i,j} simST(T'_i, T'_j)}{10 \cdot |T'|} \cdot 100 \tag{3.7}$$

Son *et al.* computes similarity value between two parse trees with a modified parse tree kernel, and state that their kernel function is able to consider also the length of the document [51]. They define the kernel function k via recursive function C where n is a node of a subtree T'. Function C finds a maximal similarity between two nodes thus authors calls it also as the *maximum node value*

$$k(T_i, T_j) = \sum_{n_i \in T'_i} \sum_{n_j \in T'_j} C(n_i, n_j) \tag{3.8}$$

This kernel function captures the similarity between two tree structures and the normalized similarity function is defined as [51]

$$sim(d_i, d_j) = \frac{k(T_i, T_j)}{\sqrt{k(T_i, T_i) \cdot k(T_j, T_j)}} \tag{3.9}$$

Another study that utilizes kernel between tree structures is by Fu *et al.* [22]. They first build abstract syntax tree from a source code by normalizing and weighting nodes with TF-IDF, then use a tree kernel to measure similarity between two tree structures. This tree kernel is defined as

$$k(T_i, T_j) = \sum_{t_i \in T_i} \sum_{t_j \in T_j} \Big(\lambda \cdot \ \text{dist}(\text{word}_{t_i}, \text{word}_{t_j}) \cdot w_{t_i,T_i} \cdot w_{t_j,T_j} \Big) \tag{3.10}$$

Where λ is a decay factor penalizing tree height, dist is the edit distance between two string values, word_t is the string value of in-order traversed subtree $t \in T'$ and $w_{t,T}$ is a weight given to a single subtree t inside abstract syntax tree T. Fu *et al.* normalize the source code by transforming every variable name, array size definition and indexing of an array into single unified symbol. Then, authors remove all leaf nodes with common symbols to reduce noise, for example round and curly brackets. The similarity score between two source codes is calculated by normalizing the kernel values, leading ultimately to equation 3.9, which is the equivalent to cosine similarity [22]. The last tree-based study is by Ganguly and Jones [23]. They use information retrieval approach and treat every document as a *pseudo-query*. This means that every document is first parsed into abstract syntax tree, then nodes belonging to a similar functionality are collected together. Finally, specific fields are gathered from this collection by using ranking scores. For example all class definitions are treated as one collection and from that collection, names of the classes are extracted as weighted terms to construct the pseudo-query. Ganguly and Jones suggest that their approach allows to differentiate usage of same string literals in different situations.

Study utilizing an ensemble of methods by Xiong *et al.* presents their system named *BUAA AntiPlagiarism* which uses abstract syntax tree to generate n-grams [61]. They first run the code through optimizer that gets rid of unnecessary complexity, then turn the simplified code into AST-representation and prune the tree by for example removing variable names and constants. This pruned tree is travelled in pre-order traversal that turns the tree into string format and forms n-grams from that representation. To calculate similarity between documents, Xiong *et al.* uses Jaccard coefficient, which was defined earlier in Equation 3.1. Muddu *et al.* continues on combining approaches and presents their system called Code Plagiarism Detection Platform (CPDP) [39]. CPDP detects plagiarism by first tokenizing the AST, then turning the generated token stream into 4-grams to be used in querying the matching documents. Finding the most closest document is done by using string matching algorithm *Karp-Rabin Greedy String Tiling* given n-grams from the set of matching documents. Finally, the last similarity detection study is from Ganguly *et al.* [24]. They also use information retrieval approach in similar fashion as they did in previous work [23], to tackle with the problem related to n-grams without AST-representation;

false-positives and exhaustive pair-wise calculation. Their method consists of building a pseudo-query and a ranked list of most matching documents. This pseudo-query is built by first retrieving three kinds of features from documents: (1) lexical, (2) structural like identifiers, function types, and data types and (3) stylistic features like average term length.

3.4.2 Authorship identification

The problem of authorship identification is different from similarity detection. Instead of trying to find a function to represent a numerical value as similarity between two source code to detect plagiarism, authorship identification aims to determine the writer of a document. It's common in following studies that the identification happens in closed environment *e.g.* student submissions. Closed environment thus implies that the true author is someone from the predefined set of possible authors and that authorship identification can be used as authentication. Upcoming 15 studies reflect these findings.

Ding's and Samadzadeh's study follows the typical method of attribute counting studies. Authors extract total of 56 metrics belonging to three classes [16]: layout, style and structure. Their feature selection is done by using variance and correlation analysis, whereas classification is done with *canonical discriminant analysis*. Lange and Mancoridis extract 18 mostly text-based metrics and use genetic algorithm to find out the best combination [34]. Their classification is done by constructing a histogram per feature for every developer and then calculating which of the histograms are most closest to the unknown source code. Kothari *et al.* uses very similar histogram-based technique but considers style metrics and character distributions, namely character level 4-grams [32]. To select the most matching features for a single author, Kothari *et al.* uses information entropy which uses the distributions to make probabilistic evaluations. To classify the author, their approach is to have a database of writer profiles, extract metrics from source code and calculate the likelihood which known writer is the author. Arabyarmohamady *et al.* uses writing style to identify an author [4]. They build a profile for every author by transforming the source code into a feature vector *i.e.*, fingerprint and compare it to database of profiles to choose the most closest author profile. Plagiarism clusters are created by comparing the similarity of each feature vector with Euclidean distance, thus allowing the detect issues with authorships and reveal plagiarism cases. Bandara and Wijayarathna has nine metrics that they use to generate tokens and token frequencies [6]. For example, one of their metrics is number of characters per line (LLC) and to tokenize it, one creates token for specific length n (LLC_n) and calculates the frequencies. This distribution of tokens is input to learning process called *sparse auto-encoder* that learns to encode the features with neural network. Weights of this neural network are used as features to *logistic regression* which classifies the author to document. Finally, similar study by again

Bandara and Wijayarathna, uses full neural networks for the same task [5]. They use the same nine metrics with tokenization to get distributions per metric, and use them as a input to their deep neural network to learn to predict author from features.

Authorship identification with n-grams mostly use a profile-based method called *The Source Code Author Profile (SCAP)* [20, 21, 53, 54]. The idea of SCAP is following: all known source codes from author a are concatenated into one text file, n-grams are generated and only L most frequent are kept per author to generate a profile P. To predict the author $\hat{y}$ of a source code d, one calculates how many n-grams does a unknown profile P_d has in common with pre-existing author profile P_a, or respectively

$$\hat{y} = \underset{a \in A}{\operatorname{argmax}} |P_a \cap P_d| \tag{3.11}$$

The first study that uses its own method is by Burrows and Tahaghoghi. They approach the problem with information retrieval and consider author and document as queries [9]. Normalizing the documents is done by keeping only operators and keywords, while n-grams are used to present one document as overlapping sequences. Ranking the documents to create a ranked list, happens with a proposed measure called *Author1*. Author1-measure evaluates the similarity between documents and a query, and is defined using term frequencies for both query q and document d

$$\text{Author1}(q, d) = \sum_{t \in q \cup d} \frac{1}{\min(|tf(t, q) - tf(t, d)|, 0.5)} \tag{3.12}$$

Burrow *et al.* continues on the topic of information retrieval in another study, where they experiment on six additional features on top of 6-grams [10]: white space, operators, literals, keywords, input/output (I/O) and function names. Rest of the n-gram related studies use the SCAP method, mostly using it as a baseline while trying to improve it. For example Frantzeskou *et al.* analyze the contribution of four different high-level features when using SCAP [21]. These features are comments, layout features like spacing, identifier names and keywords. In another study, Frantzeskou *et al.* continues to use SCAP, but study the significance of user-defined identifiers with four categories [20]: identifiers using basic data types like `int` for integers, class identifiers, method identifiers and all identifiers defined by the author. Tennyson and Mitropoulos study first the best profile length L for SCAP [54] and in another study, use two Bayesian methods to build an ensemble [53]. This ensemble works by utilizing the SCAP and the Burrows method as a baseline to decide the author of a document. If there exists disagreement between baseline models, probability theory is used to classify the author. These two Bayesian methods are *Maximum a Posteriori* and *Bayes Optimal Classifier*, and both of them calculate probability that author a wrote the document d given data about authors previous work.

Alsulami *et al.* utilize deep neural network which uses features derived from the abstract syntax tree of a source code [3]. Their method relies on learning the features from AST, rather than explicitly handcraft them. To learn these features, Alsulami *et al.* encode the tree as a vector by first traversing its nodes and subtrees with *depth-first search*, then map them as a multidimensional vector called *embedding layer* which works as an input for their model. Lastly, the author classification is done using the deep neural network.

Finally, two hybrid studies are by Wisse and Veenman [60] and Zhang *et al.* [62]. Wisse and Veenman approach the problem of authorship by using features extracted directly from the AST. They first parse the source code into AST, then traverse it to derive metrics belonging to three classes: structural, style and layout. Structural features include most frequent n-grams, style features statistics about comments and layout features are various spacing related metrics. The classification is done by deriving a high dimensional feature vector from the data and using *Support Vector Machine* for classification. Zhang *et al.* on the other hand, extract multiple features belonging to four classes: layout, style, structure and logic. In their study, layout features capture the usage of whitespace characters, style captures usage of variable names and lengths, structure statistics about methods and logic is defined as word-level n-grams.

3.4.3 Findings

Even though there exists multiple different ways to obtain similarity score between a pair of source code, there are some reoccurring strategies for comparison. Jaccard similarity coefficient can be used for a similarity measure between two sets of tokens [13, 14, 38, 61], string edit distance is a simple but requires exhaustive pair-wise search to find direct occurrences [8,39,64], cosine similarity can be used as similarity measure with vector space models often utilizing weighting schemes like term-frequency or TF-IDF [12, 13, 14, 19, 27] and similarity between tree structures can be calculated with a tree kernel as a dot product or by exhaustively comparing the nodes [22, 40, 51]. To reduce noise for similarity calculation, two kinds of approaches are used: preprocessing and normalization. In preprocessing the data is turned into another format *e.g.* from plain text to AST or filtered to remove unnecessary information, and in normalization some weighting scheme is often applied or keyword generalization.

Author identification mainly uses metrics and n-grams to answer the question *who wrote this code?*. Used metrics often belongs to three categories: layout, style and structure. However, coming up with meaningful metrics can be hard and can easily lead to excessive feature engineering [16], where one comes up with *ad hoc* solutions for the features to be used. N-grams are popular and used in SCAP, with information retrieval and alongside statistical

metrics. The actual classification is in many cases done by representing source code as a vector of numerical values *i.e.*, the metrics, and then using a supervised machine learning algorithm [60].

4 Research Design

Previously introduced concepts, results of systematic literature review and the focus on document similarity and authorship identification has given an overview of the problem of source code plagiarism detection. Next we propose and later evaluate a two-phase model that combines both similarity detection and authorship identification, and hypothesize that such minimizes the amount of false positives in plagiarism detection. False-positives are problematic as it means an innocent author is considered to be a possible plagiarist, and therefore having too sensitive model introduces extra work. It is notable that none of the studies that were discovered during the literature review in Section 3 combine these two approaches when making the decision of possible plagiarism. Therefore our goal is to introduce a new approach which uses document clustering to retrieve similar documents, and tasks submitted by students to form an author profile for each student.

Both of our models are based on other studies presented in the literature review and combine the high-level approach used in many tools [41]: preprocess, normalize, evaluate and predict. For the similarity detection we use lower level features that capture the structure and are resilient against transformations introduced in Section 3.4, and for the authorship identification we use higher level features which can capture the style of an author. The generalization of the proposed model is given below.

Algorithm 3 Detecting plagiarism between a set of source code files.

Require: Set of authors A
Require: Set of source code files D written by various authors $\forall a \in A$
Require: Index of the exercise of interest $i \in \mathbb{N}$
Require: Length of word level n-grams $n_w \in \mathbb{N}$
Require: Length of character level n-grams $n_c \in \mathbb{N}$
Require: Minimum rate of similarity $\varepsilon \in [0, 1]$
  procedure PLGDETECT($A, D, i, n_w, n_c, \varepsilon$)
    $D' \leftarrow$ NORMALIZE(D)
    $A_{susp} \leftarrow$ DETECTSIM(A, D'_i, n_w, ε)
    $A_{auth} \leftarrow$ TRAINANDPREDICTAUTHOR(A, D', i, n_c)
    return $A_{auth} \cap A_{susp}$
  end procedure

Algorithm 3 requires six parameters so it can fully function and the most import ones are the collection of documents D, and the set of authors A

i.e., all source codes are submitted by a known author. Remaining four parameters $(i, n_w, n_c, \varepsilon)$ can be defined freely. In this work we estimate the latter three by running a series of tests and choosing the best performing values. Therefore the only parameter we can't estimate is the index of interest, which simply defines what is the exercise to focus to.

The flow of Algorithm 3 is following. Source code files are first normalized for similarity and authorship detection separately. Then similarity is detected for a collection of documents belonging under same exercise with the function DetectSim, which forms a group of suspicious authors noted as A_{susp}. The similarity detection process is controlled by the parameter ε which acts as a threshold for the detection. For example $\varepsilon = 1.0$ means that documents must be exact copies in order to group them together. The function trainAndPredictAuthor trains our authorship identification model with previous documents that the author has written, and then predicts who are the most likely authors from the set A for ith exercise noted as A_{auth}.

Our final result is the intersection between sets A_{susp} and A_{auth}, the results of similarity detection and authorship identification. Our intuition behind this can be shown with the following example. Let there be three authors $a, b, c \in A$ and three exercises under detection $d_a, d_b, d_c \in D$, where D contains also previous submissions for each author. Let there also exist a similarity detection phase able to cluster perfectly when document similarity is over the threshold ε, and authorship identification model trained with 100% accuracy so that the expected error when classifying any of the documents d_a, d_b, d_c as is minimal. If the clustering result is that $\omega_1 = \{d_a, d_b\}$ and $\omega_2 = \{d_c\}$ implying authors a and b are suspects as they share too much structural similarity, and the identification predicts $\hat{f}(d_a) = a$, $\hat{f}(d_b) = a$ and $\hat{f}(d_c) = c$, then we have verified that authors a and b have a high chance for a case of plagiarism as their submissions are too similar and the style of a document send by author b matches more the style of the author a. We suggest that author a has probably shared the document to author b, but because in this work we ignore the direction of plagiarism, both cases should be reviewed equally by a human expert *i.e.*, we treat sharing as equally serious offense as copying.

