2013

Efficient Clustering-based Plagiarism Detection using IPPDC

Anthony Ohmann
College of Saint Benedict/Saint John's University

Efficient Clustering-based Plagiarism Detection using IPPDC

An Honors Thesis

College of Saint Benedict and Saint John's University

In Partial Fulfillment of the Requirements
for All College Honors and Distinction in the Department of Computer Science

Anthony Ohmann

Under Dr. Imad Rahal
April 2013

PROJECT TITLE: *Efficient Clustering-based Plagiarism Detection using IPPDC*

Approved by:

____

Dr. Imad Rahal, Associate Professor, Department of Computer Science

____

Dr. James Schnepf, Associate Professor and Chair, Department of Computer Science

____

Dr. Lynn Ziegler, Professor, Department of Computer Science

____

Dr. Tony Cunningham, Director, Honors Thesis Program

ABSTRACT

Vast amounts of information available online make plagiarism increasingly easy to commit, and this is particularly true of source code. The traditional approach of detecting copied work in a course setting is manual inspection. This is not only tedious but also typically misses code plagiarized from outside sources or even from an earlier offering of the course. Systems to automatically detect source code plagiarism exist but tend to focus on small submission sets. One such system that has become a standard is MOSS (measure of software similarity) [15].

In this work, we present a system called IPPDC (Intelligent Parallel Plagiarism Detection using Clustering) which is empirically shown to outperform MOSS in detection accuracy. By utilizing parallel processing and data clustering, our system is also capable of maintaining detection accuracy and reasonable runtimes even when using extremely large data repositories.

1. INTRODUCTION

In the modern day, plagiarism has become a serious problem demanding the attention of the academic community at large. The problem, or plague as described in the literature, is very common in written works especially among university students. This is due to various reasons such as time pressure, misunderstanding of what constitutes plagiarism, and the wealth of digital resources available on the Internet which make copy-and-paste activities almost natural. The latter is particularly true in the realm of computer science and source code.

Source code plagiarism can be defined as the act of copying code written in any programming language from someone else and submitting it for evaluation as if it was one's own work with no, minor, or even major modifications aimed at concealing plagiarism. However, this definition can easily be extended beyond the academic context to handle similar cases in industry where it has become commonplace to illegally include partially or fully plagiarized code snippets into commercial software products with the objective of reaping financial or other benefits.

A common approach to detecting source code plagiarism is via manual inspections. However, in programming courses, enrollments can be very high—especially in introductory courses—and the number of submissions is often beyond what a grader can reasonably be expected to inspect in a limited amount of time. Furthermore, plagiarism may occur across different sections of a course which may have separate graders—particularly with sections from previous years—making it unlikely that such plagiarism would be detected. Thus, automated inspections are often the only reasonable option.

Detection of source code plagiarism is in many ways similar to the detection of plagiarism in natural language essays or term papers, yet it can be a much more challenging problem. This is largely due to the fact that plagiarism in source code can be algorithmic or logic-based rather than solely syntactic, so one is typically searching for plagiarized logic rather than copied syntax. Furthermore, most programming languages are rich in syntax giving the plagiarist various syntactic ways to represent equivalent logic which in turn aids in concealing plagiarism. While automated plagiarism detection systems scalable to massive submission corpora exist for natural language essays such as *turnitin.com*, there has been a very limited amount of research work done in automating the large-scale detection of source code plagiarism. In addition, such efforts have focused solely on intra-corpal plagiarism detection where submissions are compared to one another within the submitted set.

In this work, we propose a scalable system called *I*ntelligent *P*arallel *P*lagiarism *D*etection using *C*lustering (or *IPPDC*) capable of performing inter-corpal plagiarism detection (in addition to intra-corpal) over a massive data repository of previous submissions where new submissions are compared against submissions currently stored in the data repository. This is reminiscent of the status quo in natural language plagiarism detection. Maintaining such a repository is a key factor in successfully source code detecting plagiarism, as our experience has shown that when plagiarism occurs, the original source is more often a submission from a previous course offering rather than the current one.

Even though the ideas proposed in this work are programming-language as well as context independent, we focus on a case study pertaining to an introductory-level programming course offered at our institution. The main rational for focusing our work on this particular course is that we have been able to collect a large repository of approximately 580 source code submissions which currently forms the backend of our proposed system. These submissions are the result of the course's required Visual Basic project in which students independently select a project topic and apply the skills and techniques learned in this course to build a working prototype. Due to the "open" nature of the project, one would expect projects to be significantly different from one another. This is similar in nature to term papers in which students propose a thesis and write a paper using the material learned in the course to support their thesis claims. Before we discuss more details, it is important to note that our system could easily be adapted to other programming languages by using a token set appropriate for the language of choice (Section 3.2).

A Visual Basic project is typically made up of a number of units called forms which resemble classes in object-oriented languages such as Java and C++. Each form in turn contains a number of procedures typically referred to as subroutines. Thus, there are three levels of granularity which seem natural for us to pursue in our plagiarism detection: *project-level*, *form-*

level, and *subroutine-level*.

The coarsest level of granularity is *project-level* plagiarism detection in which case a submission is identified as a likely plagiarism if we identify another submission in the repository such that the number of similar forms between the two submissions is above a certain threshold. Given the "open" nature of the project in the course, this approach definitely has the potential of minimizing the number of false positives, since it is unlikely to find two un-plagiarized submissions that yet share a significant number of similar forms. However, this approach can be easily misled by small amounts of plagiarism distributed among the forms within a project, so the number of false negatives (missed detections) would likely be higher than they would be for other granularities.

At the other granularity extreme is *subroutine-level* plagiarism detection. This approach identifies likely plagiarism between subroutines within different forms. Since this approach compares smaller blocks of code—i.e., subroutines each of which may be anywhere from five to around one hundred lines of code in our repository—it is highly likely that its results will produce the least false negatives. However, because each submission contains several forms that are each made up of multiple subroutines, the number of comparisons mandated at this level of granularity greatly increases. It is also important to note that in situations where submissions tend to contain some type of boiler-plate code—such as file reading and array searching or sorting—results are likely to suffer from a large number of false positives.

The level chosen for the approach described in this work is *form-level* plagiarism detection. *Form-level* detection compares each form of a submission independently and determines likely plagiarism based on the amount of shared code between two forms. This approach attempts to maximize the benefits of the other granularity levels while minimizing the limitations. For example, because *form-level* granularity compares smaller blocks of code than *project-level*, it is likely to produce fewer false negatives, and because it compares larger blocks of code than *sub-routine* level, it will require fewer comparisons and may produce fewer false positives.

After a brief literature review, the basic theory behind our plagiarism detection approach will be presented followed by an analysis of detection accuracy. We will then explore IPPDC's scalability by outlining implementations of the parallel and clustering algorithms along with each method's impact on performance and accuracy, respectively. Finally, we will present performance results of our complete system using both parallel pairwise comparisons and data clustering, and we will discuss the advantages of using the parallel strategy alone, the clustering strategy alone, and the combined strategy.

2. LITERATURE REVIEW

While there are various systems available for source code plagiarism detection, three systems: YAP3 [19], JPLAG [12], and MOSS [15], are often considered the standard for source code plagiarism detection with the latter being the most prominent. The goal of these three plagiarism detection systems, along with most others, is to be robust against as many plagiarism-disguising techniques as possible.

If plagiarism consisted only of duplicating unmodified code, detection would be trivial. Whale in [17] has identified twelve techniques for disguising similar source code that should be considered when judging the merits of a source code plagiarism detection system. They are as follows:

1. *Changing comments or formatting*

2. *Changing identifiers*
3. *Changing the order of operands in expressions*
4. *Changing data types* [float for integer; data structures]
5. *Replacing expressions by equivalents* ["while condition is true…" for "while not condition is false …"]
6. *Adding redundant statements or variables* [unnecessary initializations; additional output statements]
7. *Changing the order of independent statements* [rearranging clauses and reordering independent goals]
8. *Changing the structure of iteration statements* ["repeat" for "while"; "while" for "for"]
9. *Changing the structure of selection statements* [linearizing nested "if"s; using "if"s for "case"]
10. *Replacing procedure calls by the procedure body*
11. *Introducing non-structured statements*
12. *Combining original and copied program fragments*

The plagiarism disguises shown above can then be grouped into the following five categories based on typical approaches used to defend against them with the parenthesized numbers referring to the above list of plagiarism disguising techniques. Because copying an entire program is fairly straightforward to detect, technique 12 will often be used in conjunction with any of the eleven others. Thus, it is not explicitly included in any of the following disguise types.

Type I:	Changing comments or formatting (1)
Type II:	Changing identifiers or data types (2,4)
Type III:	Replacing expressions with equivalents, changing the structure of iteration or selection statements, replacing procedure calls with the procedure body (5,8,9,10)
Type IV:	Changing the order of operands in expressions or of independent statements (3,7)
Type V:	Adding redundant statements or variables, introducing non-structured statements (6,11)

Figure 2.1: Plagiarism disguise types with corresponding technique numbers from Whale.

Most plagiarism detection systems defend against Type I using cleaning (Section 3.1) and

Type II using tokenization (Section 3.2) [15]. As we later explain, our tokenization step also defends against some Type III plagiarism. In general, detection of types III, IV, and V is primarily where plagiarism detection systems differ and strive to improve upon current standards.

Both IPPDC and MOSS are based on representing code submissions as *k*-grams, so we will briefly present this central concept first before introducing other popular approaches in the literature.

2.1 *k*-grams

The idea of the *k*-grams technique is to create a list of all substrings of length k that appear in a particular document along with their frequencies. A *k*-gram is a contiguous substring of length k; thus, the number of *k*-grams generated by a string of length N is exactly $N-(k-1)$. For example, in Figure 2.2, $N=9$ and $k=5$, so there are $9-(5-1)=5$ generated *k*-grams.

```
abcabcabc
```
(a) A string of length 9

```
abcab bcabc cabca abcab bcabc
```
(b) The 5-grams generated from the string

```
abcab   2
bcabc   2
cabca   1
```
(c) The unique 5-grams with frequencies

Figure 2.2: The *k*-grams generation process.

The list of a document's *k*-grams and their frequencies can be treated as a vector where the *k*-gram is the position and the frequency of that *k*-gram is the value at that position. Using the example in Figure 2.2, the associated vector would be (2,2,1), although which *k*-gram corresponds to which position would also be stored. Once the *k*-gram frequency information has been compiled for all submissions being compared, similarity or distance measures can then be applied to the vectors representing each pair of documents (Section 3.4).

2.2 Winnowing

MOSS, which stands for Measure of Software Similarity [15], is a plagiarism detection system created and maintained by Alex Aiken and is publicly accessible at the following URL: *http://theory.stanford.edu/~aiken/moss/*. MOSS utilizes a pairwise comparison algorithm based on the winnowing algorithm discussed in [15] to improve on a string matching technique known as k-gram fingerprinting [5]. K-gram fingerprinting uses hashing to select a subset of the k-grams as the fingerprint for a submission—a document. Most systems that use k-gram fingerprinting choose only a subset of the hashes to use as the fingerprint for a document, since there are N-(k-1) k-grams in a string of length N. The authors of the winnowing algorithm assert that along with types I and II, their algorithm is robust against types IV and V as well. Although there is no explicit indication that MOSS is robust against type III, with the correct token set (Section 3.2) almost any algorithm can defend against this disguise type.