4.1 Assumptions

We mainly focus on academia and especially to programming courses that are offered by universities. Following five assumptions are defined to simplify the problem of plagiarism detection by allowing us to focus only on plagiarism that happens in a closed environment and within a closed set of documents.

In-class plagiarism Plagiarism has occured only within a specific course implementation. Let $\mathcal{P}(A)$ be a powerset of students within offered courses

in a university. We are only interested about a set of students referred as authors A attended in a specific course c *i.e.*, a subset $A_c \subseteq \mathcal{P}(A), A_c \neq \emptyset$. The corpus D_c is built by gathering every submission done by students in the course $\forall a \in A_c$ and a set of documents belonging to individual student is defined as $D_a = \{d \mid d \in D_c, a = auth(d)\}$.

Exercise focus Let $E_c = \{e_1, e_2, ..., e_n\}$ be a set of exercises for a course c, then submissions for a single exercise is represented by a subset $D_{c,e} \subseteq D_c$. With this assumption, we focus the plagiarism detection to submissions done to a single exercise at a time *i.e.*, plagiarism can happen only between submissions done to a single exercise, not over exercises.

Single author Every source code $d \in D_c$ is assumed to have a single author $a = auth(d), a \in A_c$. This allows us to assume that every source code submission is done as individual work, and all results that suggests otherwise implies excessive collaboration or plagiarism.

Plagiarism direction Let a file d_i be plagiarized from d_j: $d_i \xrightarrow{plag} d_j$. We treat this as same as the opposite direction $d_i \xleftarrow{plag} d_j$, making the direction of plagiarism unimportant. This means that we treat both cases sharing and copying, as an act of supporting plagiarism.

Expert interference We believe that no system can be accurate enough to autonomously accuse students of plagiarism. However, this is doable when some form of human judgment is added to the model. In principal this means that the model can make predictions about cases of plagiarisms which we call *suspects*, but the human expert must make the *allegation* of plagiarism based on the results and after questioning the students. Having guidelines about what is considered as plagiarism and how such cases are handled[6], helps both students and teachers to understand what the institution means when it accuses somebody of plagiarism.

4.2 Data set

Our approach is aimed to the MOOC setting which is for example used by undergraduate-level programming courses *Introduction to Programming* (OHPE) and *Advanced course in Programming* (OHJA) in University of Helsinki. We use three authentic data sets in Java language; students submissions done to both of latter courses during the implementation in fall 2016 and data from SOCO task from 2014. Both OHPE and OHJA includes proven cases of plagiarism, but to avoid any bias more specific information

[6]University of Helsinki's guidelines: `https://blogs.helsinki.fi/alakopsaa/?lang=en` Accessed 9th May 2018

about them is kept hidden as a golden standard until the final evaluation and we also do not have access to original student identifiers *e.g.* student numbers. SOCO's training dataset on the other hand, contains prelabeled document pairs that have conducted plagiarism which all have been discovered by human experts, and test dataset pairs labeled by majority of the submitted competitors [47].

To implement our models, we first use SOCO to train and evaluate our similarity detection model, then train and test authorship identification with OHPE and OHJA. Our proposed model is built based on these results and plagiarism is detected individually for both courses. The reason to use SOCO for similarity detection is simply that it's the only data set that contains fully labeled cases of plagiarism. OHPE and OHJA contains multiple files per author making author identification possible, but they hide the plagiarism cases. Therefore we make use of both sets and consider our model to be successful if it has a high precision, minimizing the amount of false-positives *i.e.*, false plagiarism accusations, and maximizing the amount of true-positives *i.e.*, true plagiarists. As this setting means that we need a high precision and a high recall, we resort to balancing between these metrics using the F_1-score.

Course overview

OHPE and OHJA shares the same structure; students first register to automatic scoring system called *Test My Code* (TMC) [58] which also distributes the exercises as an plugin to *NetBeans IDE*, then independently work during seven weeks by completing programming exercises within deadlines [56]. Both of these courses follow *Extreme Apprenticeship method* [57]; theoretical material is available online for students, students learn by doing *i.e.*, there exists mandatory programming exercises, weekly exercise sessions are available for those who require assistance, instructors can give feedback and students are able to track their process. In addition, there are exercises in which students are required to have a pair to program with. This is referred as *pair programming*.

Students earn points from exercises depending if all tests were successfully passed via TMC, and complete an exam at end of the course which is a programming exam that ultimately decides if a student has learned the minimum level of knowledge required to pass the course. The exam in fall 2016 was a home exam, meaning that students were able to do it individually wherever they wanted to. As there were no mandatory lectures, students were able to pass the whole course working individually without any physical attendance. Students were informed at the beginning of the course and in the course materials, that plagiarism is prohibited.

SOCO overview

Source code reuse (SOCO) data is from a 2014 competition *PAN@FIRE*, where two sets were given to detect monolingual source code re-use [47]. SOCO2014 offered a train and a test set for competitors, which contained files written in C++ and Java by various authors. The training set contains source code files and annotations made by three experts flagging pairs that are considered as plagiarized. Test set on the other hand, contains six individual scenarios labeled by majority voting from multiple submissions. The competitors were asked to retrieve the pairs with plagiarism, but the direction was completely ignored, meaning that they did not have to show who was the plagiarist and who was the sharer.

SOCO contains mainly submissions to a single exercise and documents, that are transformed from C to Java [47]. As only the plagiarized file pairs are annotated and SOCO has been used successfully used in other studies [13, 19, 23, 24], we will use SOCO to train and evaluate our similarity detection model. The number of authors is not explicitly reported in SOCO, so we make a simplifying assumption that there exist one file per one author and also that all submissions are for the same task.

Corpus statistics

As we are going to focus strictly to Java language, we only use the Java-specific part of SOCO, whereas OHPE and OHJA are fully utilized because they only contain Java files. Some non-transformative steps has been made beforehand to both OHPE and OHJA; exams are added to data set and submission containing multiple files are concatenated into one file. This allows us to assume also in OHPE and OHJA that there exist one file per submission, and we also get the benefit of having exam submissions where plagiarism is absolutely not allowed and where students work under time pressure. Exam submissions are thus picked as the main target for our plagiarism detection approach to eventually report how accurately it can perform. Statistics for all these three corpora are reported in Table 14, only applying modifications given above.

Table 14: Descriptive statistics for the unprocessed corpora. SOCO has been divided into three related corpora: train (T), test scenario without plagiarism (C1) and test scenario with plagiarism (C2). Bold values represents maximum value per metric.

Metric \ Corpus	SOCO-T	SOCO-C1	SOCO-C2	OHPE	OHJA
Authors	259	124	88	**316**	270
Exercises	1	1	1	**151**	92
Documents	259	124	88	**33 363**	15 196
Documents per author AVG.	1	1	1	**106**	56
Synthetic	Partly	Partly	Partly	No	No
LOC min	**12**	7	7	1	1
LOC AVG.	149	**155**	144	44	109
LOC max	**1696**	1398	661	679	637
Expression AVG.	63	**76**	67	17	38
Character AVG.	**3898**	3848	3751	1139	2794

Table 14 reports ten different metrics: number of total authors, exercises and documents; does the corpus contains synthetic data; averages for documents per author, character count, lines of code (LOC) and expressions ending to semicolon; and lastly minimum and maximum line counts. We can see from the Table 14, that SOCO has the smallest amount of authors but the tasks are more complex indicated by the largest LOC, amount of expressions and the average character count. When comparing OHPE to OHJA, OHPE has on average smaller submissions than OHJA, which is mostly due to OHPE having easier tasks due to being the introductory course where students are not expected to know anything about programming beforehand. OHPE also has the most largest average document-to-author ratio (106) compared to SOCO (1) and OHJA (56), making it the most richest data set when it comes to having a large amount of submissions per author. Comparing to other corpora presented in Section 3.3, our OHPE corpus is one of the largest with OHJA. They both have over four times as many authors than any of the corpora used in other studies.

A problem however arises when average line count with respect to the exercises is visualized for both OHPE and OHJA. Figure 6 visualizes this by histograms, where bin sizes are set to 50. From Figure 6 we see that majority of the submissions for OHPE has under 100 lines of code. This can create an issue, as there exists tasks where the submission can only contain a few dozen lines meaning that the similarities between solutions will be naturally high as the solution spaces of these tasks are very limited.

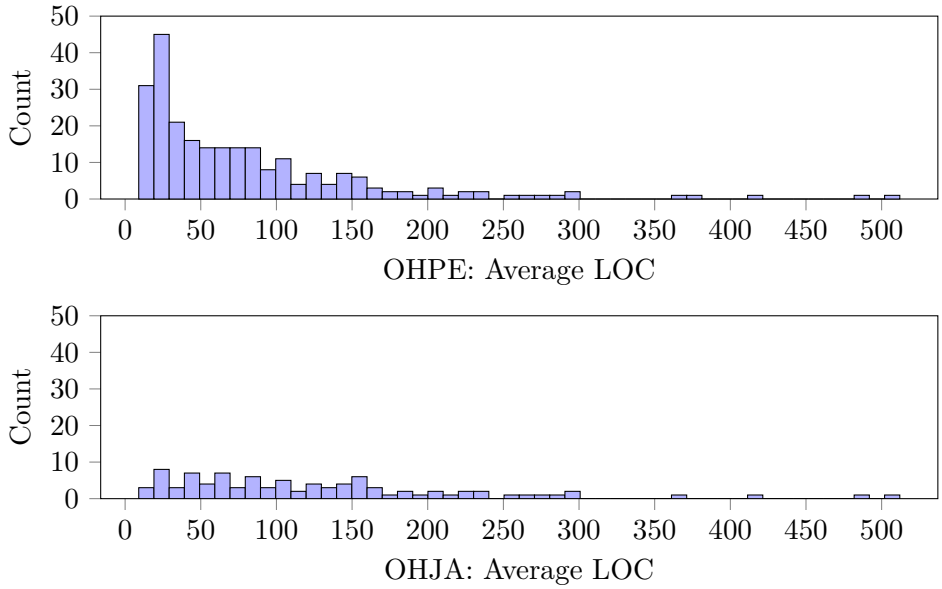


Figure 6: Histograms showing average line of count per exercise for OHPE (top) and OHJA (below). OHJA has more evenly distributed length of submissions, where as OHPE's submissions are mostly under 100 lines in length.

4.3 Document normalization

We utilize same approaches as studies reviewed in Section 3.4 to minimize the variance between documents by using normalization. The benefit of normalization is that it reduces the vocabulary size by unifying language structures which are unimportant. However, with normalization we can also emphasize certain aspects. In case of similarity detection we want to preserve as much structural information as possible, and in case of authorship identification the students author profile must be captured. This means we can ignore all stylistic preferences in similarity detection and all structural information in authorship identification, as they share different goals.

For similarity detection we transform every document into a token stream by first parsing the program with a parser and turning it into abstract syntax tree, then traversing the structure to get the stream as a string format. This method allows to capture the higher-level structure of the program, and still allows to handle it as a text. Also, it works against obfuscation strategies which were stated in Tables 1, 2 in Section 2.1, by ignoring certain structures. For example the parser will ignore all white spaces, comments, identifier names and it standardizes loop names, meaning it works against levels 1,2 and 5 of Table 2. The parser we made only works with Java and is heavily inspired by the one used in JPlag [45]. The complete list of tokens is seen in

Appendix B, which shows also the equivalencies to generate certain tokens. For example all loop constructs generate a single token "LOOP{" to indicate start of the loop, which normalizes the documents to preserve the underlying similar logic behind them.

Table 15 shows the corresponding token stream for the program A in Appendix A, where one can see how much information is discarded from the source code as we only keep the crucial structural information. It allows us to reduce the size of the possible vocabulary and generalize documents, as for example changing all integer values of the example source code leaves the token stream completely intact. Same goes for changing of the variable names, as they are not presented in any way in the token stream.

Table 15: Token stream generated from the example source code in Appendix A. No literal values are being saved to generalize documents as much as possible.

Original source code	**Token stream**

`public class A{`
 	
`  public static void main(String[] args){`
`  int a = 5;`
`  int b = 10;`
`  int c = 2;`
`  double d = (a + b + c)/(double)3;`
`  System.out.println(d);`
`  }`
`}`
	`CLASS{`
	`VOID`
	`METHOD{`
	`VARDEF`
	`ASSIGN`
	`VARDEF`
	`ASSIGN`
	`VARDEF`
	`ASSIGN`
	`VARDEF`
	`ASSIGN`
	`APPLY`
	`}METHOD`
	`}CLASS`

For authorship identification, normalization method we apply uses the same idea as in [9,10]. We discard all comments and normalize literal values to remove any possible notion of the original author, like unique student number or name in comments. The purpose behind normalization for authorship identification is therefore to leave the original document as intact as possible, maintaining the preferences that the programmer might have for variable naming or spacing. An example of the normalization procedure is given in Table 16 for the same program used in Table 15, where one can see that all numerical values have been transformed under a single dollar token `$`.

Table 16: The result of normalization procedure for the authorship identification. All literals have been mutated.

Normalized code

```
public class A{

     public static void main(String[] args){
        int a = $;
        int b = $;
        int c = $;
        double d = (a + b + c)/(double)$;
        System.out.println(d);
     }
}
```

4.4 Document representation

To represent every documents as vector, we use information retrieval techniques introduced in Section 2.4. Plagiarism detection is therefore done first by converting document into vector space model after the normalization. In both similarity detection and authorship identification, terms are first extracted, which in our case means all possible n-grams with respect to vocabulary $\mathbb{V}$. The only difference being that in similarity detection the vocabulary is formed using every document as a token stream, where as authorship identification uses only part of the complete data to form the available vocabulary *i.e.*, the training data.

To overcome the problem with varying document length and frequently appearing terms, we apply TF-IDF weighting introduced in Section 2.4.1. Table 17 shows an example of term extraction for similarity detection using word level 2-grams for program A in Appendix A. All TF-IDF weights have been normalized using Euclidean norm, formulated as follows:

$$\frac{\mathbf{x}}{\sqrt{\sum_{i}^{|\mathbb{V}|} x_i^2}} \tag{4.1}$$

Table 17: Similarity detection term extraction for document A. Terms are word-level 2-grams extracted from the token stream, whereas TF-IDF weights have been normalized and values rounded at two decimal places.