A common approach to choosing hash sets is to select all hashes that are $0 \bmod p$ for some given integer p as used in [5]. For example, if $p=4$, then every fourth hash would be used. The problem with this approach is that it does not guarantee that all matches of a certain length will be found. To overcome this limitation the authors of [15] designed the winnowing algorithm which selects the minimum hash value from a window of hashes. The size of the window of hashes is defined as $w = t - n + 1$, where t is the guarantee threshold (if a string is at least of length t then it will be matched) and n is the noise threshold (strings of length n or less are guaranteed not to be matched). By choosing one hash from every window of length w the authors of [15] show that all matches between documents of length t are guaranteed to be found. The existence of this guarantee is the winnowing algorithm's major improvement over previous *k*-gram fingerprinting techniques.

2.3 Running-Karp-Rabin Greedy-String-Tiling

The most popular alternative to the winnowing algorithm is the Running-Karp-Rabin Greedy-String-Tiling algorithm (RKR-GST), which is said to trade some of the efficiency of winnowing for increased accuracy [12]. Both JPLAG [12] and YAP3 [19] implement versions of the RKR-GST algorithm; however, JPLAG uses a unique set of run time optimizations to increase its run time efficiency [5] [16]. Greedy-String-Tiling (GST) is an algorithm that attempts to find all maximal substring matches within two strings. This is done by searching the two strings for the longest contiguous substring and then marking that substring as a tile to prevent it from being identified as a longest common substring in later iterations. It repeats these steps until all non-overlapping tiles greater than a minimum-match-length are found. The set of tiles represents the fingerprint for a document [12]. The algorithm for Greedy-String-Tiling from [18] is summarized as follows:

1. *Search the two strings being compared for the longest contiguous substring.*
2. *For each non-overlapping match of maximal length, where maximal length is the length of the longest contiguous substring found in Step 1, mark the string as a tile making it unavailable for further matches.*
3. *Repeat Steps 1 and 2 until no further tiles of length >= m are found, where m is a minimum-match-length threshold.*
4. *Return the set of tiles marked in all iterations of Step 2.*

The Running-Karp-Rabin (RKR) algorithm is designed for fast substring matching and finds all occurrences of a short string P within a longer string T by hashing all substrings of length $|P|$ and then comparing all hash values from T with the hash value of P [12]. The two algorithms work together in this way: RKR is run on two documents. For each substring identified as a match by RKR, GST is run on substring in order to extend the match beyond the bounds of the hash value. Finally, a distance measure is applied to the matched tiles to complete the process [12].

2.4 GPLAG

All previously discussed approaches to plagiarism detection utilize a central idea of tokenization, replacing language specific keywords with identifiers (Section 3.2). In [9], an approach is presented to detect plagiarism not based on token summaries of source code. Rather, it is based on the procedure-call dependencies of the program which can be summarized in a graph. The authors of [9] analyze the drawbacks of token-based plagiarism detection algorithms which include sensitivity to code insertion, code reordering, and control statement replacement (e.g., replacing a *for* loop with an infinite *while* loop and a break statement).

To overcome these weaknesses, the authors of [9] present a new system called GPLAG, which uses Program Dependence Graphs (PDGs) rather than tokens to identify likely plagiarism. The program dependence graph G for a procedure P is a 4-tuple element $G=(V, E, \mu, \delta)$, where V is the set of program vertices in P, E is the set of dependency edges, μ is a function assigning types to program vertices, and δ is a function assigning dependency types, either data or control, to edges [9]. Each program vertex is a key type identified by the authors, and a dependency edge between vertices v_1 and v_2 signifies that the execution of v_2 depends on the execution of v_1.

The authors then identify any sub-graph isomorphism between two compared PDGs G_1 and G_2 and record all matching sub-graphs between the PDGs for two source code documents represented by G_1 and G_2. By identifying the sub-graph similarities between source code documents rather than the token set similarities as required by previously mentioned approaches, the authors show empirically that this approach is not only competitive with current approaches in the literature, it is also robust against the weaknesses of these approaches [9].

This robustness can easily be understood by examining the discussed weaknesses of the other approaches. Reordering code has no effect on the PDG of a source code document whatsoever, since the PDG is based solely on the dependence of structures in the source code and not the position of the structures. In addition, while both code insertion and control statement replacement have an effect on the overall PDG, neither of the two affect the ability of GPLAG to identify likely plagiarism, since GPLAG is dealing with sub-graph isomorphism rather than graph isomorphism.

Despite GPLAG's advantages over token-based approaches, it tends to expect source code to contain function/procedure calls. These are not covered in our introductory programming course and are therefore nearly absent in our dataset. And so despite GPLAG's promise to overcome weaknesses in token-based approaches, it would perform poorly on corpora like ours that lack rich levels of dependencies.

3. DETECTION APPROACH

In this section we describe the process of converting source code into a compressed format that captures the logic embedded in the code without the specific programming language syntax, making it more conducive to logic-based code comparisons. We begin by performing cleaning in which we eliminate portions of the code that do not express its logic. Then we move to tokenization in which the logic of the code is reduced down to a concise string of tokens. Finally, k-grams are extracted, allowing us to perform direct comparisons. It should be noted that such preprocessing steps are applied to the data repository off-line and then to each new submission at comparison time.

IPPDC aims to discover possible cases of plagiarism and report them to a human operator for manual inspection. The essential approach involves comparing each source code document to all others in a pairwise fashion. Later we will describe how we use data clustering to greatly reduce the number of comparisons performed (Section 6), but first we will explore the fundamentals of IPPDC's plagiarism detection process.

3.1 Cleaning

Stylistic or excess whitespace (that which is not necessary for correct execution) and comments are a part of most source code documents. However, they do not affect the program's execution and contribute to extraneous or uninteresting *k*-grams, so they are removed during the cleaning process. In addition, it should be noted that comments are removed from source code documents in order to preserve student privacy.

```
      Dim addr     as String 'The address
```

(a) A line of Visual Basic code including an indentation, extra spaces and a comment

```
Dim addr as String
```

(b) The line after cleaning

Figure 3.1: The cleaning process.

Figure 3.1 demonstrates clearly that all essential information has been retained in the shown sample code while irrelevant parts have been removed. The excess whitespace or comments may have been intentionally altered by a plagiarizer, making cleaning essential for two reasons: (1) it defends against Type I plagiarism, and (2) IPPDC's similarity metrics (Section 3.4.1) rely on the *k*-grams that represent a document being as relevant as possible.

3.2 Tokenization

After cleaning, each document is converted into a string of tokens representing the original source code. A token is an abbreviated string corresponding to some particular keyword or phrase. Tokenizing produces a more condense version of a document (a token string) that is robust against simple disguising techniques such as variable name changes (Type II plagiarism) that have no effect on program execution.

The effectiveness of tokenization depends on the chosen token set. A token set one might call *strict* could represent an integer declaration with the token *INT*, a short integer declaration with *SHR*, and a long integer declaration with *LNG*. A *loose* token set could represent all three with the token *INT*, recognizing that all three may serve the same purpose in most situations. Thus, a loose token set helps the system know which statements can be considered equivalent and therefore provides resilience to Type II and III plagiarism. This includes alterations that do not affect program correctness such as changing a *for* loop's counter variable from an integer to a long integer or replacing a *while* loop with an equivalent *do-while* loop. For this reason, IPPDC uses a loose token set, a portion of which is shown in Figure 3.2.

```
PUBLICSINGLE        DN
PUBLICDOUBLE        DN
PUBLICINTEGER       DN
PUBLICLONG          DN

STARTIF             (I
STARTSELECT         (I
ENDIF               I)
ENDSELECT           I)

STARTFOR            (L
STARTDO             (L
STARTWHILE          (L
ENDFOR              L)
ENDDO               L)
ENDWHILE            L)

ASSIGNMENT          ==
```

Figure 3.2: Part of a loose token set for Visual Basic.

While source code syntax is specific to a given programming language, we argue that the token set that represents it should be universal. A token called *BEGINLOOP* could represent the start of a *while* loop in C++, a *for* loop in Java, or a *repeat-until* loop in Pascal. While IPPDC has only been tested on a dataset of Visual Basic submissions (Section 4), it could detect plagiarism between source code documents in different languages given small changes to the tokenization process.

The tokenization process thus generates a representative token string for each source code document. Before proceeding further, IPPDC at this point discards documents which contain only a small number of tokens, by default 20. Documents this small will generate only a few *k*-grams and cannot possibly contain any substantial amount of plagiarized code.

3.3 *k*-grams Generation

Token strings are next broken into *k*-grams as discussed in Section 2.1. The appropriate value for *k* is not obvious, so we chose this parameter based on experimental results (Section 4.1). For each submission, IPPDC stores the document's *k*-grams along with their frequencies, i.e., the number of occurrences of that *k*-gram within the document. Figure 3.3 shows an example of the stored *k*-gram information.

```
(I==I)(L        12
==I)(L==         8
(L(L(L(L         2
```

Figure 3.3: A document's *k*-grams and associated frequencies where *k=4*. Note that each token such as == is two characters in length.

3.4 Pairwise Comparison

IPPDC is designed to maintain a large set of previously-seen documents against which to check a set of submitted documents for possible plagiarism. Throughout this work, we will refer to the former set as database documents and the latter set as submitted documents. In this next step, each submitted document is compared to every other submitted document and to every database document.

To perform the comparisons, the documents' lists of *k*-gram frequencies are treated as vectors as per Section 2.1 and compared using one of three similarity metrics described in Section 3.4.1. Any *k*-gram appearing in one document but absent in a second is considered to have a frequency of 0 in the second document's vector when the two are compared. For each pair of documents being compared, we will call the first document's vector *p*, the second's vector *q*, and the number of elements (or the number of unique *k*-grams) *n*. IPPDC supports comparison using three different similarity metrics: *Euclidean*, *Manhattan*, and *Cosine*.

3.4.1 Similarity Metrics

Euclidean similarity is a simple variation of Euclidean distance, or the Minkowski L_2 norm (Figure 3.4). The Euclidean distance between vectors *p* and *q* are normalized to a range between 0 and 1 via division by the maximum distance of any vector in the dataset. Finally, Euclidean similarity is calculated by subtracting the normalized Euclidean distance from one [2] (Figure 3.4).

$$dist_{Euclidean} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}$$

$$sim_{Euclidean} = 1 - \frac{dist_{Euclidean}}{\max(dist_{Euclidean})}$$

Figure 3.4: The formulas for Euclidean distance and Euclidean similarity.

Manhattan similarity is calculated correspondingly but using Manhattan distance, or the Minkowski L_1 norm (Figure 3.5).

$$dist_{Manhattan} = \sum_{i=1}^{n} |p_i - q_i|$$

$$sim_{Manhattan} = 1 - \frac{dist_{Manhattan}}{\max(dist_{Manhattan})}$$

Figure 3.5: The formulas for Manhattan distance and Manhattan similarity.

Cosine similarity corresponds to the cosine of the angle between the two vectors representing *p* and *q* in *n*-dimensional space. It is commonly used when comparing documents in text mining and information retrieval [11]. As shown in Figure 3.6, Cosine similarity is found by dividing the dot product of vectors *p* and *q* by the product of their magnitudes.

$$sim_{Cosine} = \frac{p \cdot q}{\|p\|\|q\|} = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} {p_i}^2}\sqrt{\sum_{i=1}^{n} {q_i}^2}}$$

Figure 3.6: The formula for Cosine similarity.

3.4.2 TF*IDF Weighting

Some programming patterns will be very frequent in a set of source code documents simply because they are common coding practice (e.g. two nested *for* loops containing an assignment) or because they are boiler plate code. This will be particularly true of submission sets from an academic setting such as a programming assignment in a course where the instructor provided a sorting algorithm. To a naïve plagiarism detection system, many documents in this set might appear very similar even when no actual plagiarism is present.

Our solution is to utilize a technique commonly used in the text-mining and information retrieval literature known as term frequency by inverse document frequency (TF*IDF) weighting, which will give less weight to k-grams appearing more frequently in the dataset and more weight to highly unique k-grams. Term frequency (TF) is the number of times a k-gram t appears in a document d which is then normalized via division by the maximum TF of any k-gram in the document. Inverse document frequency (IDF) is defined as $log_2 \frac{N}{N_t}$ where N is the number of documents in the dataset and where N_t is the number of documents containing t. IDF is normalized in the same fashion as TF [13]. Now when computing the similarity metric values described in Section 3.4.1, the TF*IDF of each k-gram frequency is used rather than the simple frequency.

4. ACCURACY

The foremost purpose of IPPDC is to accurately detect likely cases of plagiarism, making detection accuracy our primary goal. It is only after achieving useful accuracy that the optimizations described in Sections 5, 6, and 7 become meaningful. We will first show how we chose an optimal k value and similarity metric (Section 4.1). Note that throughout this paper, we use the term accuracy in a general sense and are never referring to the statistical measure.

As aforementioned, our dataset consists of Visual Basic projects submitted by students in an introductory computer science course; 2935 of these documents (or project forms) meet the minimum token length requirement (Section 3.2). In our tests, these will be our initial database, or the documents which would have been previously submitted and then stored. From these, we have manually plagiarized 180 documents to various degrees: 30 of each plagiarism type (I through V) and 30 which include multiple types of plagiarism. These 180 will be our submission documents, where a perfect system would detect all 180 as plagiarized from their original versions.

In our accuracy tests, we primarily use the F_1 score, a common measure that quantifies a combination of P (precision) and R (recall), specifically their harmonic mean [16]. These metrics are defined in Figure 4.1.

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

$$F_1 = \frac{2PR}{P + R}$$

Figure 4.1: Definition of precision (P), recall (R), and F_1 score.

TP (true positives) is the number of manually plagiarized documents that are correctly classified as plagiarized from their original versions. FP (false positives) is the number of manually plagiarized documents that are classified as plagiarized from any document other than their original versions. FN (false negatives) is 180 (the number of manually plagiarized documents) minus TP, or the number of plagiarized documents that were not correctly classified.

All tests in this paper are performed using TP, FP, and FN calculated based on IPPDC's closest 200 matches, or the 200 pairs of documents that the system reports as being the most similar. MOSS reports its results equivalently, making the comparisons between the two approaches more meaningful.

4.1 Finding Optimal Parameters

Realizing that no specific pair of a k value and a similarity metric will be optimal in all cases, we sought only to find an effective pair for our dataset. One k value and similarity metric pair that results in a higher F_1 score than a second pair is considered better than that second pair.

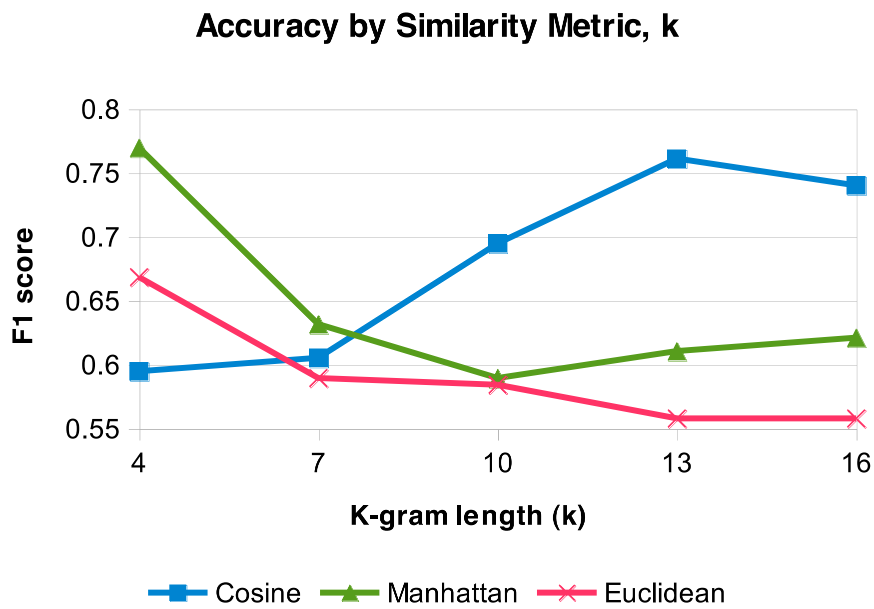

Figure 4.2: F_1 scores of various combinations of similarity metric and k.

As depicted in Figure 4.2, the best two pairs of parameters tested were Manhattan similarity where $k=4$ and Cosine similarity where $k=13$. For simplicity and clarity, we chose to report most other tests (Sections 4.2, 5, 6, 7) using a single pair. While the Manhattan pair resulted in very slightly higher accuracy, we tend to display only the Cosine pair because it ultimately produced better clustering results (Sections 6, 7).

Euclidean similarity is rarely used to compare vectors, but we included it as one of our metrics for benchmarking purposes. It is therefore not surprising that it was consistently outperformed by the other two metrics. There likely is no perfect combination of similarity metric and k value that will perform best with every dataset. However, we will use Cosine similarity where $k=13$ and Manhattan similarity where $k=4$ for the purposes of this paper, as we are testing only on our single dataset where this combination works well.

4.2 Comparison with MOSS

As aforementioned, after IPPDC performs pairwise comparisons, it returns the 200 pairs of documents which it found to be most similar of all document pairs in order from most to least similar. For example, if one document is an exact copy of another, then this would likely be the first pair reported, and a pair of very similar but slightly different documents might be the second. Any document pairs found to be less similar than the first 200 are assumed not to be plagiarized and are not returned.

MOSS reports its results in the same fashion: the most similar pair of documents is returned first, then the second most similar, etc. We computed the F_1 score for MOSS and for IPPDC using the 200 most similar document pairs reported by each system using the same method and the same input documents (2935 original plus 180 manually plagiarized) as described in Section 4. Figure 4.3 shows F_1 scores of MOSS and of IPPDC using the two best pairs of similarity metric and k value as per Section 4.1, Manhattan similarity where $k=4$ and Cosine similarity where $k=13$. Accuracies for detecting each plagiarism type I through V as well as mixed types are shown for each system.

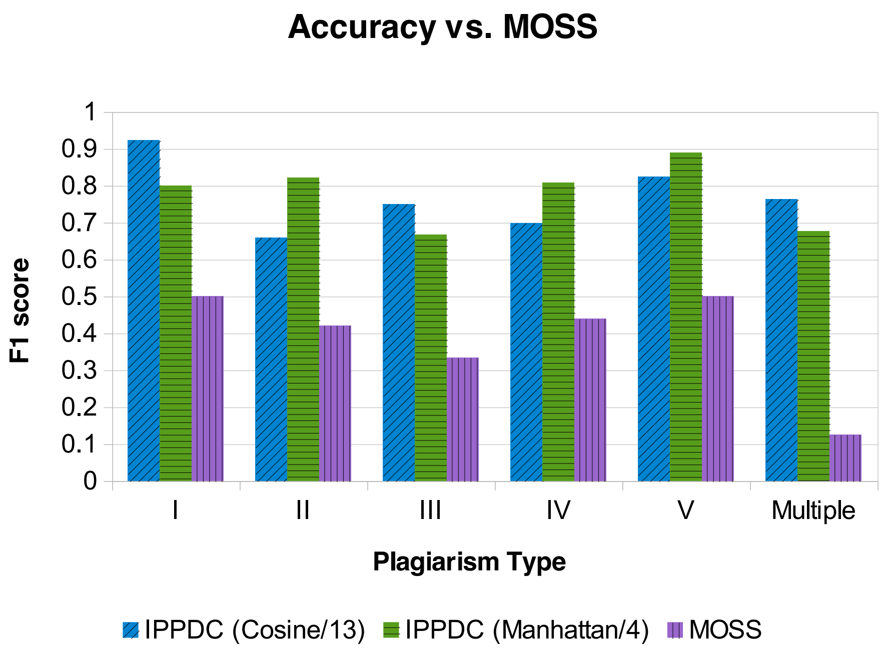

Figure 4.3: F_1 scores of IPPDC and MOSS with respect to documents containing each single type of plagiarism from Figure 2.1 as well as multiple types. Shown are IPPDC results when using Cosine similarity where *k=13* and Manhattan similarity where *k=4*.

The two IPPDC configurations shown in Figure 4.3 performed very comparably. The Cosine version better detected the manual plagiarism of Types I and III as well as multiple plagiarism types at once. The Manhattan version performed better on plagiarism types II, IV, and V. MOSS detects plagiarism types I and V best and performs poorly on the documents which contain multiple types of plagiarism. IPPDC outperforms MOSS' detection accuracy for all types. Having empirically established useful detection accuracy, we proceed to discuss efforts at increasing IPPDC's runtime efficiency without reducing this accuracy.

5. PARALLEL APPROACH

As physical barriers continue to prevent the development of significantly faster CPU cores, the computing world is becoming increasingly dependent on efficient multi-threaded programming. For this reason, we optimized IPPDC for parallel execution in an effort to reduce its runtime. We focused on developing a parallel programming approach for the document comparison step (Section 3.4), as this makes up the vast majority of the total system runtime. Relative to the pairwise comparisons, all other steps take a negligible amount of time to complete, so we deemed it unnecessary to develop parallel algorithms for them.

The environment for our parallel tests (Sections 5.3, 7) is the following:

OPERATING SYSTEM:	*Fedora 14 Linux x86_64 with Linux kernel version 2.6.35.14-95*
LANGUAGE:	*Java version 1.6.0_30*
PROCESSOR:	*AMD Opteron 6168 (12 cores at 1.9GHz; 128KB x12 L1 Cache; 512KB x12 L2 Cache; 12MB shared L3 Cache) – x4 (48 cores total)*
MEMORY:	*2GB DDR3 SDRAM ECC Unbuffered DDR3 1333MHz Memory – x32 (64 GB total)*

The SDRAM is divided equally among the four Opteron processors, 16GB each. The machine is a Non-Uniform Memory Access (NUMA) system. Applicable characteristics of NUMA architectures are explained in Section 5.2. When performing sequential (or single-core) runtime tests (Sections 5.3, 6.5, 6.6), the same environment is used apart from utilizing only one of the forty-eight CPU cores.

5.1 Fork/join Tasks

Starting in Java 7, a fork/join framework FJTask is available for use in the Java standard libraries. We used this framework as an external library in Java 6 because our test environment did not support Java 7. Fork/join parallelism follows the standard divide-and-conquer strategy of the following form [7]:

```
if (problem is small)
    directly solve problem
else
    split problem into independent parts
    fork new subtasks to solve each part
    join all subtasks
    compose result from subresults
```

Fork/join algorithms naturally tend toward recursion. The FJTask framework is designed with this approach in mind, creating a pool of persistent worker threads that execute a queue of lightweight tasks [7]. Thus, the overhead of creating and destroying threads can be avoided even with large numbers of tasks.

Furthermore, this framework provides support for work-stealing. If one worker thread has tasks queued but is busy whereas another thread is idle, the second thread can steal tasks from the first [7]. This results in better utilization of all processors and better overall runtimes.