Term	**Raw frequency**	**TF-IDF weight**
`APPLY }METHOD`	1	0.14
`ASSIGN APPLY`	1	0.18
`ASSIGN VARDEF`	3	0.55
`CLASS{ VOID`	1	0.14
`METHOD{ APPLY`	0	0.00
`METHOD{ VARDEF`	1	0.18
`VARDEF ASSIGN`	4	0.74
`VOID METHOD{`	1	0.14
`}METHOD }CLASS`	1	0.14

An example of how the calculation is done in Table 17 is given next. To get the value 0.18 for a term `ASSIGN APPLY` in document A one sees first that the value of tf is 1 from Table 17. The *idf* is formed by dividing number of documents with the number of total term frequency over all documents, and taking a logarithm *i.e.*, $idf = \log(N/df) = \log((1+3)/(1+2)) + 1 \approx 1.29$. Note that we add extra ones to avoid division with zero and to diminish the effect of terms appearing only in training set. Now tf-*idf* is simply $tf \cdot idf = 1 \cdot 1.29 = 1.29$. Finally, after calculating non-normalized weight for each term, we can derive the value 0.18 dividing 1.29 with the Euclidean norm over the weights which gives $tf\text{-}idf_{norm} = 1.29/\|\mathbf{w}\|_2 = 1.29/6.98 = 0.18$.

The vocabulary $\mathbb{V}$ that forms the set of possible tokens in Table 17, is the union between every token appearing in three example documents *i.e.*, $\mathbb{V} = \bigcup_{i=1}^{3} V_i$ where $V_i = \{t_1, t_2, \cdots, t_n\}$. Therefore some terms may appear zero like the term `METHOD{ APPLY` for the document A in Table 17, as it exists only in the token stream of document C. The smoothing we apply in Equation 2.15 prevents the complete product to become zero. Terms like `ASSIGN VARDEF` and `VARDEF ASSIGN` have a high weight as they mostly appear in document A, implying that document A has more variable assignments than document B or C, which is true when one looks at the raw source code documents.

With our approach, we can now represent document as a vector of weights *e.g.* document A as $\mathbf{x} = [0.14, 0.18, \cdots, 0.14, 0.14]$, where the dimension of $\mathbf{x}$ is the size of vocabulary $\mathbb{V}$. The visualization of these three programs as vectors of weights can be seen in Figure 7, where it is clear that program C is the outlier whereas A and B share more similarities between each others.

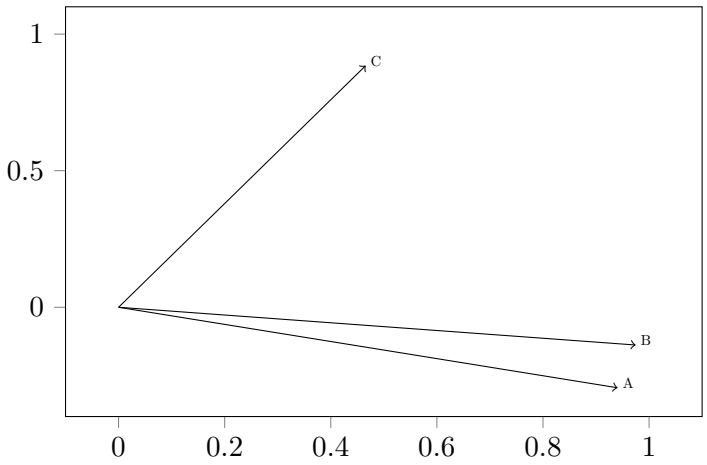


Figure 7: Three sample programs from Appendix A visualized in two dimensions. TF-IDF weights have been calculated from the token streams.

4.5 Similarity detection

After submissions for a given exercise have been normalized into a token stream, we apply the DetectSim function of the Algorithm 3, which retrieves set of authors we call suspicious authors. These authors share a lot of structural similarity to each others within a given task, implying that there it is possible that plagiarism might have occurred within this set.

The DetectSim in other words, is the similarity detection method of our study, where we first calculate the vector similarity to form a distance matrix $\mathbf{M}$. In this matrix $\mathbf{M}_{i,j}$ implicates the similarity between documents d_i and d_j. Then, we calculate the similarity by using cosine similarity introduced in Section 2.4.2, which was also extensively utilized by other studies in Section 3.4. Lastly, we apply DBSCAN clustering to the values in similarity matrix $\mathbf{M}$ to form a groups of suspicious authors. The pseudocode for the DetectSim function can be seen in Algorithm 4, which is dependant from two parameters: length of n-grams and similarity threshold ε. These two hyperparameters are tuned with SOCO data set before the final evaluation. The overall flow of operations is following: first we extract all word level n-grams and turn the documents into raw term frequencies, then terms are weighted using TF-IDF and cosine similarity is calculated between every document. Finally DBSCAN clustering algorithm is used to form clusters of similar documents. Because we know every author of each document and we assume single authorship, these clusters are identical to clusters of authors.

Algorithm 4 Detecting suspicious authors.

Require: Set of authors A
Require: Set of documents D belonging to authors $a \in A$
Require: Every document $d \in D$ is represented as a token stream
Require: Preferred length of word level n-grams $n \in \mathbb{N}$
Require: Minimum rate of similarity $\varepsilon \in [0, 1]$
Assume: $MinPts \leftarrow 2$
  procedure DETECTSIM(A, D, n, ε)
    $\mathbf{X} \leftarrow$ EXCTRACTNGRAMS(D, n)
    $\mathbf{W} \leftarrow$ TFIDF($\mathbf{X}$)
    $\mathbf{M} \leftarrow$ COS($\mathbf{W}$)
    $\Omega \leftarrow$ DBSCAN($\mathbf{M}, \varepsilon, MinPts$)
    return Ω
  end procedure

Note that in Algorithm 4, the value of $MinPts$ is assumed to be value 2, as only two documents are needed to form a cluster of suspicious documents. This refers to a real life situation where two students have shared source code between each others.

4.6 Authorship identification

The second method we apply, is the author identification from a collection of source codes. Like in similarity detection, we apply this to one exercise at a time but as this model requires training, we define the training set to be all documents that the author has previously written. For example when considering a course which consist of seven weeks and the final exam, we can use all seven weeks per author to train the model *i.e.*, try to capture the preferred style of an author, and then predict the most likely author of a random sample from a collection of exam submission.

The algorithm for authorship identification we use is based on the probabilistic model Naïve Bayes from Section 2.5. We utilize n-grams which was a popular method among other studies in Section 3.4, because it captures preferences that the author might have when writing a program by using character-level information. That is also why we don't apply a lot of normalization for the authorship identification, as this information would be lost if too much transformation would be applied. The pseudocode for our authorship identification is seen in Algorithm 5, which is dependent from one hyperparameter, the length of character level n-grams. The value for it will be tuned using both data sets OHPE and OHJA, and choosing the value which performs best on average.

Algorithm 5 Detecting the most likely author for a source code.

Require: Set of authors A
Require: Set of documents D belonging to authors A
Require: Index of the exercise under detection $i \in \mathbb{N}$
Require: Length of character level n-grams $n \in \mathbb{N}$
  procedure TRAINANDPREDICTAUTHOR(A, D, i, n)
    $\mathbf{X} \leftarrow$ EXCTRACTNGRAMS(D, n)
    $\mathbf{W} \leftarrow$ TFIDF($\mathbf{X}$)
    $\mathbf{W}_{train}, \mathbf{y}_{train}, \mathbf{W}_{test}, \mathbf{y}_{test} \leftarrow$ SPLIT($\mathbf{W}, A, i$)
    $NB \leftarrow$ TRAINNAÏVEBAYES($\mathbf{W}_{train}, \mathbf{y}_{train}$)
    $A_{auth} \leftarrow$ PREDICT($NB, \mathbf{W}_{test}$)
    return A_{auth}
  end procedure

The remaining flow of Algorithm 5 is following. After the weight matrix $\mathbf{W}$ has been formed we split the data into training and test sets with appropriate classes $\mathbf{y}$, which indicates the authorship assignments. The split is done by treating the ith exercise as a test set, and everything before it as a training data. For example if the interest is the exam, which can be thought as the final task of the course, there are 135 exercises before it in OHPE and 79 in OHJA[7] that can be used to capture the individual style of an author. The appropriate training data is given to the Naïve Bayes algorithm in TRAINNAÏVEBAYES, which theoretical background is given in Section 2.5. The training of the Naïve Bayes algorithm allows therefore to estimate the probabilistic parameters inside the model. These parameters are reflected into the function PREDICT, being the maximum a posteriori probability (MAP) estimate, is then able make the author prediction.

4.7 Evaluation

We first introduce a metric called *accuracy*, which can be used in both binary and multiclass evaluation. Accuracy score, simply being the fraction between correct predictions and the total number of predictions, can be defined by using the confusion matrix given in Section 2.4.2 as

$$ACC = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.2}$$

Both models of our approach are first evaluated against the data. This means that the similarity detection part uses SOCO to tune its parameters (n-gram and ε) and evaluate the performance of our model. Evaluation happens by reporting average precision and F_1-metric of document retrieval, and we mainly focus on the amount of correctly classified documents retrieved.

[7]Values 135 and 79 are after pair programming tasks are filtered out from both sets.

After the hyperparameters for the similarity detection have been tuned, we compare it by calculating the agreement to the state of the art software plagiarism detection called JPlag [45]. The agreement with respect to the JPlag is based on the Jaccard similarity, which was given in Section 3.4 between two sets and we expect Jaccard similarity to be close to one, as our methods should get similar results as the JPlag. However, we can't say for sure that did JPlag retrieve all possible cases of plagiarism as we don't have direct access to true classes without going through every possible document in OHPE/OHJA. This means we can't calculate precision nor recall for the plagiarism detection, and we must resort to human judgement to base our final evaluation.

When evaluating the authorship identification, our classification problem is no longer binary. It's a multiclass classification problem, and in order to use F_1-score, it must be redefined. The multiclass-version of the F_1-score, which treats all classes equally, is called *macro-averaged* F_1 [50]. It's defined as

$$F_M = 2 \cdot \frac{\text{Precision}_M \cdot \text{Recall}_M}{\text{Precision}_M + \text{Recall}_M} \tag{4.3}$$

Where Precision_M and Recall_M are averaged over every class as

$$\text{Precision}_M = \frac{\sum_{c \in \mathbb{C}} \frac{TP_c}{TP_c + FP_c}}{|\mathbb{C}|} \tag{4.4}$$

$$\text{Recall}_M = \frac{\sum_{c \in \mathbb{C}} \frac{TP_c}{TP_c + FN_c}}{|\mathbb{C}|} \tag{4.5}$$

Using above metrics for a multiclass classification, we are able to tune the parameter n which controls the length of character-level n-grams. Tuning is done by first dividing both OHPE and OHJA into seven splits which corresponds each week, then taking the set of authors who submitted and using their previous work as a training data, finally predicting the authors of the last exercises and collecting calculating the average performance. For example, when we evaluate our authorship identification on the first week of OHPE, we take the subset of authors $A' \subseteq A$ who submitted to the last exercise of the first week. Then the last exercise is left out as the test data, and for each author $a \in A'$, we collect their submissions to form the training data.

Our final result will be a set of detected documents for both OHPE's and OHJA's exam tasks, and we will use a human expert who manually goes through the retrieved documents and classifies which ones she considers as real plagiarism. By using a human judgement, we get as unbiased and realistic evaluation as possible, but also information about the decision process. When the human expert has gone through all documents and

evaluated them, we calculate following four metrics to score our final result: number of true positives, number of false positives, detected cluster sizes and Jaccard similarity.

5 Results

Following sections describe the results we gathered during the evaluation of our models. All results are generated using Python version 3.6.0[8] and scikit-learn version 0.19.1[9].

As explained in the Section 4.7, we first evaluate both models individually and lastly combine the results to create a final prediction which is evaluated by a human expert. Our similarity detection is trained with SOCO data set and authorship identification with OHPE and OHJA without using the exams. A summary of these exam tasks is given below.

Table 18: Submission count and average line count for exam tasks. A refers to OHPE and B to OHJA.

Task	1.A	2.A	3.A	4.A	1.B	2.B	3.B
Submissions	244	242	227	240	200	198	197
Avg. LOC	37	39	47	110	160	86	150

It's clear from the Table 18 that OHJA's tasks are more longer than OHPE's. Some of the tasks of OHPE's exam have a very low average line count that creates a challenge for the detection.

5.1 Document similarity

We start evaluating our similarity detection by tuning the hyperparameters n for n-gram length and ε for the epsilon-range *i.e.*, minimum distance to other documents. By evaluation various values for the hyperparameters, we try to find a general model which could perform well with any dataset. Results are gained by turning all documents into binary vector based on the SOCO labels *i.e.*, vector $\mathbf{y}$ where $y_i = 1$ and $y_j = 1$ if ith and jth documents are reported as plagiarized pairs. Our predictions are compared against this golden standard.

Table 19 shows averaged F_1-score, weighted by label counts, for the SOCO-T data. One can see from it that the F_1-score is highest when $n \in [4, 7]$ and $\varepsilon \in [0.4, 0.6]$. However, allowing 40-50% dissimilarity between documents means that there is a high chance for false-positives, especially when submissions are relatively short and the task is well-defined like in OHPE and OHJA, meaning that the solution space for a given task can

[8] `https://www.python.org/` Accessed 14th May 2018

[9] `http://scikit-learn.org/stable/` Accessed 14th May 2018

be limited. Therefore to avoid overfitting similarity detection to SOCO's training data, we use also the test sets of SOCO C1 and C2.

Table 19: Average F_1-score for n-gram length and ε-range for SOCO-T containing 115 cases of plagiarism. The smaller the ε-range is, the more similar documents have to be. F_1-scores close or over 0.8 are bolded.

Epsilon \ N-**gram**	1	2	3	4	5	6	7	8	9	10
0.1	0.31	0.69	0.63	0.60	0.59	0.56	0.55	0.55	0.52	0.52
0.2	0.28	0.59	0.73	0.66	0.63	0.62	0.60	0.59	0.56	0.55
0.3	0.27	0.43	**0.78**	0.73	0.70	0.67	0.64	0.63	0.59	0.58
0.4	0.27	0.31	0.72	**0.81**	**0.78**	0.72	0.71	0.69	0.65	0.64
0.5	0.27	0.29	0.57	**0.80**	**0.81**	**0.80**	**0.81**	**0.78**	0.77	0.74
0.6	0.27	0.27	0.39	0.71	**0.83**	**0.89**	**0.90**	**0.86**	**0.85**	**0.85**

Table 20: Precision with respect to plagiarized class, ranging various n-gram lengths and ε-ranges for SOCO-T. Values close or over 0.9 are bolded.