5.2 Implementation

Because submitted documents are expected to be far fewer in number than database documents, our target region for adding parallelism within the pairwise comparison section can become yet more specific: the database. Our general approach is

1. *Perform cleaning, tokenization, and k-gram generation sequentially on the submitted documents.*
2. *Combine both submitted and database documents into one large list, allDocs.*
3. *Perform pairwise comparisons in parallel by having each task compare every submitted document to its unique and disjoint sub-list of allDocs.*
4. *Combine the results of all tasks and report results.*

In general, manually rewriting a recursive algorithm using loops will tend to result in better runtimes. Knowing this, we performed early tests comparing:

1. *a recursive algorithm using FJTask,*
2. *a loop-based algorithm using FJTask, and*
3. *(for comparison purposes only) a loop-based algorithm using the standard Java Thread class, Java's implementation of native operating system threads.*

In version 1, the full *allDocs* list is given to the first task, and it is then split recursively until a threshold number of documents is reached (Section 5.1), after which the comparisons are performed. For example, if there are 50 submitted documents and 950 database documents, *allDocs* will be of size 1000, and version 1 would begin by forking two FJTask tasks each responsible for 500 documents. These two would each fork again, resulting in four tasks each responsible for 250 documents. This process would continue until reaching the threshold after which all comparisons would be performed, and the results then would be communicated up the tree of tasks in reverse of the order in which they were created.

In both loop-based versions (2 and 3), the main thread immediately divides *allDocs* evenly among a set number of FJTask tasks or Java Threads, respectively. For example, if *allDocs* is of size 1000, and we wish to use 20 processors, then 20 tasks or Threads would be immediately created and responsible for 50 documents each. Each task or Thread directly performs the comparisons, and the results are communicated back to the main thread where they are combined.

As we expected, the loop-based FJTask implementation (version 2) soundly outperformed the recursive implementation (version 1). It also consistently outperformed the loop-based version using standard Java Threads (version 3) which we did not predict. We expect the cause to be the optimizations present in the framework that our Java Threads implementation lacked. Ultimately, we are most concerned with reducing the runtime of comparisons as much as possible, and thus we only show performance results (Section 5.3) using this fastest implementation.

IPPDC's parallel execution is optimized specifically for the NUMA (non-uniform memory access) architecture. In a NUMA system, processors are distributed among NUMA regions, and each portion of main memory can be accessed faster by processors in the corresponding NUMA region than it can be accessed by others [8]. We will refer to this portion of memory as local to the group of processors in that region and all other portions of memory as non-local to that group. A parallel algorithm targeting the NUMA architecture must be acutely aware of memory locality, minimizing the number of accesses to slower non-local memory. One method of accomplishing this is to direct each processor individually to allocate memory for the data it will use. This forces the data needed by each processor to be allocated in memory local to it which quickens later accesses to this data [8]. For this reason, our most important NUMA optimization is simple but effective: each task copies its lists of documents to local memory before performing comparisons.

5.3 Performance

In the following tests, runtimes refer only to the time required to perform pairwise comparisons, as the remainder of the system runtime is both fairly constant and insignificant, particularly with the large datasets used in these tests. The test consisted of submitting our 180 manually plagiarized documents (Section 4) to a database of various sizes and timing the pairwise comparison step when run sequentially and then when run using our parallel implementation.

The metric used to quantify the efficiency of our parallel solution is called parallel speedup. Parallel speedup is defined as the sequential (single-core) runtime divided by the parallel (in this case, 48-core) runtime (Figure 5.1) [8]. A perfect parallel algorithm will, except under special circumstances, not exceed a speedup equal to the number of processors used (here, 48). This is referred to as linear speedup [8]. In other words, if our sequential runtime is 48 times as large as our parallel runtime, we would have achieved linear speedup because we are testing with 48 cores. If our sequential runtime is 24 times as large as our parallel runtime, our parallel speedup is 24.

$$parallel\ speedup = \frac{runtime_{sequential}}{runtime_{parallel}}$$

Figure 5.1: The definition of parallel speedup.

When constructing a parallel version of an originally-sequential algorithm, some additional time will always be required. In our case, this will include time to create tasks or threads, to destroy them after the comparisons are finished, and to communicate data (the documents to be compared) to the tasks or threads. This additional time is referred to as parallel overhead, the overhead required by the parallel algorithm which was not present in the sequential algorithm [8]. Achieving linear speedup might mean the parallel algorithm is 100% efficient in how it divides up work and has negligible, near-zero parallel overhead. This is very rare. Speedup which is linear or even faster commonly results from special circumstances which are described later in this section.

In Figure 5.2, the numbers of database documents tested were chosen because 11740, 17610, and 23480 are four times, six times, and eight times our original database size (2935 documents), respectively. We generated these datasets by duplicating the database documents. This is acceptable because the uniqueness of documents has no effect on comparison time. Figure 5.2 shows our achieved parallel speedups when performing IPPDC's pairwise comparisons in sequence and then in parallel using all 48 available processors.

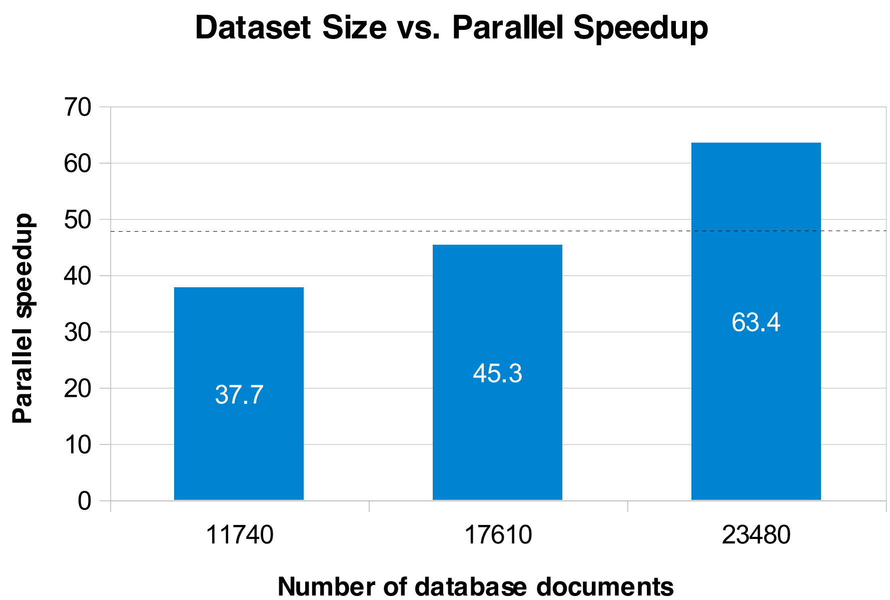

Figure 5.2: Parallel speedup as the dataset becomes larger. The dashed line at 48 represents linear speedup.

As mentioned, a speedup of 48 represents linear speedup with our 48-core machine. The speedup of 37.7 when using the smaller dataset is acceptable, but when using the middle-sized dataset, the speedup becomes even better, almost linear.

We experience super-linear speedup (faster than linear speedup), specifically 63.4, when using the largest dataset. Generally speaking, this is an extremely rare phenomenon. As described previously, even linear speedup is uncommon. Super-linear speedup typically results when the running program or its data end up in different parts of the memory hierarchy (usually the CPU caches or main memory specifically) when comparing sequential to parallel executions.

In our case, we believe super-linear speedup occurred due to the main memory configuration on our NUMA system: the largest dataset did not fit entirely into the local memory of the single processor used to perform the sequential comparisons, so some memory accesses were made to non-local (much slower) memory as shown in Figure 5.3. Parallel execution allowed full use of the 64 GB of main memory while making all, or nearly all, memory accesses local ones because the dataset was split between all 48 cores.

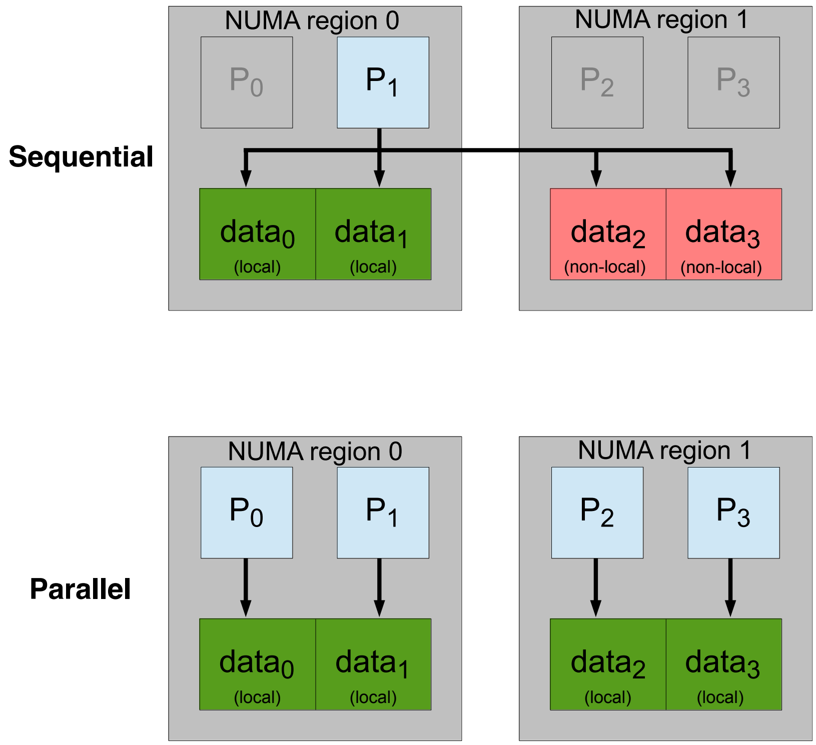

Figure 5.3: An example data layout which might result in super-linear speedup. Sequentially, P_1 must make non-local memory accesses to half of the required data, as the data does not fit entirely in memory local to P_1. The parallel run makes all local accesses.

Thus, we see that not only does our parallel implementation produce useful speedups—nearly linear under typical conditions—but as the dataset grows increasingly large, we see additional performance gains due to our parallel algorithm's utilization of memory locality.

6. CLUSTERING APPROACH

The above parallel optimizations have been shown to be crucial to IPPDC's usefulness as its database of previously-seen documents increases in size. However, such improvements are limited by available processors and computing power. In addition, our approach so far is brute-force-driven in the sense that it compares each newly submitted document to every document in the database. If the database were to double in size, the number of required comparisons would also double. If IPPDC were used with massive corpora on the order of millions of documents, the number of required comparisons could result in unacceptable runtimes. This could so far be remedied only by finding a faster machine with more available processors. To provide another alternative, we propose an intelligent solution using data clustering to greatly reduce the number of pairwise comparisons necessary from the start, further improving IPPDC's ability to scale to very large databases.

Data clustering is a data mining technique which aims to place objects into logically similar groups or clusters where objects within a cluster are more similar to one another than they are to objects in other clusters [11]. The precise definition of what defines a cluster and how to create them will vary based on the chosen algorithm. Density-based and partitional clustering algorithms are two of the more popular approaches. We implemented one algorithm from each category in IPPDC. Note that it is common to refer to objects being clustered as data points, which we will do also. Our data points will be our database documents, and distance between data points is found by comparison of vectors of k-gram frequencies as in Section 3.4.

6.1 Density-based Clustering

Density-based algorithms create clusters of high-density areas separated by areas of low density. One of the most popular algorithms of this type is DBSCAN (density-based spatial clustering of applications with noise) [3]. DBSCAN uses its two parameters, $minPts$ and ε, to categorize every data point (here, database document) as a core point, a border point, or a noise point. A point's ε-neighborhood consists of all points, including itself, that are within distance ε from the point. A core point contains at least $minPts$ points in its ε-neighborhood. A border point is in a core point's ε-neighborhood but is not a core point itself. All other points are noise points.