Epsilon \ n-**gram**	1	2	3	4	5	6	7	8	9	10
0.1	0.45	0.77	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
0.2	0.45	0.53	**0.98**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
0.3	0.44	0.48	0.83	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
0.4	0.44	0.45	0.63	**0.87**	**0.97**	**0.98**	**0.98**	**1.00**	**1.00**	**1.00**
0.5	0.44	0.45	0.54	0.75	**0.90**	**0.92**	**0.97**	**0.98**	**1.00**	**1.00**
0.6	0.44	0.44	0.47	0.62	0.77	**0.87**	**0.94**	**0.93**	**0.95**	**0.96**

We see from the Table 20, that as we grow the number of n-grams, the precision starts converging to 1.00. Having a high precision means that the set of retrieved documents contains high number of true positives, as we have effectively minimized the amount of false positives, and no document is falsely accused of plagiarism. This happens because longer n-grams grow the size of vocabulary $\mathbb{V}$, thus making already dissimilar documents even more dissimilar and allowing the threshold to grow. The most smallest n-gram having a near perfect precision over plagiarized class is when $n = 3$ and $\varepsilon \in [0.1, 0.2]$. This kind of high similarity value ranging between 80-99% is also used in other studies [12, 13, 27, 61].

One sees from the following table that the F_1-score starts to deteriorate in all cases, when no plagiarism occurs between a set of documents. One must either have a high similarity threshold or increase the n-gram length to get a high F_1-score, because having a low threshold quickly introduces false

positives. The model thus becomes too sensitive and retrieves documents where similarity has occurred naturally, adding work for the human expert who must go through the detected pairs and label them again.

Table 21: F_1-score for SOCO-C1, which contains no cases of plagiarism. False-positives are introduced as the threshold gets lower.

Epsilon \ n-**gram**	1	2	3	4	5	6	7	8	9	10
0.1	0.24	**0.94**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**
0.2	0.11	0.56	**0.98**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**
0.3	0.06	0.38	**0.95**	**0.99**	**0.98**	**0.99**	**0.99**	**0.99**	**0.99**	**0.99**
0.4	0.03	0.20	**0.87**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**
0.5	0.03	0.16	0.59	**0.95**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**
0.6	0.02	0.08	0.29	**0.88**	**0.96**	**0.98**	**0.98**	**0.98**	**0.98**	**0.98**

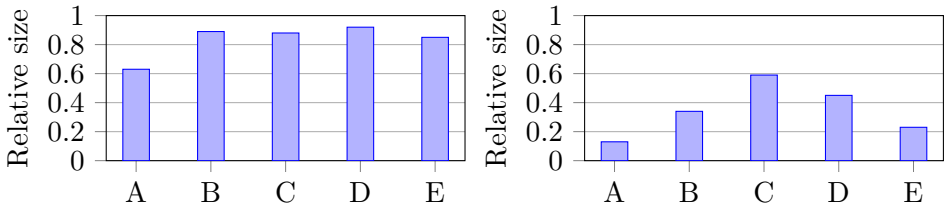
Table 22: F_1-score for SOCO-C2, which contains 28 cases of plagiarism.

Epsilon \ n-**gram**	1	2	3	4	5	6	7	8	9	10
0.1	0.34	**0.92**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
0.2	0.27	0.57	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
0.3	0.20	0.38	**0.92**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
0.4	0.15	0.31	0.75	**0.97**	**0.97**	**0.99**	**0.99**	**0.99**	**0.99**	**1.00**
0.5	0.15	0.27	0.47	**0.91**	**0.97**	**0.97**	**0.97**	**0.97**	**0.99**	**0.99**
0.6	0.15	0.22	0.33	0.78	**0.92**	**0.97**	**0.97**	**0.97**	**0.97**	**0.97**

As in Table 21, Table 22 shows that having $n = 3$ with similarity threshold being around 80%, yields one of the highest F_1-score with the lowest n used.

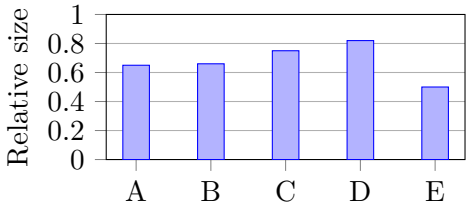
Taking the best scoring models over all scores for each n-gram and excluding $n \geq 8$ as they aren't improving the performance compared to $n = 7$, we end up with five model candidates A ($n = 3, \varepsilon = 0.2$), B ($n = 4, \varepsilon = 0.4$), C ($n = 5, \varepsilon = 0.5$), D ($n = 6, \varepsilon = 0.6$), E ($n = 7, \varepsilon = 0.6$). As we compare these five models by their relative size of the largest cluster across all formed clusters of OHPE's exam tasks, we see in Figure 8 that in majority of the cases model A's largest cluster has the lowest relative size. In Figure 8a models B and D have relative size close to 1.0, meaning that the largest cluster contains almost every single retrieved document. The most probable cause for this is that all submissions share natural similarity, caused by the restricted task description or that the solution space for a given task might be very limited. Thus this kind of super cluster can contain a lot of false

positives in form of similar documents which are not necessarily plagiarized, but rather correct similar solutions for the task.

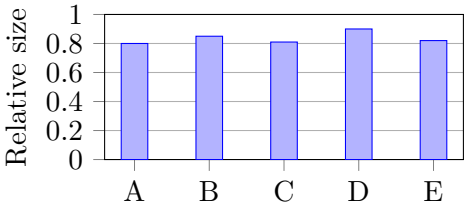


(a) OHPE 1st exam task.

(b) OHPE 2nd exam task.

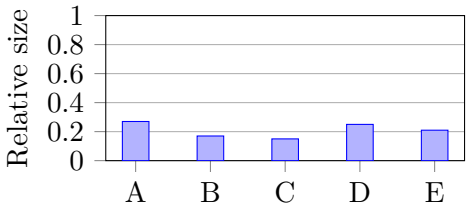


(c) OHPE 3rd exam task.

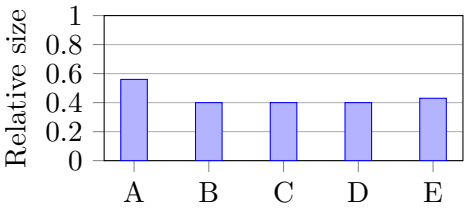


(d) OHPE 4th exam task.

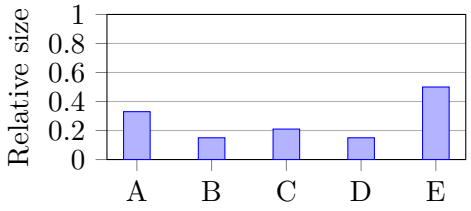
Figure 8: Relative size of the largest cluster in OHPE.



(a) OHJA 1st exam task.



(b) OHJA 2nd exam task.



(c) OHJA 3rd exam task.

Figure 9: Relative size of the largest cluster in OHJA.

Figure 9 shows same results for OHJA, which is an advanced course where a lot more programming skills are required from the students. This

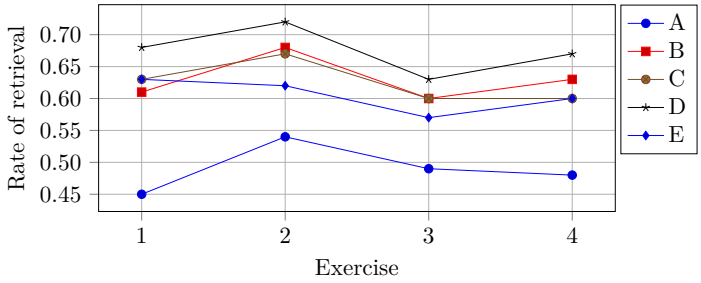
allows the tasks to be more difficult and longer, and as we see, the relative size has gone down in all cases compared to OHJA. Cluster sizes are a lot smaller as exam tasks are more open-ended and more demanding. In other words, exam tasks have a range of multiple solutions and ways to do them, which minimizes the natural similarity between documents. This can be seen as a trend where none of the models now suffer from forming clusters containing majority of the retrieved documents, as the size is around 0.5 (50%) at maximum.

Retrieving majority of the documents is not optimal for plagiarism detection as these documents often needs to be manually inspected at the end. When inspecting the retrieval rate *i.e.*,

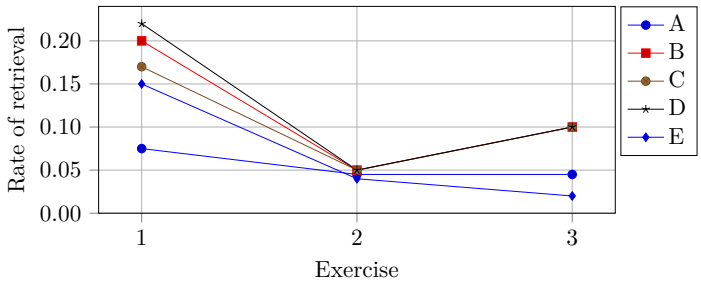
$$\text{Retrieval rate} = \frac{\#\text{Documents retrieved}}{\#\text{Documents in total}} \tag{5.1}$$

We can see that this rate should not get values near 1.0, as this would mean that all documents are detected as plagiarism, which is very unlikely in cases with hundreds of documents. Thus models with a very high retrieval rates are almost guaranteed to have a high number of false positives.

The retrieval rates of all models can be seen from the Figure 10.



(a) OHPE's exam tasks and the retrieval rates. Most models consider around 60-70% of all document to be over the similarity threshold.



(b) OHJA's exam tasks and the retrieval rates. Models have similar retrieval rates and the rate of retrieval is low.

Figure 10: Retrieval rates of all models across every exam task. The model A keeps the lowest retrieval rate overall.

The Figure 10a shows how the the model A keeps the retrieval rate lowest around 50%, meaning that half of the documents contain too much similarities between each others, and as said before it's very unlike so many documents are plagiarized. When comparing the model A to other models, they claim the rate of plagiarized documents to be even higher, which can not be true. In Figure 10b, the models agree quite well, only having some level of disagreement with first exam exercise of OHJA. The second exercise shows good agreement, as all models have near 5% retrieval rate.

Results on similarity detection show that tuning the two parameters n and ε is very data dependent as choosing the best performing combination might lead to very different results for other data sets. In our case, we choose the model A ($n = 3, \varepsilon = 0.2$) for the final evaluation, because that model had a decent F_1-score in SOCO-T the precision for SOCO-T was nearly perfect, and F_1 for both SOCO-C1 and SOCO-C2 were near 1.00. The model A also kept the largest cluster relatively small compared to other models and the retrieval rate for both OHPE and OHJA was the lowest, implying it could maintain a low rate of false positives. Keeping the rate of false positives minimal is more valuable us than retrieving every single plagiarism case, so we allow the model's detection rate to suffer with the benefit of having a high precision.

To get perspective on how well our chosen model compares to the state of the art Java plagiarism detection tools, we first run JPlag detection for OHPE's and OHJA's exam tasks, then run our model for the same set of exercises and finally report the Jaccard similarity between the set of detected documents. For the JPlag, we use its default parameters and collect all document pairs where the reported similarity is over 80%. This threshold is chosen purely based on the fact that 70% felt too low and 90% too high, therefore there is a possibility that some other threshold values could work better for JPlag.

Following tables show results for both OHPE and OHJA with five metrics: documents detected by JPlag, documents detected by our chosen model, size of the intersection between the set of detected documents, number of unique documents retrieved in total and the Jaccard similarity score.

Table 23: Retrieval metrics for model A compared to JPlag with OHPE's exam tasks.

Exam question	1.	2.	3.	4.
JPlag - Documents retrieved	127	134	106	156
Model A - Documents retrieved	109	130	111	114
Common documents	98	109	95	102
Unique documents	138	155	122	168
Jaccard similarity	0.71	0.70	0.78	0.61

Table 23 shows how our model agrees quite well with JPlag, as around 100 documents per exam task are shared. But even with the state of the art tool like JPlag, one retrieves a lot of documents with a threshold like 80% for OHPE as the retrieval rate with JPlag for all OHPE's tasks is around 50%.

Table 24: Retrieval metrics for model A compared to JPlag with OHJA's exam tasks. JPlag retrieves just a few documents when using 80% threshold.

Exam question	1.	2.	3.
JPlag - Documents retrieved	2	2	0
Model A - Documents retrieved	15	9	9
Common documents	2	2	0
Unique documents	15	9	9
Jaccard similarity	0.13	0.22	0.00

The retrieval rate for OHJA's tasks for all our model candidates was very low, and this same result is reflected in Table 24 where JPlag retrieves only two documents or no documents at all. It seems that our model retrieves more documents than JPlag, but without a human interference it's impossible to say which one of the models is more correct. However, the retrieval from tasks 1. and 2. share the same two documents that JPlag detected, meaning that our model performs similar to JPlag but the scoring it produces is more consistent which can be seen when we inspect the third task where the level of agreement was the lowest.

As we inspect every pair our model retrieved from OHJA's third task and compare the similarity scores to JPlag, we get five unique document pairs which are denoted here as $p_i, i \in [0, 5]$, formed by a total of nine documents. The results are visible in Figure 11.

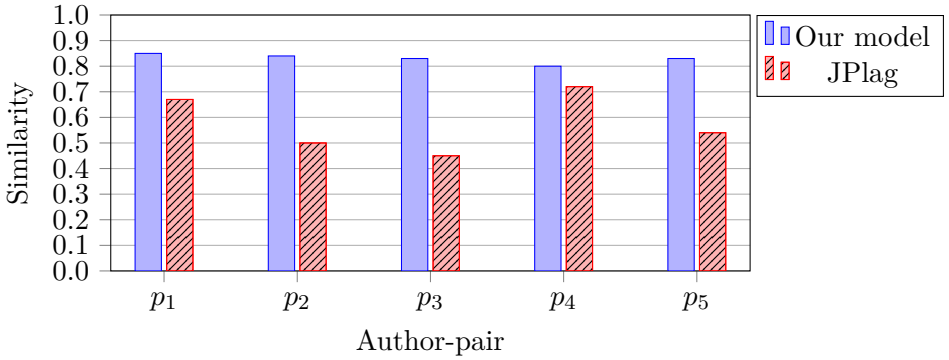


Figure 11: The difference between JPlag's reported similarity value and our model for OHJA's third exam task.