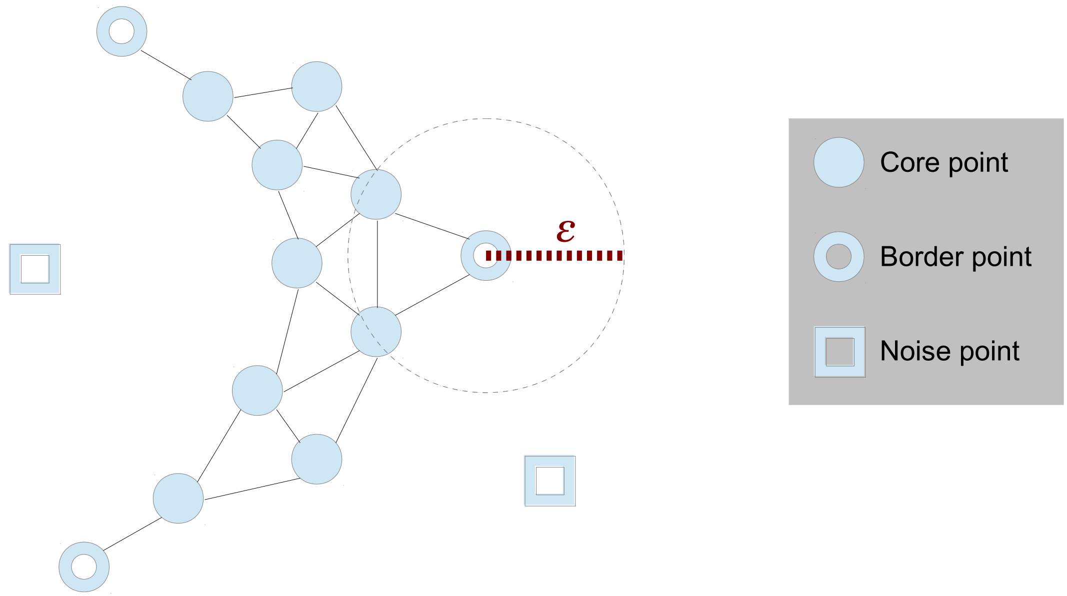

Figure 6.1: DBSCAN visualized when *minPts* is 4. The area within the dashed circle represents the ε-neighborhood of the rightmost border point. Solid lines connect points that are within ε of one another.

After each point is classified, noise points are eliminated and all core points in the ε-neighborhood of each other are connected to form clusters. Next, border points become part of their corresponding core point's cluster. This is repeated until there are no changes to the cluster configuration. In the example of Figure 6.1, all points connected by solid lines are part of the same cluster. When using the DBSCAN algorithm, the number of clusters is not a parameter because it depends on *minPts*, ε, and the data.

6.2 Partitional Clustering

While DBSCAN starts with individual points and creates clusters from the bottom up, partitional algorithms create clusters by starting with the entire dataset and then breaking it into groups. *K*-medoids is a partitional clustering algorithm which represents clusters by their medoid. A medoid is the representative center of a cluster that is also a valid data point [6], corresponding to the median of a set of numbers. The algorithm takes one parameter which we will call $\mathcal{K}$ to avoid confusing it with the length of a k-gram.

PAM (partitioning around medoids) is a popular implementation of k-medoids [6]. The algorithm is as follows:

1. *Randomly choose $\mathcal{K}$ initial medoids from all data points.*
2. *Assign all non-medoids to their closest medoid.*
3. *Iterate over the medoids, replacing them with a non-medoid if the swap would reduce the overall cost (the total distance between each non-medoid and the medoid closest to it) of the clustering configuration.*
4. *Repeat steps 2 and 3 until an iteration passes without any medoids changing, i.e., until the algorithm converges.*

6.3 Implementation

As aforementioned, what we call data points in Sections 6.1 and 6.2 correspond to our source code documents. Manhattan and Euclidean similarities described in Section 3.4.1 are originally distance metrics before modification, and Cosine similarity can be converted into a distance metric by subtracting it from one [11]. Thus, we already have a notion of both data points and distance between them, making the implementation of the DBSCAN and PAM algorithms straightforward.

Less clear is which documents to cluster and how to proceed after clustering. Clustering large numbers of documents will be far more time-consuming than comparing them, so IPPDC only clusters its database documents periodically, thereby using idle time to reduce comparison time during future submissions. When documents are submitted, each document is compared only to database documents belonging to the closest C clusters. For example, if there are 64 clusters of documents and $C=2$, each submitted document is only compared to documents in 2 of the 64 clusters, greatly reducing the number of comparisons performed.

For each submitted document, the C clusters most similar to the document are found. For PAM, this is straightforward. Each of the $\mathcal{K}$ clusters is already represented by a medoid document, so the similarity between each medoid and the given submitted document is found using the same combination of similarity metric and k as will be used for pairwise comparisons. The clusters represented by the C medoids that are most similar to the submitted document are the C closest clusters.

Density-based clustering algorithms such as DBSCAN often produce non-globular (e.g., in 2 dimensions, non-circular) clusters that cannot be accurately represented by a single mean or medoid point like PAM clusters can be. For this reason, we developed an algorithm to assign index points to clusters. An index point is a representative point for a cluster, and a cluster may have multiple index points. Non-globular clusters will be particularly well-represented by multiple index points so that the true shape of the cluster—not merely the center or average—is captured. Recall that all data points—including index points—are database documents in IPPDC's clustering implementation. The algorithm takes one parameter δ which is the minimum distance between index points. A large value for δ will result in fewer index points per cluster, reducing the number of pairwise comparisons to submitted forms later but also potentially representing the cluster less accurately. The opposite is true of a smaller δ value. The algorithm is as follows:

```
For each cluster i
    For each point p in i
        If p is not within the chosen distance δ of any current index points for i, p is now an index
        point for i.
```

To find the closest clusters to a submitted document, the document is compared to all index points from all clusters. Whichever cluster has the index point most similar to the submitted document is the most similar cluster, and the cluster with next most similar index point is the second most similar cluster. In this way, the C closest clusters for the submitted document are found.

6.4 Comparing DBSCAN and PAM

We tested our best two parameter pairs (Section 4.1) with both DBSCAN and PAM clustering using the same 2935-document dataset as in Section 4. Our foremost concern is to retain plagiarism detection accuracy. Only after this stipulation is met does comparison runtime become a concern.

As noted in the captions, Figures 6.1, 6.2, and 6.3 all depict results using $C=1$, as comparing each submitted document to larger numbers of close clusters only occasionally increased true positives slightly but most often had no effect at all apart from increasing comparison runtime.

DBSCAN performed significantly better when using Manhattan similarity, while PAM worked best with Cosine similarity. DBSCAN was able to achieve accuracies corresponding to those seen with PAM only when using values for ε and *minPts* that resulted in significantly higher comparison times as per Figure 6.2. Ultimately, we found the best choice of DBSCAN parameters for our dataset to be $\varepsilon=0.0017$ and *minPts=3*. In Figure 6.2, we show PAM results where *Ж=128* for reasons explained in Section 6.5.

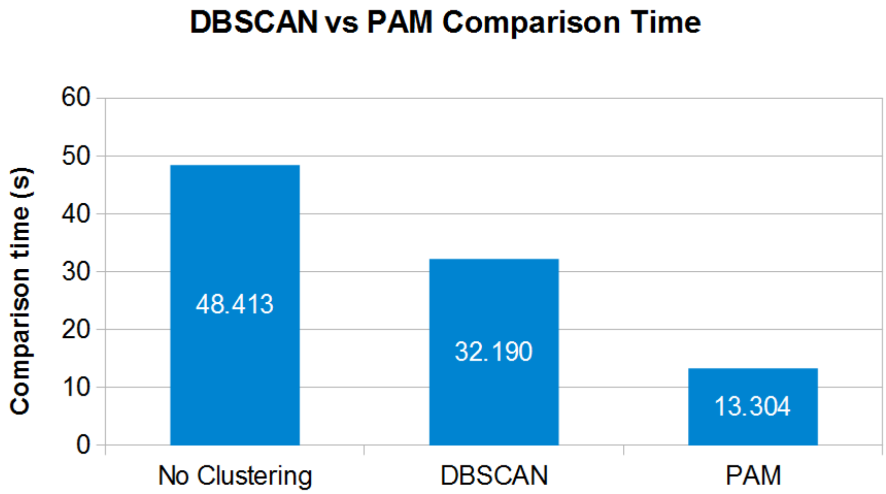

Figure 6.2: Comparison times of IPPDC without clustering, after clustering with DBSCAN where $\varepsilon=0.0017$ and where $minPts=3$, and after clustering with PAM where $K=128$. Each submitted document is compared to its single closest cluster ($C=1$).

Because comparison times after clustering with DBSCAN were far worse, we concentrate on using PAM as IPPDC's clustering algorithm of choice henceforth.

6.5 Accuracy after Clustering

As mentioned, PAM clustering achieved better accuracies when using Cosine similarity where $k=13$ than when using Manhattan similarity where $k=4$. Not only did most clustering configurations using the former combination retain the original accuracies, but in some cases, accuracies were even slightly improved (Figure 6.3).

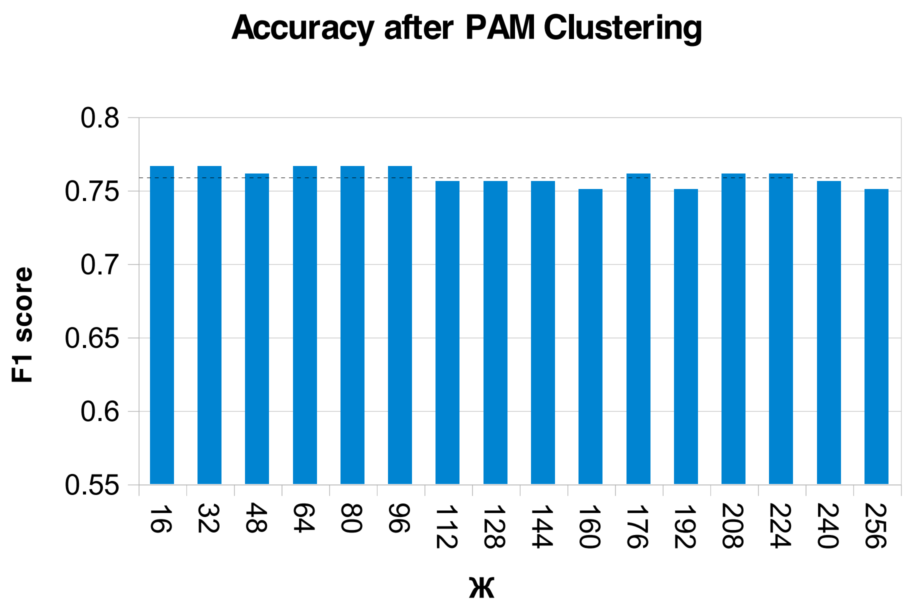

Figure 6.3: Accuracy when comparing each submitted document to its single closest cluster (*C=1*) with varying numbers of clusters (*Ж*). The dashed line is accuracy without clustering.

Accuracy variances with different *Ж* values in Figure 6.3 are not significant enough to point to an obvious best value. We therefore introduce pairwise comparison runtime results in Figure 6.4 merely as a method of choosing a best *Ж* value. The effect of clustering on comparison time will be explored further in Sections 6.6 and 7.

As per Figure 6.4, comparison runtimes initially drop significantly as *Ж* values are increased but level off around *Ж=128*. While comparison runtimes do continue to fall slightly at higher *Ж* values, the reductions are fairly insignificant whereas it takes 5 times as long to perform the initial clustering with *Ж=256* than with *Ж=128*. As per Section 6.3, clustering is performed occasionally and during idle time, and its runtime is far less significant than the time to perform the pairwise comparisons. However, clustering time must remain reasonable for the system to be useful, so we deem *Ж=128* a suitable value for IPPDC with our dataset.

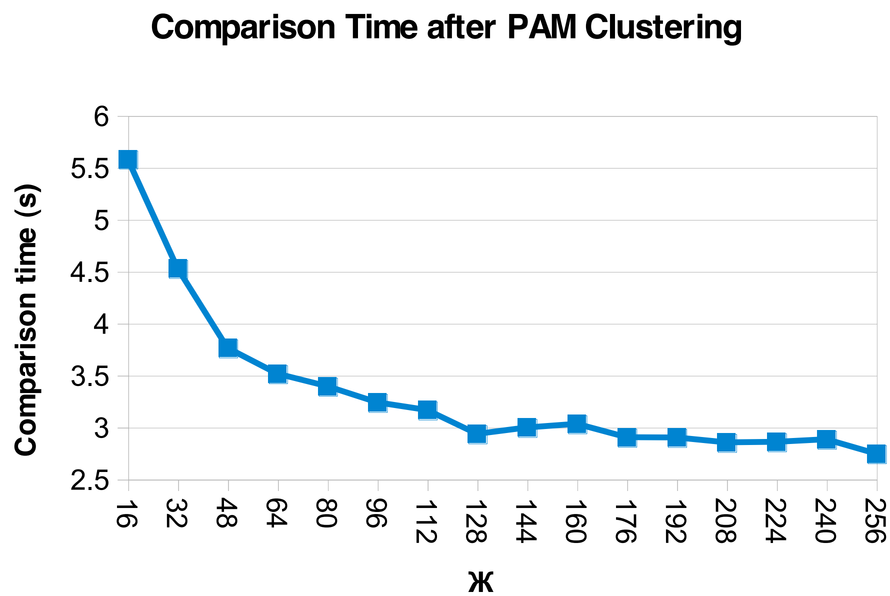

Figure 6.4: Pairwise comparison time when comparing each submitted document to its single closest cluster (*C=1*) with varying numbers of clusters (*Ж*).

6.6 Performance after Clustering

To illustrate the value of the proposed clustering strategy, we now aim to show that not only is accuracy maintained after clustering (Section 6.5), but useful reductions in comparison runtime are achieved. In Figure 6.5, we show the improvement in comparison runtime between our standard approach without clustering and our approach using PAM clustering where *Ж=128*, both without using our parallel strategy. We will again use the speedup metric introduced in Section 5.3, this time meaning the original brute-force comparison time divided by the intelligent comparison time after clustering. Recall that in Section 6.5, we found for our dataset that all values of *Ж* maintain good accuracy but that runtime reductions are less significant as *Ж* is increased above 128.

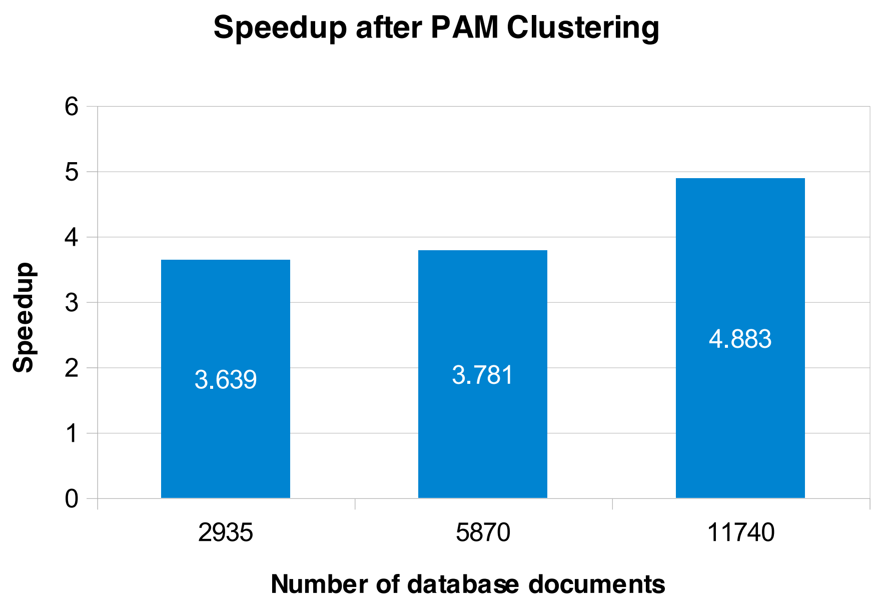

Figure 6.5: Comparison time speedup achieved after PAM clustering as opposed to comparison time without clustering using one, two, and four times the original dataset size.

With our original dataset and multiple duplications of it, our intelligent clustering solution consistently provides a speedup of better than three and half times. Our dataset is by nature somewhat homogenous because all projects are from the same course where basic programming structures and standard algorithms are taught. It is possible that a more heterogeneous dataset—such as projects from many different courses or universities—would afford our clustering solution more impressive speedups. In such a case, higher values of $\mathcal{K}$ would be appropriate to fit the sparse data meaning fewer database documents would be in a given cluster. If incoming documents were then compared (as they were in this paper) to only the single closest cluster, fewer comparisons would ultimately be performed resulting in potentially better speedups.

7. COMBINED APPROACH

Having presented the runtime reductions offered by our parallel solution alone (Section 5.3) and our clustering solution alone (Section 6.6), we will now explore the performance results of the combined system. In this setup, we use the clustering and parallel parameters that we found to be best in Sections 5 and 6. The database documents are first clustered using PAM (Section 6.2), and then pairwise comparisons are performed using the loop-based FJTask algorithm (Section 5.2).

The computational complexity of PAM is $O(k(n-k)^2)$ per iteration and is therefore fairly inefficient especially over large datasets [10]. This is ultimately not a significant concern for IPPDC: after the dataset is initially clustered, it needs to be clustered again only periodically, and the system may be used to detect plagiarism while the new clustering is being created. However, the complexity is inhibitive to performing experiments on very large datasets. Thus, we do not use the largest dataset utilized in Section 5.3 when we test the runtime of our combined solution but instead use only our second largest, 17610-document dataset (six times the size of our original dataset).

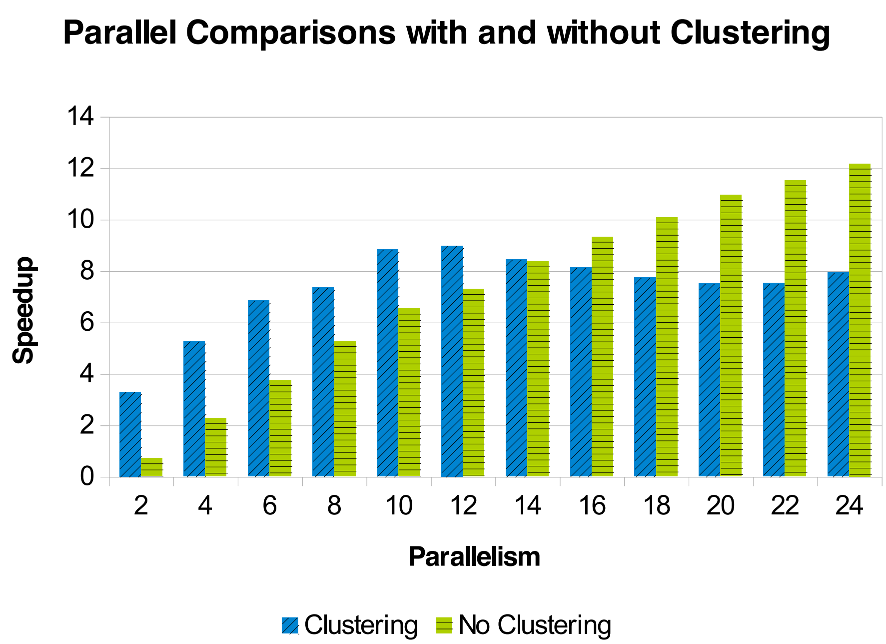

Figure 7.1: Comparison time for different levels of parallelism (number of processors used) with and without clustering using PAM where *Ж=128*.

Figure 7.1 compares speedups when using the combined system with speedups when using only the clustering solution with sequential comparisons. We see that IPPDC achieves its best parallel comparison runtimes using the intelligent clustering solution (Section 6.3) when a modest number of processors are available, specifically 14 or fewer with our dataset. Like any parallel algorithm, our brute-force parallel solution (Section 5.2) requires overhead—in part, splitting up the data between cores and communicating results back to the core on which the main thread is running. As the parallelism increases in Figure 7.1, the number of comparisons performed by each core decreases, while the overhead will either remain the same or more likely even increase. Since our clustering solution (Section 6.3) greatly reduces the number of comparisons necessary, the comparisons per core using our dataset duplicated six times clearly becomes too small to be useful at high levels of parallelism (about 16 or larger).

8. CONCLUSIONS

We have shown that by utilizing *k*-gram-based pairwise document comparison with optimizations for parallel NUMA architectures and an implementation of PAM clustering, our IPPDC system achieves high plagiarism detection accuracy with far lower runtimes than usually associated with similar comparison strategies. For those with datasets similar in size to ours (on the order of 10,000 documents) or smaller and with access to a highly-parallel machine, we presented promising runtimes using parallel document comparisons (Section 5). For those using datasets of any size without access to such a parallel computer, we presented useful runtime reductions using sequential comparisons after PAM clustering (Section 6). IPPDC using our combined parallel and clustering strategy (Section 7) requires further experimentation with very large datasets before a recommendation can be made. However, we suspect that those with such large datasets and a machine with two or more processing cores would see runtime improvements beyond what would be seen using either our parallel or clustering strategies independently.

9. FUTURE WORK

Optimal parameter values for various algorithms implemented in IPPDC may be variable based on the dataset. A technique such as Silhouette [14] might be implemented to more quickly select a suitable Ж value for PAM without requiring extensive tests of the entire IPPDC system. A parallel implementation of PAM could also be added to facilitate keeping the database up to date without requiring days or weeks to re-cluster after many new documents are added. Furthermore, many additional effective clustering algorithms exist apart from the two we use, some of which may potentially work well with our data. Testing all aspects of the system using other datasets, possibly in various other programming languages, might help facilitate additional improvements. It would be useful to further explore how different datasets affect optimal choices for k, the similarity metric, the clustering algorithm, and the corresponding parameters. Finally, further testing should be done with very large datasets using the combined clustering and parallel solution (Section 7).

10. ACKNOWLEDGMENTS

This work builds upon Jeremy Iverson and Dr. James Schnepf's implementation and development [4] of the plagiarism detection approach originally presented by Joseph Degiovanni and Dr. Imad Rahal [1].