Figure 11 visualizes how our model keeps the similarity score near 80% for every pair, whereas JPlag's score varies. The most similar scores are with

pairs p_1 and p_4, where the difference is around 0.1 compared to our model. In other cases, it seems that JPlag can produce more specific results, because the comparing process differs from ours. We use the whole vocabulary to produce the similarity score, whereas JPlag forms the score by string matching the token streams.

We have now trained and evaluated our similarity detection model. The model we chose uses n-gram length of three, and retrieves any document where the calculated similarity value is above the 80% threshold, which is reflected as ε-range of 0.2 in our clustering method. Our model was compared to JPlag and the retrieved documents were mostly the same, but there were some variance in number of documents retrieved. In following section, we train and evaluate the second model, the authorship identification.

5.2 Authorship identification

Our authorship identification model requires one parameter to be tuned, the length of character-level n-grams to be extracted. We tune this parameter based on the average F_1-score and accuracy over seven split points for both OHPE and OHJA. For every weekly split the final exercise is left out as a test data, 80% of the remaining data is used for training and 20% for validation. The training data is used purely to tune the model, whereas validation is used to find the best performing n-gram length. After the value for n has been found, we evaluate the final model with the test data.

Tables below show the splits we make, the number of students submitted to the last exercise of the week and the average profile size. The profile size refers simply to the amount of documents students have submitted before the split.

Table 25: OHPE's splits. Profile size grows naturally as students progress the course.

Week	1.	2.	3.	4.	5.	6.	7.
Students	230	239	189	174	127	138	53
AVG. Profile size	24	40	64	76	85	94	102

Table 26: OHJA's splits. The profile size is much lower than in OHPE.

Week	1.	2.	3.	4.	5.	6.	7.
Students	144	114	137	90	111	121	113
AVG. Profile size	11	21	30	36	43	50	53

We see in Table 25 how the amount of students varies quite a bit for the final week as only 53 students submitted. This is probably because students have calculated that they already got the points they need in order to pass the

course with the exam, so they skip the last exercise. The amount of students remains more stable in OHJA seen in Table 26, where the profile size grows more steadily.

For every split in OHPE we calculate the macro-averaged F_1 for the validation data, and these results are visible in following table.

Table 27: Macro-averaged F_1-score calculated for each validation set of OHPE.

Week / n**-gram**	1.	2.	3.	4.	5.	6.	7.
2	0.01	0.02	0.03	0.02	0.02	0.02	0.03
4	0.01	0.03	0.03	0.04	0.04	0.04	0.04
6	0.02	0.04	0.05	0.05	0.05	0.05	0.05
8	0.02	0.04	0.05	0.06	0.06	0.06	0.06
10	0.02	0.05	0.06	0.06	0.07	0.07	0.07
12	0.02	0.05	0.06	0.06	0.07	0.07	0.07
14	0.02	0.05	0.06	0.07	0.07	0.07	0.07

Table 27 shows how the model fails to predict the correct authors in a multiclass setting, where each document can be predicted only to one author. We see that the F_1-score slightly increases when $n \geq 10$ and when the used weeks grows *i.e.*, the submission amount per student grows. The same evaluation was also run for the OHJA, but the results were as poor as for the OHPE, and therefore they are not shown here. Based on these result we fix the n-gram length as 10 as it's the best overall result we got with the smallest n used, which also limits the size of vocabulary.

Figure 12 reveals how the vocabulary size grows when the n-gram length gets larger. Even by using a small value of n which keeps the vocabulary size smallest and should effectively capture *e.g.* the spacing used after operator symbols, gives poor results as seen in Table 27. However the problem with large vocabulary is that the training consists a lot of noisy features *i.e.*, features that could be dismissed completely, that the model is unable to find important features and weight them properly.

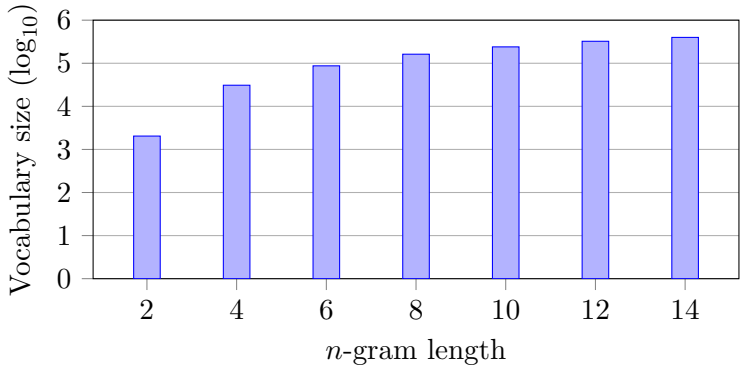


Figure 12: Vocabulary size in $\log_{10}$-scale with respect to character n-gram length. Vocabulary is formed using 80% of OHPE's tasks (training set) and its size for 10-grams is around 278 000.

As we look from Table 28 the ten most common n-grams formed from the training using OHPE's data set and their frequencies, we see how similar most features can be.

Table 28: Ten most frequent 10-grams encountered while training the Naïve Bayes model. All of them contain the same overlapping sequence which is the statement for printing in Java.

N-**gram**	**Frequency**
`System.out`	18643
`ystem.out.`	18643
`stem.out.p`	18643
`tem.out.pr`	18643
`em.out.pri`	18643
`m.out.prin`	18643
`.out.print`	18643
`System.ou`	18640
`out.println`	15171
`ut.println`	15171

In Table 28, most of the programs contain various sequences of the same print statement in Java language. These statements exist in almost every document, as many of the tasks in OHPE and OHJA require to print various values to the console in order to evaluate the correctness of the submission. All of these 10-grams can be considered as noise, because their informative value is close to zero as they are used in similar way in all documents. Features like these are problematic for our identification because as the vocabulary size grows, the vector representing the document is starting to contain mostly zeroes and the non-zero ones can contain a lot of non-informative duplicates

as are the 10-grams in Table 28.

We looked would the data be skewed while training *i.e.*, would some author have a majority of the documents, giving untrue prior probabilities for authors. However this was not the case as seen from the following plot.

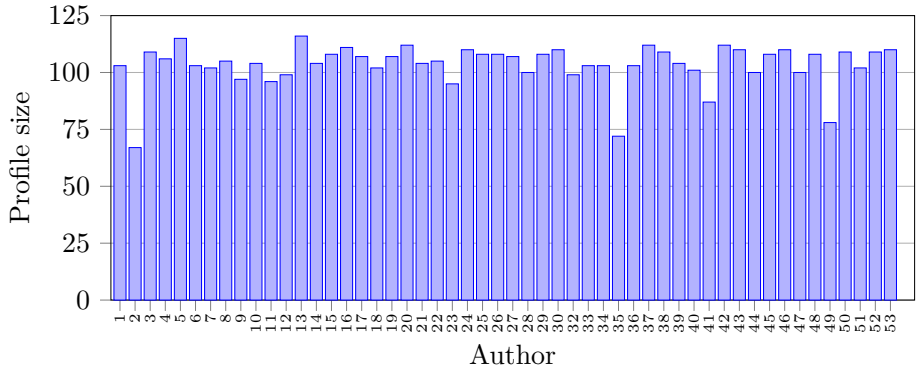
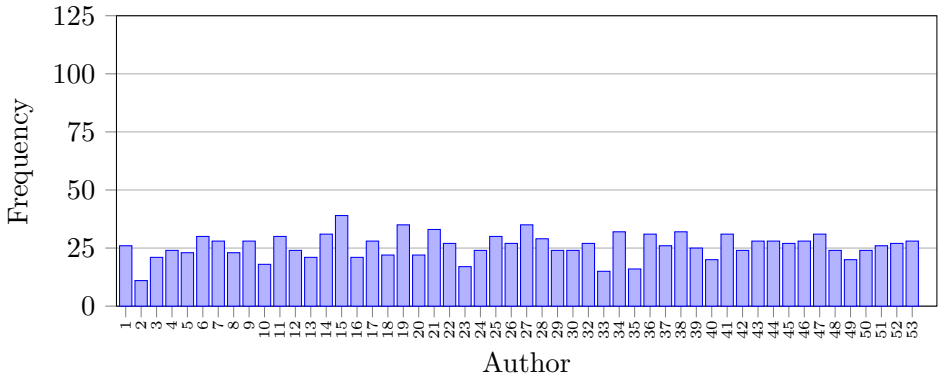


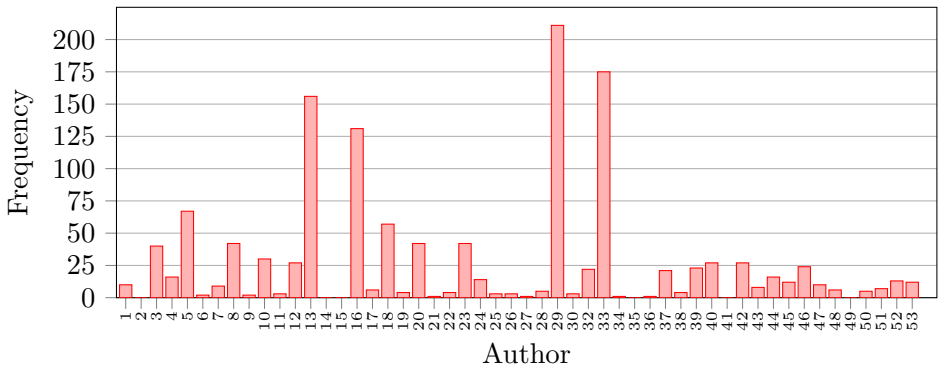
Figure 13: The profile size of each included author in OHPE's training set.

There are only four authors 2, 35, 41 and 49 who are below the average profile size in Figure 13, so we can't say that our data would have been skewed in the training process. The data is very evenly spread amongst the authors, as nearly all have around 100 documents for their profile size. When comparing these numbers to other studies presented during literature review in Section 3.3, we see that not only we have around ten times more documents per authors, but also our author pool is a lot larger. Having a lot of sample documents for each author should generate more distinct writing preferences, but in our case it's not the case as submissions seem to be too similar. Excessive similarity was a problem also in our similarity detection evaluation in Section 5.1.

To visualize the missclassification of our model, we formed the frequencies of true authors and the predicted authors in the validation data of OHPE. Figure 14 shows how the authorship identification should produce a uniform distribution of authors, where each author has around 25 documents classified for them. However, our model can't find enough unique stylistic preferences during the training, thus misclassifying majority of the documents to four authors. This result reflects the same observation that was made earlier about the excessive similarity of the documents, which leads to a situation where some amount of authors might be so close to other profiles, that there any not enough discriminating n-grams that could divide authors apart from each other. This problem is visualized in Figure 14 as spikes, where four authors become author archetypes who capture the writing style of everybody else.



(a) True author distribution. The number of samples for each class in the validation data is averaging around 25, with just a few outliers.



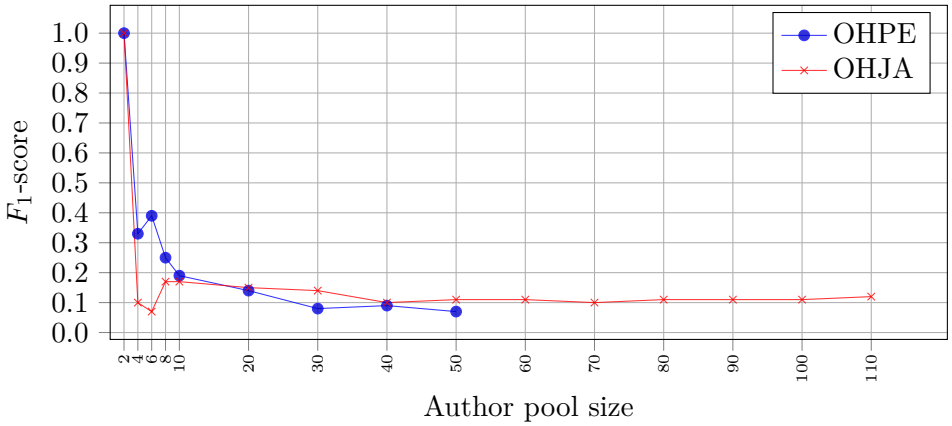
(b) Predicted author distribution. There are several authors without any documents classified to them, and four authors who have the majority.

Figure 14: The true distribution of authors in the validation data of OHPE (a) compared to the predicted distribution (b). Our classifier predicts most of documents to belong to just four unique authors.

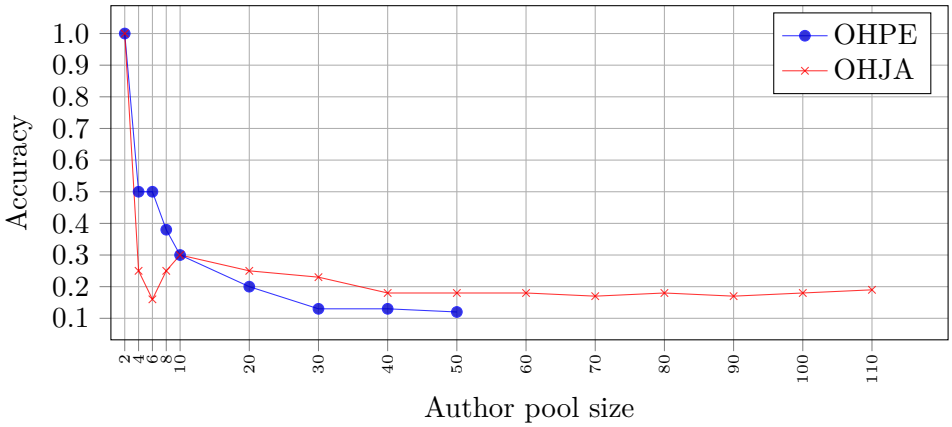
Inspecting the probabilities of the Multinomial Naïve Bayes which is trained with OHPE, the mean prior is 0.02 (2%) and the standard deviation 0.002. This means that the prior probabilities $P(y)$ are very close to each other so their influence is diminished at the prediction phase. The likelihood *i.e.*, probability of the ith feature appearing given the class $P(x_i|y)$, is also very small for every feature and class combination, as using 10-grams there are around 278 000 unique features. For every class the mean conditional probability is 4.1×10^{-6} and standard deviations for conditional probabilities are in range $[1.3 \times 10^{-7}, 2.0 \times 10^{-7}]$, showing again how similar all values are because our vocabulary is too large.