11. REFERENCES

[1] Degiovanni J and Rahal I (2008). Towards efficient source code plagiarism detection: an n-gram-based approach. In: Proceedings of the ISCA 21st International Conference on Applications in Industry and Engineering. ISCA, Honolulu, Hawaii, pp 174-179

[2] Elmore KL and Richman MB (2001) Euclidean distance as a similarity metric for principal component analysis. *Monthly Weather Review*, 129(3): 540-549

[3] Ester M, Kriegel HP, Sander J, et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining. AAAI Press, Portland, OR, pp 226-231

[4] Iverson J and Schnepf J (2010). VOCS: an online clustering system for source code plagiarism detection. Unpublished BA thesis. Saint John's University, Collegeville, MN

[5] Karp RM and Rabin MO (1987) Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 31(2): 249-260

[6] Kaufman L and Rousseeuw PJ (1990) Finding groups in data: an introduction to cluster analysis. John Wiley & Sons, New York, NY

[7] Lea D (2000) A Java fork/join framework. In: Proceedings of the ACM 2000 Conference on Java Grande. ACM, San Francisco, CA, pp 36-43

[8] Lin C, Snyder L (2008). Principles of Parallel Programming. Addison-Wesley, Boston, MA

[9] Liu C, Chen C, Han J, and Yu PS (2006) GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, Philadelphia, PA, pp 872-881

[10] Ng RT and Han J (1994) Efficient and Effective Clustering Methods for Spatial Data Mining. In: Proceedings of the 20th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, pp 144-155

[11] Pang-Ning T, Steinbach M, Kumar V (2006). Introduction to Data Mining. Addison-Wesley, Boston, MA

[12] Prechelt L, Malpohl G and Philippsen M (2002) Finding Plagiarisms among a Set of Programs with JPLAG. Journal of Universal Computer Science 8: 1016-1038

[13] Rahal I, Wang B, and Schnepf J (2009) A primer on text-data analysis, In: (2nd ed) Encyclopedia of Information Science and Technology. IGI Publishing, Hershey, PA, pp 3111-3118

[14] Rousseeuw PJ (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics* 20: 53-65

[15] Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. ACM, San Diego, CA, pp 76-85

[16] Van Rijsbergen CJ (1977) A theoretical basis for the use of co-occurrence data in information retrieval. *Journal of Documentation* 33(2): 106-119

[17] Whale G (1990) Identification of program similarity in large populations. *The Computer Journal* 33(2): 140-146

[18] Wise MJ (1995) A System for Comparing Biological Sequences Using the Running Karp-Rabin Greedy String-Tiling Algorithm. In: Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology. AAAI Press , Cambridge, United Kingdom, pp 393-401

[19] Wise MJ (1996) YAP3: Improved Detection of Similarities in Computer Program and Other Texts. In:Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education. ACM, Philadelphia, PA, pp 130-134

12. APPENDIX

12.1 Selected Source Code

12.1.1 ProgramEngine

```
/**
 * Engine to run the entire IPPDC system
 *
 * @author Jeremy Iverson
 * @author Tony Ohmann
 * @author Fei Wu
 *
 */
public class ProgramEngine {

	/**
	 * Run the IPPDC system.
	 *
	 * @param args
	 *     Accepts the following options (ex. -proj /home/me/projs), all optional:
	 *
	 *            -proj    Directory of projects to be submitted (must exist)
	 *            -out     Directory for output
	 *            -trans   Translation table file (must exist)
	 *            -index   Index file
	 *            -db      Database directory
	 *            -partype Parallel comparison type (none, jthreads, forkjoin,
	 *                       forkjoinloop)
	 *            -par     Level of parallelism
	 *            -clust   Clustering algorithm to use (none, kmedoids, dbscan)
	 *            -nclust  Number of clusters, or 'k' (if clust=kmedoids)
	 *            -nclose  Number of closest clusters with which to compare each
	 *                       submission form
	 *            -minpts  DBSCAN's minPts parameter (if clust=dbscan)
	 *            -eps     DBSCAN's eps parameter (if clust=dbscan)
	 *            -dist    Distance metric (cosine, manhattan, euclidean)
	 *            -k       k-gram length
	 *            -cload   If clustering, whether to load the previously saved
	 *                       clustering ("true") or re-cluster ("false")
	 */
	public static void main(String[] args) {

		// Parameters with default values
		String projectsStr    = "./projects";
		String outputStr      = "./output";
		String translationStr = "./translationtable.tlt";
		String indexStr       = "./index.ind";
		String db             = "./db";
		String parallelType   = "none";
		int parallelism       = Runtime.getRuntime().availableProcessors();
		String clustering     = "none";
		int nClustOrMinPts    = 1;
		double nCloseOrEps    = 1.0;
		String distMetricStr  = "cosine";
		int kLength           = 13;
		boolean clustersLoad  = false;

		try {

			// Parse any passed options
			//     Note: the option's value is at [i], and the flag is at [i-1]
			for(int i=1; i<args.length; i+=2)

				if(args[i-1].equals("-proj"))
					projectsStr = args[i];

				else if(args[i-1].equals("-out"))
					outputStr = args[i];

				else if(args[i-1].equals("-trans"))
					translationStr = args[i];
```

```
				else if(args[i-1].equals("-index"))
					indexStr = args[i];

				else if(args[i-1].equals("-db"))
					db = args[i];

				else if(args[i-1].equals("-partype"))
					parallelType = args[i];

				else if(args[i-1].equals("-par"))
					parallelism = Integer.parseInt(args[i]);

				else if(args[i-1].equals("-clust"))
					clustering = args[i];

				else if(args[i-1].equals("-nclust") || args[i-1].equals("-minpts"))
					nClustOrMinPts = Integer.parseInt(args[i]);

				else if(args[i-1].equals("-nclose") || args[i-1].equals("-eps"))
					nCloseOrEps = Double.parseDouble(args[i]);

				else if(args[i-1].equals("-dist"))
					distMetricStr = args[i];

				else if(args[i-1].equals("-k"))
					kLength = Integer.parseInt(args[i]);

				else if(args[i-1].equals("-cload"))
					clustersLoad = Boolean.parseBoolean(args[i]);

		} catch(Exception e) {
			throw new IllegalArgumentException("ERROR: Malformed options.");
		}

		File output = null;
		File projects = null;
		File translation = null;
		int distMetric = -1;

		// Parameter validation
		try {

			// Projects directory
			projects = new File(projectsStr);
			if(!projects.exists())
				throw new IllegalArgumentException("ERROR: Projects directory does not
exist.");
			else if(!projects.isDirectory())
				throw new IllegalArgumentException("ERROR: Projects exists but is a file, not
a directory.");

			// Output directory
			output = new File(outputStr);
			if(!output.exists())
				output.mkdir();
			else if(!output.isDirectory())
				throw new IllegalArgumentException("ERROR: Output exists but is a file, not a
directory.");

			// Translation table file
			translation = new File(translationStr);
			if(!translation.exists())
				throw new IllegalArgumentException("ERROR: Translation table file does not
exist.");

			// Parallel comparison type and options
			if(parallelType.equals("jthreads") || parallelType.equals("forkjoin") ||
parallelType.equals("forkjoinloop"))
				if(parallelism < 0)
					throw new IllegalArgumentException("ERROR: Parallel comparisons
requested, but parallelism is negative.");
				else if(parallelism < 2)
					parallelType = "none";
			else if(!parallelType.equals("none"))
				throw new IllegalArgumentException("ERROR: Parallelism parameter must be in
{none, rootleaf, forkjoin, forkjoinloop}");

			// Distance metric
			if(distMetricStr.equals("cosine"))
				distMetric = 0;
			else if(distMetricStr.equals("manhattan"))
```

```
						distMetric = 1;
					else if(distMetricStr.equals("euclidean"))
						distMetric = 2;
					else
						throw new IllegalArgumentException("ERROR: Similarity metric parameter must
be in {cosine, manhattan, euclidean}");

			} catch(Exception e) {
				throw new IllegalArgumentException("ERROR: General parameter error.");
			}

			// Timings
			double compareTime = -1.0;
			double clusterTime = -1.0;

			// Indexes to hold global n-gram counts
			Index index = new Index(indexStr);

			// Removing formatting and comments (cleaning)
			CleanEngine ce = new CleanEngine();
			ce.run(projects, output, new File(db));

			// Converting code into tokens (tokenizing)
			TokenizeEngine te = new TokenizeEngine();
			te.run(output, translation);

			// Converting tokens into n-grams (ngrammizing)
			NGramEngine nge = new NGramEngine(index, kLength);
			nge.run(output);

			// Clustering [optional] and finding distance between forms (comparing)
			NGramCompareEngine ngce = new NGramCompareEngine(index, output, new File(db), distMetric);
			if(clustering.equals("none"))
				compareTime = ngce.run(parallelType, parallelism);
			else
				clusterTime = ngce.cluster(clustering, nClustOrMinPts, nCloseOrEps, clustersLoad,
parallelType, parallelism);

			// Write new or updated indexes
			index.write();

			if(compareTime >= 0.0)
				System.err.println("Compare time: " + compareTime);
			if(clusterTime >= 0.0)
				System.err.println("Cluster time: " + clusterTime);
	}
}
```