As we have now shown the results for training and validation, selected the n-gram length as 10 and looked some of the reasons why the model fails

to predict the author, we next present the results for the test data. It consist of the last exercises of OHPE and OHJA, and uses all possible data for the training phase, as shown in Table 25 and Table 26. We will restrict the author pool size to ease the problem and use the last week for both OHPE and OHJA to have a full data set, and these results for both F_1-score and accuracy are visible in Figure 15.



(a) F_1-score for OHPE's and OHJA's test set using 10-grams and varying the amount of possible authors.



(b) Accuracy for OHPE's and OHJA's test sets using 10-grams with various author pool sizes.

Figure 15: Evaluation results for the final authorship identification model. Our model is not able to predict the author at a satisfactory level.

We can observe how in Figure 15a, the F_1-score quickly deteriorates as the number of possible authors grow. This same observation can be seen

also in Figure 15b where the accuracy is shown. In both cases there exist fluctuation caused by random sampling when the author pool size is below 10. When the number of authors reaches 20, the F_1-score settles quickly around 0.1 regardless of the data set. Interestingly the F_1-score and accuracy remains around 0.1 and 0.2, implying that a portion of authors are always classified correctly.

Finally, we compare our model to SCAP-method which was introduced as an authorship identification method in Section 3.4 using same F_1-tests as in Figure 15a. We conduct this experiment to see if a method found during systematic literature review would get better or similar results than the Naïve Bayes model we used. As a recap, in SCAP-method one concatenates all documents per author to one large document, forms n-grams and keeps only the L most frequent n-grams to generate author profiles. A test document is then compared using this same technique to all existing author profiles. Comparison happens by taking intersection between n-gram set of a profile and document to get a non-normalized similarity value, and the decision is based on the largest intersection size. Selecting a small value of L allows to reduce the vocabulary size greatly, so we run tests with three candidate models with different profile sizes but using the same 10-grams as our model. The three different values of L we test are $10^2, 10^3$ and 10^4, which all are a lot smaller than our original vocabulary size $2.78 \cdot 10^5$. The results for F_1-scores using OHPE's data are seen in below figure.

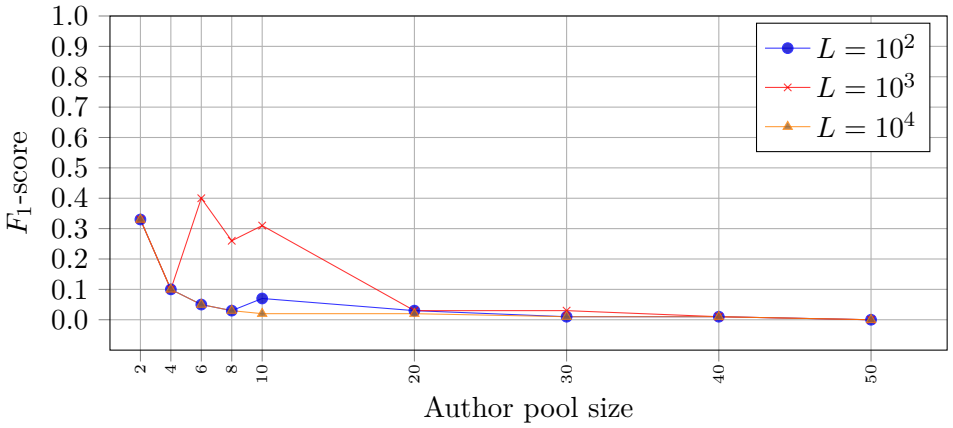


Figure 16: F_1-scores for three different profile lengths using 10-grams in OHPE. The only slight improvement compared to our model is when $L = 10^3$.

Visible in Figure 16, even the SCAP-method is not able to predict the author on a decent level of success, as the results are somewhat same when $L = 10^3$ as with our model. The fluctuations when the author pool size is under 10, are caused by random sampling and the existence of similar authors inside

the same sampled pool. When we looked the three most closest authors in every case, we saw the correct author was often in that set. However, as the author pool size was grown there were similar confusion happening as in Figure 14, where just few authors were labeled as authors of the most of the documents. Because SCAP wouldn't improve the results in OHPE, we decide to not to run the evaluation for OHPE's data.

We have now shown the results of our authorship identification model and seen how the problem with our data sets is too difficult for both Multinomial Naïve Bayes using TF-IDF weighted 10-grams and SCAP using 10-grams with varying profile sizes. In the next section we show the the final results for our plagiarism detection from the exam tasks of both OHPE and OHJA.

5.3 PLGDetect

Because the Multinomial Naïve Bayes and the SCAP evaluated poorly with our data sets, we decide not to use authorship identification for the final results as even reducing the amount of authors would diminish the possibility of finding any plagiarists as random sampling would leave some students out of the detection. This is a drawback for our approach and we discuss the implications at the discussion. However, our similarity detection model evaluated well and can be still used for exploring and detecting the possible plagiarists. What we can't do is to restrict efficiently the amount of false detections by using the authorship identification model.

Before we can discuss the final results, we must consider an issue with the retrieval rate of our similarity detection. Looking from the Table 23 and Table 24 there are around 500 total documents retrieved, which is too many documents for the human expert to go through in reasonable time. To overcome this issue the we select only a subset of the exam tasks reducing the amount of documents to 144. These are OHPE's third exam task (3.A) and all of the exam tasks of OHJA's (1.B, 2.B, 3.B). A brief description of each selected task is given below.

3.A (OHPE) Students were required to fill a method to find the most common number from the Java's ArrayList structure. The methods name, return value and parameters were given as a template.

1.B (OHJA) Students were required to make a text interface for adding books with name and year information. The outline of the text interface was given for the students. After the initial adding phase, added books were printed in wanted order.

2.B (OHJA) This task measured how well students are able to manipulate text data. The task required to have a small text interface to read a text file, censor every occurrence of a given word and write the results to a new

text file. This exercise had a hint, which recommended to use a specific Java class to read and write text files.

3.B (OHJA) Task required to create a text interface to emulate a simple storage management software. The actions that had to be implemented were adding, listing, searching, removing items and exiting the interface. A small piece of code was given as a hint for this exercise.

In all exam tasks, also the scoring and example output was given for the students, so that they could mimic the wanted functionality of these programs. The reason behind this was to guide the student into right direction and also to be able to automatically score the submissions.

To see the difference between these tasks, descriptive statistics about them is given in Table 29. It shows how OHPE differs from OHJA, as its task is quite constrained having only around 50 lines to get a correct answer. OHPE also creates a lot more clusters, as the similarities between OHJA's submissions are more varied.

Table 29: Results before the evaluation by the human expert. These results are produced by our similarity detection model which uses parameters $n = 3$ for the n-gram length and $\varepsilon = 0.2$ for the maximum allowed distance between the documents, which reflects that the documents have to score over 80% similarity in order to cluster them together.

Task	3.A	1.B	2.B	3.C
Number of submissions	227	200	198	197
Average line count	47	160	85	150
Documents retrieved	111	15	9	9
Clusters emerged	15	5	3	4

As the final result, we first show the pair-level detection results and then the more general result, which shows the precision with respect to documents considered containing plagiarism. For each of these tasks we inspect every cluster and the true and false positives in them, where the results are given by our human expert who has manually gone through detected documents. Results for each task is given in following figures, where we show the frequencies of retrieved pairs compared to true positives. Note that this format is more fine grained than what we have used before as earlier we have reported only the number of documents detected, and that we had to prune the first cluster of OHPE's third task, as it contained nearly 410 pairs. Pruning was done by keeping only the pairs where the cosine similarity was 1.0.

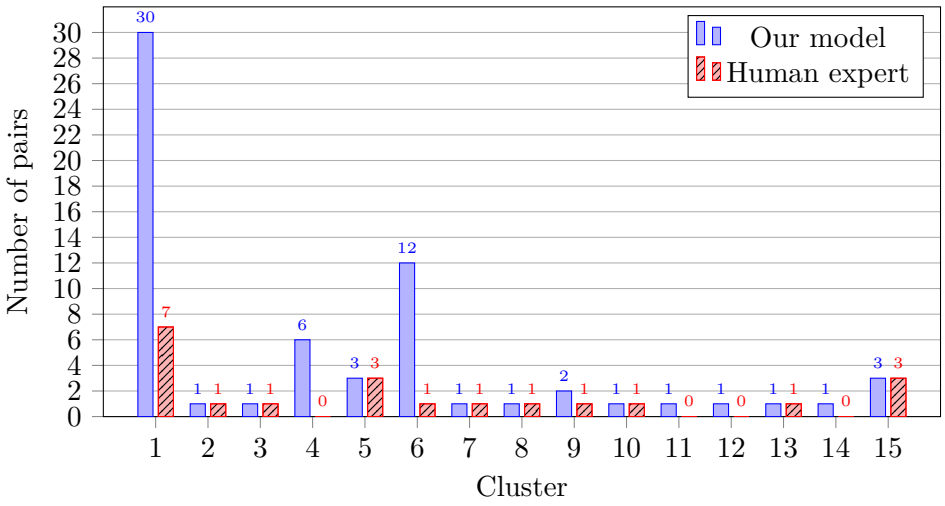


Figure 17: Detected and true pairs of 3.A OHPE. False positives in the first cluster were mostly correct submissions which were similar to model solution. Fourth cluster contained almost empty submissions and sixth cluster similarly wrong solutions with two highly suspicious authors.

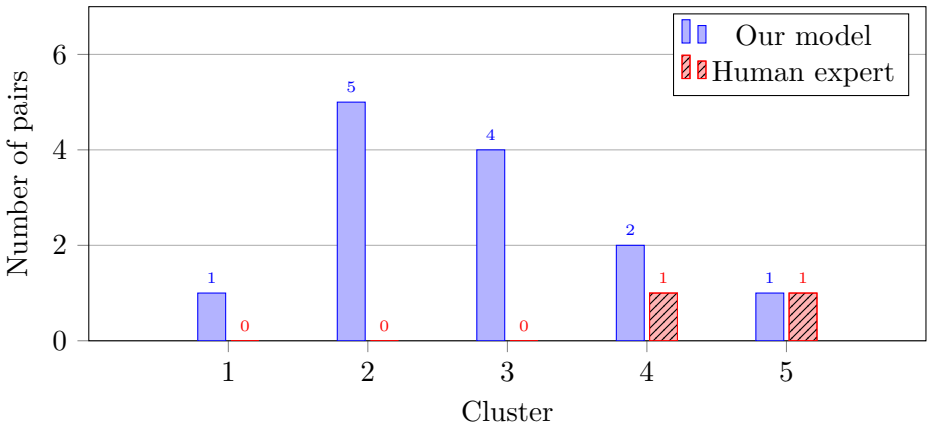


Figure 18: Detected and true pairs of 1.B OHJA. Most of the pairs were reported to be close to model solution without any signs of plagiarism. However, there were two pairs which were flagged for further attention.

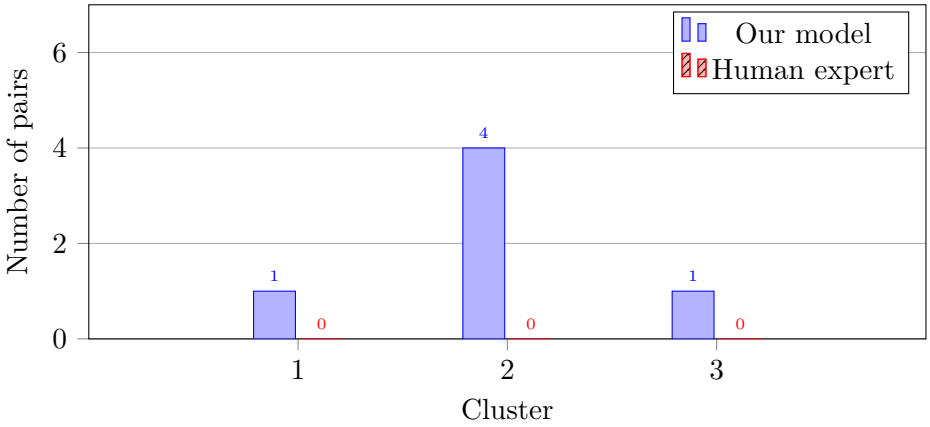


Figure 19: Detected and true pairs of 2.B OHJA. All of the detected pairs in this task were false positives. However, two non-paired authors were flagged for further attention.

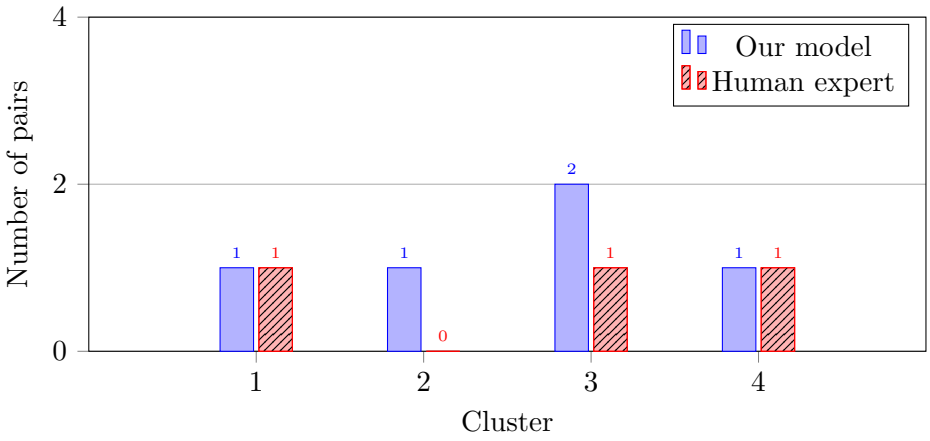


Figure 20: Detected and true pairs of 3.B OHJA. Three pairs were flagged for further attention, but as difficult cases.

In Figures 17, 18 and 20, we see that our approach is able to retrieve suspicious documents. As reported by the human expert, most of true positives contain direct copies and renaming of the variables. However, there exist false positives as seen in Figure 19 where most of these false positives are caused by natural similarity between the submissions. The human expert reported also that in most of the cases one can't say for sure that the document pair is plagiarism. Therefore, the reported pairs are flagged if they are considered as suspicious and would require further information *e.g.* other submissions done by the pair of authors. In Table 30, one sees the document level results of false and true positives with the level of precision for each task.

Table 30: Document-level results of our plagiarism detection. There are false positives introduced to our detection results.

Task	3.A	1.B	2.B	3.B
True Positives	30	4	0	6
False Positives	26	11	9	3
Precision	0.54	0.27	0.00	0.67

The low precision in Table 30 shows how our model fails to limit the amount of false positives, which can be mostly due to the fact that we had to use only the similarity detection part of our approach. As seen before, all of the submissions for OHPE and OHJA contain a high level of natural similarity, which introduces many false positives even with as high threshold as 80%. To help the work of our human expert, we had to prune the first cluster of OHPE's third task. In reality there would be near 400 detected document pairs, which are clearly all false positives due to the restricted solution space of the task.

After the human expert evaluated the detected documents, the plagiarists caught in 2016 were revealed. Our model was able to retrieve documents belonging for all of these authors in OHPE's third task.

6 Discussion

In following sections we first answer to our research questions, then discuss about limitations of our study and finally present ideas for the future work.

6.1 Revisiting research questions

We asked following three research questions related to a more bigger question *How plagiarism can be automatically detected?*.

Q1: *What kind of approaches exist to detect source code plagiarism?*

Q2: *What are the possible benefits of using code structure for plagiarism detection?*

Q3: *How can one obtain a model with a high detection accuracy?*

What kind of approaches exist to detect source code plagiarism? One must first define the term *plagiarism* to be able to detect it. We defined it as stealing and passing ideas as one's own which closely relates copying other students work *i.e.*, having similar documents. In Section 3, we made the first division first between a similarity detection and authorship identification methods based on the literature survey. Using similarity detection, one is able to form a value between two documents that tells how similar these

documents are, whereas using authorship identification, one can verify the author in theory. We found five subcategories from both methods: attribute counting, segment matching, n-grams, tree-based and hybrid. Approaches related to attribute counting forms frequencies or metrics from the source code, and some examples of metrics are number of lines, number of operators and number of tabs. Segment matching appears more frequently in similarity detection, where similarity value can be formed as easily as by calculating how many substrings two documents have in common. N-grams are easily retrievable despite the used programming language in documents, but the dimensionality of the data grows fast. Tree-based approaches transforms the source code into a tree format to get additional features. Lastly, hybrid approaches combine previously mentioned ideas to create more complex models.

All previously mentioned subcategories handle the source codes in slightly different ways and many of the studies we encountered during the literature survey had varying levels of granularity. They all however, share the same two principles: source code documents have to be normalized in order to remove unnecessary features and one should always use a human expert to evaluate the final results. Using a human, the automatic detection is not in practise about directly detecting plagiarism, but more about retrieving interesting documents which stands out from the mass. Retrieving documents based on a given criteria resembles closely the functionality of a search engine and therefore the same theoretical principles can be applied, which was seen in many studies during literature review.

The domain of plagiarism detection offers many use cases for machine learning and data exploration models because documents by themselves contain a lot of data hidden inside the actual written text. There is also a need for document suggestions so human experts do not have to go through everything manually. In our case, one batch of submissions contained around 200 files totaling 19 900 unique pairs that needs to be compared.

What are the possible benefits of using code structure for plagiarism detection? The structure offers more data than what can be extracted solely from written text. It allows to generalize the source code, so it is able to show more high-level information, which can doable by generating the abstract syntax tree. However, even more important benefits are the ability to reduce the noise, to generalize the documents and make the detection process more stable. For example, we use the structure for similarity detection to battle against obfuscation strategies like variable renaming or changing the order of expressions. By using the structure one is then able to 1) gain new features for the detection process, 2) restrict the variance of observed data, 3) make the detection resilient against common obfuscation strategies and 4) turn the focus more into the logic of the source code.

How can one obtain a model with high plagiarism detection accuracy? Our hypothesis was that a model using the results of similarity detection and authorship identification would be able to reduce the number of false positives *i.e.*, authors who are being wrongly accused. However, by using Naïve Bayes with n-grams, a probabilistic machine learning model, we couldn't get results decent enough to be even consider using it to verify the detected authors. The size of our author pool was during the evaluation around 50 students with nearly 100 documents for each as a training data, which is nearly double than other studies in Section 3. Also, most of those studies had a pool under 10 authors.

There are several possible reasons for why the proposed model did not perform as expected. First of all our vocabulary size was large and feature selection methods could have been utilized. Having a large vocabulary during authorship identification diminished the importance of possible features we hoped the model would capture. Second, both OHPE and OHJA guide the programming into a specific direction where students learn a unified style to program these tasks. Having a general style learned from the course material and exercises, makes it hard to detect authors when using only the submissions. Finally, the documents are very similar naturally as the exercises measure a specific knowledge per task, meaning that the solution space is limited.

Something can be still said about reducing false accusations. Our similarity detection uses a parameter to control the search range, which reflects the threshold that decided when two source code documents are too similar. With a low threshold, the amount of false accusations grows quickly and vice versa *i.e.*, in order to minimize false positives, one can use a high threshold value. However as we saw during the evaluation, choosing the best n-gram length and threshold value can be difficult, as these values are highly data dependent. For example the submissions to SOCO competition differs a lot from our data set and the only the training set of it is human evaluated. Despite these shortcomings, SOCO's training set is one of the few publicly available data sets, that offers a plagiarized documents which are all prelabeled and not synthetic. JPlag, on the other hand, is an existing tool for plagiarism detection and as we saw during our evaluation, also it generates false positives. The people using it must also come up with a way of deciding a working threshold value, which in many cases can be impossible due the lack of proper domain knowledge. Another thing that introduces false positives, are the submissions for the same exercise, where the solution space for is just too small to generate varying solutions.

6.2 Limitations of the study

Our study has limitations, mostly due to the assumptions we made. These are 1) in-class plagiarism, 2) exercise focus, 3) single author and 4) plagiarism

direction. As these assumptions restrict the plagiarism to exist inside a single course and exercise, we can't say it is very realistic situation as plagiarism can also be ongoing and exist between various courses by same plagiarists. For example a student can use answers from previous iterations of the course, can use material found from the Internet, and use a friend who has already completed the course. These cases are difficult to detect as most of the information can only be found outside the course, and thus many academic studies use same assumptions as we, only difference being detecting the plagiarism direction. There are many cases however, where the direction is important for example when one wants to find the original source *i.e.*, the sharer. Another major limitation is, as pointed out before, the poor performance of authorship identification. With our real life data sets, the amount of false positives in OHPE's third task was so much that we had to limit it by filtering out detected documents and our strategy was to gather only documents for the biggest cluster emerged in OHPE, where the similarity was maximum. Because we had to resort to this action in order limit the work of the human expert, there can exist many true positives our approach could not caught.

Another set of limitations are based on the choice of our approach, the most problematic being the choice of the training data set for similarity detection. We used the full training set of SOCO competition and its two test sets, one containing no plagiarism and the another 28 cases decided by a majority voting of other models, to tune our similarity detection model. Our results show that the choice for hyperparameters using another non-course-based data set is not easy to make as there exists a lack of proper domain knowledge. For example 0.4 as the ε-range could work well for SOCO, but produce large amount of false positives in OHPE. Other similarity detection related limitation is that we use a fixed similarity measure and did not try out other measures like for example the Euclidean distance. On the other hand, limitations related to authorship identification include two major topics: we did not restrict the vocabulary size using feature selection strategies and we used a relatively simple probabilistic model. The vocabulary size for us was 278 000 unique features using character-level n-grams, which could have been minimized by applying statistical methods *e.g.* chi-squared test, to select only a set of most important features for classification. When speaking of the model we used, it is a simple probabilistic model which is often used as a baseline model and the assumption about non-existing feature correlation is not true in real life, because source code is written by following the grammar rules and various stylistic preferences *e.g.* spacing.

6.3 Future work

For the future, there are many approaches which could be taken. As an example, one could use the process contrast to the final product, to get

more in-depth analysis of how the plagiarism occurs. One could also track the style of an author inside a time frame, to see if it remains same over time or changes. Different normalization methods and tokenization for both similarity detection and authorship identification should be used, but at the moment it is difficult as every source code data set differs from each other thus no general guidelines can be given. Another important aspect is also how the source code files are represented and handled during the detection. We used the tree-format and project-level detection, but as well found from other studies varying representations and granularities. For example to minimize the amount of false positives during similarity detection, one could first decompose a source code document into lines or other more coarse units and then run the detection so that only parts of the documents would be under the detection. But, even this approach wouldn't help if the tasks are too restricted by nature. Our results also revealed clusters which could be studies more and generally more information about possible plagiarism can be used to enhance the detection, like are same students regularly inside same clusters and how correct non-plagiarized solutions could be filtered out from the process. These emerging false positives are such a large issue, that more effort should be put into tackling with them and looking if it's possible at all to find plagiarism inside a single course with short exercises.

We suggest that the concept of source code plagiarism should be more researched, which means creating more studies answering to questions like 1) *What are the other topics related to source code plagiarism detection than similarity detection and authorship identification?*, 2) *What is the amount of data one needs to have for a precise detection?*, 3) *What kind of authorship identification is possible to apply in a course with hundreds of students?*, 4) *How much and what type of normalization can reduce the noise enough to have a precise model?* and 5) *What types of data can be collected from a student, so she can be accused from the act of plagiarism?*. The field of source code plagiarism detection offers many use cases for techniques like large-scale machine learning, statistical modeling, data mining, data visualization and artificial neural network, but the problem lies more in defining the actual plagiarism. Even our human expert told after the evaluation, that it was hard for some tasks to tell if it was plagiarism or just coincidental similarity, which raises the question that how much evidence the plagiarism detection process should gather and from where. We believe that using one course as a detection target makes the overall detection difficult and using only one model, makes accurate detection process nearly impossible. Thus more effort should but into research of multi-model approaches, where similarity detection plays a small part of much bigger detection process. There should also exist more publicly available datasets containing proven cases of source code plagiarism, which could be used for further academic research. In this way, researchers could validate between different models.

7 Conclusion

The problem of software plagiarism detection is difficult as the term itself is very vague. In its simplest form, plagiarism can mean direct copying and in its most complex forms it is undetectable because the plagiarist has obfuscated the document. To detect plagiarism in large amounts of source codes, as seen in many programming courses, one must resort to tools that can help humans with the overwhelming process of comparing large amount of documents together.

In this thesis, we (1) conducted a systematic literature review on the topic of *How source code plagiarism can be detected in a set of documents*, (2) proposed a novel categorization between various plagiarism detection models, (3) proposed an n-gram-based approach to detect software plagiarism using two real-life data sets, and (4) compared it to an existing plagiarism detection tool.

Our proposed approach consists of two parts: similarity detection and authorship identification. The similarity detection model, which measures the similarity between documents, works fairly well when our results were verified by a human expert. We selected this model by running multiple tests ranging from numerical evaluation to inspecting sizes of emerged similarity groups, and found that false positives were often detected. Authorship identification was tested using various training data and author pool sizes, but we were unable to apply it in a useful way for our problem. The setback we faced with the authorship identification was, in our opinion, due to the large size of the formed vocabulary which minimized the importance of individual words.

Our preliminary results show that to some extent our approach is able to detect plagiarism, as it was able to find all proven cases of plagiarism in a predefined set of exams. However, the problem is not trivial because the complexity of a human written text *i.e.*, source code, provides interesting challenges, as there are many ways to express the same logical meaning. This means that the used vocabulary can grow very large for each student and a fine balance must be found to decide which words or word pairs are important for the detection task. Another challenge related to source code plagiarism detection is to both acquire written samples for each student and to decide which amount of samples is enough.

Based on the results of this thesis, we can say that decisions based on single detection results are problematic, because the motivations and actions behind a real plagiarism are multidimensional and proving plagiarism is difficult. To be able to have a good detection approach, one must have additional information about the possible plagiarism case, *e.g.* who the sharer is and who has copied the document, if the creation process can reveal more insight of the cheater, and how the suspects can be confirmed. Still, computer-based tools can help with the overall process and reduce manual work tremendously.

References

[1] Acampora, Giovanni and Cosma, Georgina: *A fuzzy-based approach to programming language independent source-code plagiarism detection.* In *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–8, Aug 2015.

[2] Ali, Asim M. El Tahir, Abdulla, Hussam M. Dahwa, and Snásel, Václav: *Overview and comparison of plagiarism detection tools.* In *DATESO*, pages 161–172, 2011.

[3] Alsulami, Bander, Dauber, Edwin, Harang, Richard, Mancoridis, Spiros, and Greenstadt, Rachel: *Source code authorship attribution using long short-term memory based networks.* In *Computer Security – ESORICS 2017*, pages 65–82. Springer International Publishing, 2017.

[4] Arabyarmohamady, S., Moradi, Hadi, and Asadpour, Masoud: *A coding style-based plagiarism detection.* In *Proceedings of 2012 International Conference on Interactive Mobile and Computer Aided Learning (IMCL)*, pages 180–186, Nov 2012.

[5] Bandara, Upul and Wijayarathna, Gamini: *Deep neural networks for source code author identification.* In *Proceedings, Part II, of the 20th International Conference on Neural Information Processing - Volume 8227*, ICONIP 2013, pages 368–375. Springer-Verlag New York, Inc., 2013.

[6] Bandara, Upul and Wijayarathna, Gamini: *Source code author identification with unsupervised feature learning.* Pattern Recogn. Lett., 34(3):330–334, February 2013, ISSN 0167-8655.

[7] Breslow, Lori, Pritchard, David, DeBoer, Jennifer, Stump, Glenda, D. Ho, Andrew, and Seaton, Daniel: *Studying learning in the worldwide classroom: Research into edX's first MOOC.* Research & Practice and Assessment, June 2013.

[8] Brixtel, Romain, Fontaine, Mathieu, Lesner, Boris, Bazin, Cyril, and Robbes, Romain: *Language-independent clone detection applied to plagiarism detection.* In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 77–86, Sept 2010.

[9] Burrows, Steven and Tahaghoghi, Seyed MM: *Source code authorship attribution using n-grams.* In *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*, pages 32–39. Citeseer, 2007.

[10] Burrows, Steven, Uitdenbogerd, Alexandra L., and Turpin, Andrew: *Application of information retrieval techniques for source code authorship attribution.* In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*, DASFAA '09, pages 699–713, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Cosma, Georgina and Joy, Mike: *Towards a definition of source-code plagiarism.* IEEE Transactions on Education, 51(2):195–200, 2008.

[12] Cosma, Georgina and Joy, Mike: *An approach to source-code plagiarism detection and investigation using latent semantic analysis.* IEEE Transactions on Computers, 61(3):379–394, March 2012.

[13] Cruz, Aarón Ramírez de la, Rosa, Gabriela Ramírez de la, Sánchez Sánchez, Christian, and Jiménez Salazar, Héctor: *On the importance of lexicon, structure and style for identifying source code plagiarism.* In *Proceedings of the Forum for Information Retrieval Evaluation*, FIRE '14, pages 31–38. ACM, 2015.

[14] Cruz, Aarón Ramírez de la, Rosa, Gabriela Ramírez de la, Sánchez Sánchez, Christian, Jiménez Salazar, Héctor, Rodríguez-Lucatero, Carlos, and Luna-Ramírez, Wulfrano Arturo: *High level features for detecting source code plagiarism across programming languages.* In *FIRE Workshops*, pages 10–14, 2015.

[15] Dick, Martin, Sheard, Judy, Bareiss, Cathy, Carter, Janet, Joyce, Donald, Harding, Trevor, and Laxer, Cary: *Addressing student cheating: Definitions and solutions.* SIGCSE Bull., 35(2):172–184, June 2002, ISSN 0097-8418. `http://doi.acm.org/10.1145/782941.783000`.

[16] Ding, Haibiao and Samadzadeh, Mansur H.: *Extraction of Java program fingerprints for software authorship identification.* J. Syst. Softw., 72(1):49–57, June 2004, ISSN 0164-1212.

[17] Ester, Martin, Kriegel, Hans Peter, Sander, Jörg, and Xu, Xiaowei: *A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise.* In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996. `http://dl.acm.org/citation.cfm?id=3001460.3001507`.

[18] Faidhi, J. A. W. and Robinson, S. K.: *An empirical approach for detecting program similarity and plagiarism within a university programming environment.* Comput. Educ., 11(1):11–19, January 1987, ISSN 0360-1315. `http://dx.doi.org/10.1016/0360-1315(87)90042-X`.

[19] Flores, Enrique, Barrón-Cedeño, Alberto, Moreno, Lidia, and Rosso, Paolo: *Uncovering source code reuse in large-scale academic environments.* Computer Applications in Engineering Education, 23(3):383–390, 2015.

[20] Frantzeskou, Georgia, MacDonell, Stephen, Stamatatos, Efstathios, Georgiou, Stelios, and Gritzalis, Stefanos: *The significance of user-defined identifiers in Java source code authorship identification.* International Journal of Computer Systems Science and Engineering, 26:139–148, March 2011.

[21] Frantzeskou, Georgia, MacDonell, Stephen, Stamatatos, Efstathios, and Gritzalis, Stefanos: *Examining the significance of high-level programming features in source code author classification.* J. Syst. Softw., 81(3):447–460, 2008, ISSN 0164-1212.

[22] Fu, Deqiang, Xu, Yanyan, Yu, Haoran, and Yang, Boyang: *WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection.* Scientific Programming, 2017:7809047:1–7809047:8, 2017.

[23] Ganguly, Debasis and Jones, Gareth J. F.: *An information retrieval approach for source code plagiarism detection.* In *Proceedings of the Forum for Information Retrieval Evaluation*, FIRE '14, pages 39–42. ACM, 2015.

[24] Ganguly, Debasis, Jones, Gareth J. F., Cruz, Aarón Ramírez de la, Rosa, Gabriela Ramírez de la, and Villatoro Tello, Esaú: *Retrieving and classifying instances of source code plagiarism.* Information Retrieval Journal, Sep 2017.

[25] Golub, G. H. and Reinsch, C.: *Singular value decomposition and least squares solutions.* Numer. Math., 14(5):403–420, April 1970, ISSN 0029-599X.

[26] Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome: *The elements of statistical learning: data mining, inference and prediction.* Springer, 2nd edition, 2009.

[27] Heblikar, Saimadhav, Sharma, Poorva, Munnangi, Manogna, and Bankapur, Channa: *Normalization based stop-word approach to source code plagiarism detection.* In *FIRE Workshops*, 2015.

[28] Hellas, Arto, Leinonen, Juho, and Ihantola, Petri: *Plagiarism in take-home exams: Help-seeking, collaboration, and systematic cheating.* In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 238–243, New York, NY, USA, 2017. ACM, ISBN 978-1-4503-4704-4. `http://doi.acm.org/10.1145/3059009.3059065`.

[29] Johnson, Stephen C: *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[30] Joy, Mike and Luck, Michael: *Plagiarism in programming assignments.* IEEE Transactions on Education, 42(2):129–133, May 1999, ISSN 0018-9359.

[31] Kibriya, Ashraf M., Frank, Eibe, Pfahringer, Bernhard, and Holmes, Geoffrey: *Multinomial Naive Bayes for text categorization revisited.* In *Proceedings of the 17th Australian Joint Conference on Advances in Artificial Intelligence*, AI'04, pages 488–499, Berlin, Heidelberg, 2004. Springer-Verlag, ISBN 3-540-24059-4, 978-3-540-24059-4.

[32] Kothari, Ja., Shevertalov, Maxim, Stehle, Edward, and Mancoridis, Spiros: *A probabilistic approach to source code authorship identification.* In *Information Technology, 2007. ITNG '07. Fourth International Conference on*, pages 243–248, April 2007.

[33] Krsul, Ivan and Spafford, Eugene H.: *Authorship analysis: identifying the author of a program.* Computers & Security, 16(3):233 – 257, 1997, ISSN 0167-4048.

[34] Lange, Robert Charles and Mancoridis, Spiros: *Using code metric histograms and genetic algorithms to perform author identification for software forensics.* In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 2082–2089. ACM, 2007.

[35] Levenshtein, V. I.: *Binary Codes Capable of Correcting Deletions, Insertions and Reversals.* Soviet Physics Doklady, 10:707, February 1966.

[36] Manning, Christopher D., Raghavan, Prabhakar, and Schütze, Hinrich: *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008, ISBN 0521865719, 9780521865715.

[37] Mccallum, Andrew and Nigam, Kamal: *A comparison of event models for Naive Bayes text classification.* In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48, May 2001.

[38] Moussiades, Lefteris and Vakali, Athena: *PDetect: A clustering approach for detecting plagiarism in source code datasets.* The Computer Journal, 48(6):651–661, Nov 2005.

[39] Muddu, Basavaraju, Asadullah, Allahbaksh, and Bhat, Vasudev: *CPDP: A robust technique for plagiarism detection in source code.* In *2013 7th International Workshop on Software Clones (IWSC)*, pages 39–45, May 2013.

[40] Ng, Sin Chun, Lui, Andrew Kwok Fai, and Wong, Lai Shan: *Tree-based comparison for plagiarism detection and automatic marking of programming assignments.* In *Engaging Learners Through Emerging Technologies*, pages 165–179, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[41] Novak, Matija: *Review of source-code plagiarism detection in academia.* In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 796–801, May 2016.

[42] Ohmann, Tony and Rahal, Imad: *Efficient clustering-based source code plagiarism detection using PIY.* Knowledge and Information Systems, 43(2):445–472, May 2015.

[43] Okoli, Chitu and Schabram, Kira: *A guide to conducting a systematic literature review of information systems research.* SSRN Electronic Journal, 10, May 2010.

[44] Pieterse, Vreda: *Decoding code plagiarism.* In *Proceedings of the 44th Annual Conference of the Southern African Computer Lecturers' Association (SACLA)*, 2014.

[45] Prechelt, Lutz, Malpohl, Guido, and Philippsen, Michael: *Finding plagiarisms among a set of programs with JPlag.* J. UCS, 8(11):1016, 2002.

[46] Roy, Chanchal K., Cordy, James R., and Koschke, Rainer: *Comparison and evaluation of code clone detection techniques and tools: A qualitative approach.* Sci. Comput. Program., 74(7):470–495, May 2009, ISSN 0167-6423. `http://dx.doi.org/10.1016/j.scico.2009.02.007`.

[47] Sáez, Enrique Flores, Rosso, Paolo, Boronat, Lidia Ana Moreno, and Villatoro-Tello, Esaú: *PAN@ FIRE: Overview of SOCO track on the detection of source code re-use.* ACM, 2014.

[48] Schubert, Erich, Sander, Jörg, Ester, Martin, Kriegel, Hans Peter, and Xu, Xiaowei: *DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN.* ACM Trans. Database Syst., 42(3):19:1–19:21, July 2017, ISSN 0362-5915. `http://doi.acm.org/10.1145/3068335`.

[49] Smith, T.F. and Waterman, M.S.: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147(1):195 – 197, 1981, ISSN 0022-2836.

[50] Sokolova, Marina and Lapalme, Guy: *A systematic analysis of performance measures for classification tasks.* Information Processing & Management, 45(4):427 – 437, 2009, ISSN 0306-4573.

[51] Son, Jeong Woo, Noh, Tae Gil, Song, Hyun Je, and Park, Seong Bae: *An application for plagiarized source code detection based on a parse tree kernel.* Eng. Appl. Artif. Intell., 26(8):1911–1918, September 2013, ISSN 0952-1976.

[52] Stamatatos, Efstathios: *A survey of modern authorship attribution methods.* J. Am. Soc. Inf. Sci. Technol., 60(3):538–556, March 2009, ISSN 1532-2882.

[53] Tennyson, Matthew F. and Mitropoulos, Francisco J.: *A Bayesian ensemble classifier for source code authorship attribution.* In Traina, Agma Juci Machado, Traina, Caetano, and Cordeiro, Robson Leonardo Ferreira (editors): *Similarity Search and Applications*, pages 265–276. Springer International Publishing, 2014.

[54] Tennyson, Matthew F. and Mitropoulos, Francisco J.: *Choosing a profile length in the SCAP method of source code authorship attribution.* In *IEEE SOUTHEASTCON 2014*, pages 1–6, March 2014.

[55] Verco, Kristina L. and Wise, Michael J.: *Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems.* In *Proceedings of the 1st Australasian Conference on Computer Science Education*, ACSE '96, pages 81–88, New York, NY, USA, 1996. ACM, ISBN 0-89791-845-2. http://doi.acm.org/10.1145/369585.369598.

[56] Vihavainen, Arto, Luukkainen, Matti, and Kurhila, Jaakko: *Multi-faceted support for MOOC in programming.* In *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE '12, pages 171–176, New York, NY, USA, 2012. ACM, ISBN 978-1-4503-1464-0. http://doi.acm.org/10.1145/2380552.2380603.

[57] Vihavainen, Arto, Paksula, Matti, and Luukkainen, Matti: *Extreme apprenticeship method in teaching programming for beginners.* In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 93–98, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0500-6. http://doi.acm.org/10.1145/1953163.1953196.

[58] Vihavainen, Arto, Vikberg, Thomas, Luukkainen, Matti, and Pärtel, Martin: *Scaffolding students' learning using Test My Code.* In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 117–122,

New York, NY, USA, 2013. ACM, ISBN 978-1-4503-2078-8. http://doi.acm.org/10.1145/2462476.2462501.

[59] Wise, Michael: *String similarity via Greedy String Tiling and running Karp-Rabin matching*. Online Preprint, January 1993.

[60] Wisse, Wilco and Veenman, Cor: *Scripting DNA: Identifying the JavaScript programmer*. Digital Investigation, 15:61 – 71, 2015. Special Issue: Big Data and Intelligent Data Analysis.

[61] Xiong, Hao, Yan, Haihua, Li, Zhoujun, and Li, Hu: *Buaa antiplagiarism: A system to detect plagiarism for C source code*. In *2009 International Conference on Computational Intelligence and Software Engineering*, pages 1–5, Dec 2009.

[62] Zhang, Chunxia, Wang, Sen, Wu, Jiayu, and Niu, Zhendong: *Authorship identification of source codes*. In *Web and Big Data*, pages 282–296. Springer International Publishing, 2017.

[63] Zhang, Harry: *The optimality of Naïve Bayes*. In *In FLAIRS2004 conference*, 2004.

[64] Zhang, Li ping and Liu, Dong sheng: *AST-based multi-language plagiarism detection method*. In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pages 738–742, May 2013.

A Sample programs

These three functionally similar source codes belong to three imaginary authors A, B and C. They are used throughout the study as examples. The task for all is to create a program that calculates mean between three numbers: 5, 10, 2.

Listing 1: Java example belonging to author A

```
public class A{

     public static void main(String[] args){
        int a = 5;
        int b = 10;
        int c = 2;
        double d = (a + b + c)/(double)3;
        System.out.println(d);
     }
}
```

Listing 2: Java example belonging to author B

```
public class B{

     public static void main(String[] b){
        int sum = 5 + 10 + 2;
        double res = sum / 3.0;
        System.out.println(res);
     }
}
```

Listing 3: Java example belonging to author C

```
public class C{
     public static void main(String[] b){
        System.out.println((5 + 10 + 2)/3.0);
     }
}
```

B Token list

Table 31: Token list for Java.

	Token	Equivalency	Example
1	IMPORT	Import declaration	`import java.awt.*;`
2	PACKAGE	Package declaration	`package foo;`
3	VARDEF	Variable declaration	`int a;`
4	CLASS{	Enter class declaration	`public class A{`
5	CATCH{	Enter catch clause	`try {catch (...){} }`
6	INCLASS{	Statement inside a class	-
7	ENUM{	Enter enum declaration	`public enum Day {`
8	APPLY	Method call, Explicit constructor invocation, Generic invocation	`System.out.print(...);`
9	NEWCLASS	Create object	`new A(...);`
10	NEWARRAY	Create array object	`new int[5];`
11	TRY{	Enter try declaration	`try {`
12	INTERF{	Enter interface declaration	`interface Foo {`
13	METHOD{	Enter method declaration	`void foo(int a) {`
14	VOID	Void method	`void main(String[] args)`
15	CASE	Case in switch statement	`case MONDAY:`
16	CONSTR{	Enter constructor declaration	`public A(int a, int b) {`
17	ARRINIT{	Enter array initialization	`new int[] {1, 2};`
18	ASSIGN	Variable assignment	`a += 5;`
19	COND	Conditional expression	`(a > b) ? a : b;`
20	LOOP{	Enter for, while, do statement	`for(...) {`
21	IF{	Enter if clause	`if(...) {`
22	THROW	Throw statement	`throw new Exception();`
23	BREAK	Break statement in loop	`break;`
24	CONTINUE	Continue statement in loop	`continue;`
25	RETURN	Return statement	`return a + b;`
26	SWITCH{	Enter switch statement	`switch(...) {`

Table 31 shows the token list used to transform a parse tree into a continuous string of tokens. Every token with ending bracket also has a reserved token when exiting the statement.