12.1.2 NGramCompareEngine

```
/**
 * Compares sets of VB forms, calculating distance based on n-gram counts and
 * a given distance measure. Outputs most similar pairs of documents into a
 * results file.
 *
 * @author Jeremy Iverson
 * @author Tony Ohmann
 *
 */
public class NGramCompareEngine {

	// The number of reported most similar pairs
	private static int NUM_REPORTED_PAIRS = 300;

	private static boolean OUTPUT_LIKE_MOSS = false;
	private Matrix matrix;
	private File databaseDir;
	private Index index;
	private ArrayList<CompareForm> submissionForms;
	private ArrayList<CompareForm> databaseForms;
	private int numSubmissionForms;
	private int numDatabaseForms;
	private int distMetric;
	private String outputPath;
```

```
	/**
	 * Sets up matrix and lists of submission and database forms
	 *
	 * @param index_ The file containing indexed n-grams
	 * @param outputDir_ The directory where ngramized submission forms are stored (IPPDC's output dir)
	 * @param databaseDir_ The directory where ngramized database forms are stored
	 * @param distMetric_ The distance metric with which to compare forms
	 */
	public NGramCompareEngine(Index index_, File outputDir_, File databaseDir_, int distMetric_) {
		index = index_;
		outputPath = outputDir_.getPath();
		submissionForms = new ArrayList<CompareForm>();
		databaseForms = new ArrayList<CompareForm>();
		distMetric = distMetric_;

		databaseDir = databaseDir_;
		File ngramizedSource = new File(outputDir_.getPath() + "/ngramized");

		// Filter that only accepts files
		FileFilter fileFilter = new FileFilter() {
			public boolean accept(File file) {
				return !file.isDirectory();
			}
		};

		if(!databaseDir.exists())
			databaseDir.mkdir();

		// Get arrays of submission and database forms
		File[] submissionFiles = ngramizedSource.listFiles(fileFilter);
		File[] databaseFiles = databaseDir.listFiles();

		numSubmissionForms = submissionFiles.length;
		numDatabaseForms = databaseFiles.length;

		// Matrix to hold distances between forms
		matrix = new Matrix(numSubmissionForms + numDatabaseForms, numSubmissionForms);

		File dir = new
File(ngramizedSource.getPath()+"/"+DateUtils.now("yyyy.MMMMM.dd'_'HH:mm:ss")+"_Finished");
		if(!dir.exists())
			dir.mkdir();

		// Initialize all submission forms to be compared
		for(File submissionForm: submissionFiles) {
			// Turn each ngramized file into a CompareForm
			CompareForm s = new CompareForm(submissionForm, index, distMetric);
			submissionForms.add(s);

			// Put the clean project in the directory created above to
			submissionForm.renameTo(new File(dir.getPath()+"/"+submissionForm.getName()));
		}

		// Initialize all database forms to be compared
		for(File databaseForm: databaseFiles) {
			// Turn each ngramized database form into a CompareForm
			CompareForm d = new CompareForm(databaseForm, index, distMetric);
			databaseForms.add(d);
		}

		// Write each submission form to an ngramized form in the database
		for(CompareForm c: submissionForms)
			c.toCompareFormFile(databaseDir);

		// Free memory
		submissionFiles = null;
		databaseFiles = null;
	}



	/**
	 * Cluster using the specified algorithm (or load previously-saved
	 * clustering) and then perform pairwise comparisons (in parallel if
	 * specified).
	 *
	 * @param clustering_ Which clustering algorithm to use: {kmedoids, dbscan,
	 * none}
	 * @param clusterParam0_ If kmedoids, numClusters. If dbscan, minPts
	 * @param clusterParam1_ If kmedoids, numClosestClusters. If dbscan, eps
	 * @param clustersLoad_ Whether to load a previously-saved clustering
```

```
	 * @param parallelType_ Parallel comparison type in {none, forkjoinloop}
	 * @param parallelism_ Level of parallelism (if parallelType_ != none)
	 *
	 * @return Total clustering runtime
	 */
	@SuppressWarnings("unchecked")
	public double cluster(String clustering_, int clusterParam0_, double clusterParam1_, boolean 
clustersLoad_, String parallelType_, int parallelism_) {
		int numClusters=0, minPts=0, numClosestClusters=0;
		double eps=0.0;
		double clusterTime = -1.0;

		ClusteringAlgorithm km=null, dbscan=null;
		ClusteringAlgorithm clusterer;

		GregorianCalendar t0 = new GregorianCalendar();
		GregorianCalendar t1 = null;

		if(clustering_.equals("kmedoids")) {

			numClusters = clusterParam0_;
			numClosestClusters = (int)Math.floor(clusterParam1_+0.5d);

			// Perform clustering and save
			if(!clustersLoad_) {
				km = new KMedoids(databaseForms, outputPath);
				km.cluster(numClusters);
				((KMedoids)km).saveClustering(outputPath + "/km" + numClusters + ".bin");
			}

			// Load clustering
			else {
				km = KMedoids.loadClustering(outputPath + "/km" + numClusters + ".bin");
			}

			t1 = new GregorianCalendar();

			// Get and print statistics on clustering
			double wss = ((KMedoids)km).getWSS();
		 double bss = ((KMedoids)km).getBSS();
		 System.out.println("k= "+numClusters+"   WSS= "+wss+"   BSS= "+bss+"   BSS/WSS= "+bss/wss);

			clusterer = km;

		} else if(clustering_.equals("dbscan")) {

			minPts = clusterParam0_;
			eps = clusterParam1_;
			numClosestClusters = 1;

			// Perform clustering
			dbscan = new DBSCAN(databaseForms);
			dbscan.cluster(eps, minPts);

			t1 = new GregorianCalendar();

			// Get and print statistics on clustering
			numClusters = ((DBSCAN)dbscan).getNumberOfClusters();
			System.out.println("minPts= "+minPts+"   eps= "+eps+"   #clusters= "+numClusters);

			clusterer = dbscan;

		} else
			return -1.0;

		clusterTime = (t1.getTimeInMillis()-t0.getTimeInMillis())*1.0 / 1000;
	System.out.println("Clustering time: " + clusterTime + " sec\n");

		// Submission form names for matrix column
		ArrayList<String> fullCol = new ArrayList<String>();
		for(CompareForm s : submissionForms)
			fullCol.add(s.getName());

		// Database and submission form names for matrix row
		ArrayList<String> fullRow = new ArrayList<String>();
		for(CompareForm d : databaseForms)
			fullRow.add(d.getName());
		for(CompareForm s : submissionForms)
			fullRow.add(s.getName());

		// Set up level of parallelism
```

```
		int parallelism = Runtime.getRuntime().availableProcessors();
		if(parallelism_ > 1)
			parallelism = parallelism_;

		// Set up final results file
		PrintWriter out = null;
		try {
			if(clustering_.equals("kmedoids"))
				out = new PrintWriter(new FileWriter(new File(outputPath +
"/results_kmedoids_" + numClusters + "_" + numClosestClusters + ".txt")));
			else if(clustering_.equals("dbscan"))
				out = new PrintWriter(new FileWriter(new File(outputPath + "/results_dbscan_"
+ new DecimalFormat("#.######").format(eps) + "_" + minPts + "_" + numClosestClusters + ".txt")));
		} catch (IOException e) {
			e.printStackTrace();
		}

		// Compare each submission form to closest clusters
		matrix = new Matrix(numSubmissionForms + numDatabaseForms, numSubmissionForms);
		matrix.setCol((ArrayList<String>)fullCol.clone());
		matrix.setRow((ArrayList<String>)fullRow.clone());

		GregorianCalendar t2 = new GregorianCalendar();

		// PARALLEL compare
		if(parallelType_.equals("forkjoinloop"))
			clusterCompareParallel(clusterer, numClosestClusters, fullRow, parallelism);

		// SEQUENTIAL compare
		else if(parallelType_.equals("none"))
			clusterCompareNone(clusterer, numClosestClusters);

		GregorianCalendar t3 = new GregorianCalendar();

		double compareTime = (t3.getTimeInMillis()-t2.getTimeInMillis())*1.0 / 1000;

		// Print document pairs found to be most similar
		out.print(compareTime + "\n" + getHighestPairs(NUM_REPORTED_PAIRS));
		out.close();

		return clusterTime;
	}



	/**
	 * Perform pairwise comparisons in parallel using the FJTask framework
	 * without recursion given a complete clustering
	 *
	 * @param clusterer_ Clustering algorithm instance
	 * @param numClosestClusters_ Number of closest clusters against which to
	 * compare each submission form
	 * @param row_ Row header for matrix, database and submission form names
	 * @param parallelism_ Level of parallelism
	 */
	private void clusterCompareParallel(ClusteringAlgorithm clusterer_, int numClosestClusters_,
ArrayList<String> row_, int parallelism_) {

		int parallelism = parallelism_;

		// Numbers for calculating tasks' submission document offsets
		int docsPerTask = (int) Math.floor((numSubmissionForms + 0.0) / (parallelism + 0.0));
        int remainingDocs = numSubmissionForms - (docsPerTask * parallelism);
        int[] taskOffset = new int[parallelism+1];   // "end" offset simplifies code

        // Calculate the document index offset for each task
        taskOffset[0] = 0;
        for(int i = 1; i < parallelism+1; ++i)
            taskOffset[i] = taskOffset[i-1] + docsPerTask + (remainingDocs-- > 0 ? 1 : 0);

		// Fork-join task variables and task pool
        RecursiveTask<Matrix> task;
        ArrayList<RecursiveTask<Matrix>> tasks = new ArrayList<RecursiveTask<Matrix>>(parallelism);
		ForkJoinPool pool = new ForkJoinPool(parallelism);

		// For each submission form, a list of the forms in the clusters closest to it
		ArrayList<ArrayList<CompareForm>> closeFormsLists = new ArrayList<ArrayList<CompareForm>>();

		// Get close cluster(s) for each submission form
	for(CompareForm s : submissionForms)
			closeFormsLists.add( clusterer_.findClosestClusters(s, numClosestClusters_) );
```

```
		// Create tasks and queue them
	for(int i=0; i<parallelism; ++i) {
		task = new ClusterTask(taskOffset[i], taskOffset[i+1]-1, submissionForms, closeFormsLists,
distMetric, row_);
		tasks.add(task);
		pool.execute(task);
	}

	int resultOffset;
	int nextOffset;
	Double srcArr[][];
	Double destArr[][] = new Double[numSubmissionForms][numDatabaseForms+numSubmissionForms];
	Double finalArr[][] = new Double[numDatabaseForms+numSubmissionForms][numSubmissionForms];
	Matrix resultMatrix = null;
	RecursiveTask<Matrix> resultTask;
	boolean containsNonNull;

	// Combine result matrices from all tasks
	while(true) {
		containsNonNull = false;
		for(int i = 0; i < parallelism; ++i) {

			// Get task and check its status
			resultTask = tasks.get(i);
			if(resultTask != null)			// task is finished
				containsNonNull = true;
			else							// task is already
processed
				continue;
			if(!resultTask.isDone())		// task is still running
				continue;

			// Get Matrix returned by task
			resultMatrix = resultTask.join();

			// Indicate that this task has already been processed
			tasks.set(i, null);

			// Get beginning and ending form offsets for this task
			resultOffset = taskOffset[i];
			nextOffset = taskOffset[i+1];

			// Copy results from this task into our results array
			srcArr = resultMatrix.getMatrixArray();
			for(int j=resultOffset; j<nextOffset; ++j)
				destArr[j] = srcArr[j-resultOffset];
		}

		// Check if all tasks have been processed
		if(!containsNonNull)
			break;

		// Prevent excessive CPU grinding, especially early on when tasks are still processing
	   try{ Thread.sleep(10); } catch(Exception e) {}
	}

	// Transpose destArr: tasks had rows and cols switched for efficiency
	for(int i=0; i<numSubmissionForms; ++i)
		for(int j=0; j<numDatabaseForms+numSubmissionForms; ++j)
			finalArr[j][i] = destArr[i][j];

	// Set constructed distance array for the Matrix
	matrix.setMatrixArray(finalArr);
	}



	/**
	 * Perform pairwise comparisons sequentially given a complete clustering
	 *
	 * @param clusterer_ Clustering algorithm instance
	 * @param numClosestClusters_ Number of closest clusters against which to
	 * compare each submission form
	 */
	private void clusterCompareNone(ClusteringAlgorithm clusterer_, int numClosestClusters_) {

		// The forms contained within the clusters closest to the current submission form
		ArrayList<CompareForm> closeForms = null;

		for(CompareForm s: submissionForms) {
```

```
                closeForms = clusterer_.findClosestClusters(s, numClosestClusters_);

                //Compare each submission form to closest-cluster forms
                for(CompareForm d: closeForms)
                        matrix.add(s.getName(), d.getName(), s.compare(d));

                //Compare each submission form to all other submission forms including self
                for(CompareForm d: submissionForms)
                        matrix.add(s.getName(), d.getName(), s.compare(d));
        }
 }



 /**
  * Perform pairwise comparisons without clustering
  *
  * @param parallelType_ Parallel comparison type in {none, forkjoinloop}
  * @param parallelism_ Level of parallelism (if parallelType_ != none)
  *
  * @return Total comparison runtime
  */
 public double run(String parallelType_, int parallelism_) {
        double runtime = -1.0;

        GregorianCalendar t0 = new GregorianCalendar();

        if(parallelType_.equals("forkjoinloop"))
                compareForkJoinLoop(parallelism_);

        else if(parallelType_.equals("none"))
                compareNone();

        GregorianCalendar t1 = new GregorianCalendar();
        runtime = (t1.getTimeInMillis()-t0.getTimeInMillis())*1.0 / 1000;


        // Set up final results file
        PrintWriter out = null;
        try {
                out = new PrintWriter(new FileWriter(new File(outputPath + "/results.txt")));
        } catch (IOException e) {
                e.printStackTrace();
        }

        // Print document pairs found to be most similar
        out.print( getHighestPairs(NUM_REPORTED_PAIRS) );
        out.close();

        return runtime;
 }



 /**
  * Perform pairwise comparisons in parallel using the FJTask framework
  * without recursion and without clustering
  *
  * @param parallelism_ Level of parallelism
  */
 private void compareForkJoinLoop(int parallelism_) {
        int parallelism = parallelism_;

// Create a combined list of submission and db forms
ArrayList<CompareForm> allDocs = new ArrayList<CompareForm>();
allDocs.addAll(databaseForms);
allDocs.addAll(submissionForms);

// Numbers for calculating tasks' document offsets
int numDocs = allDocs.size();
int docsPerTask = (int) Math.floor((numDocs + 0.0) / (parallelism + 0.0));
int remainingDocs = numDocs - (docsPerTask * parallelism);
int[] taskOffset = new int[parallelism+1];   // "end" offset simplifies code

// Calculate the document index offset for each task
taskOffset[0] = 0;
for(int i = 1; i < parallelism+1; ++i)
    taskOffset[i] = taskOffset[i-1] + docsPerTask + (remainingDocs-- > 0 ? 1 : 0);

// Fork-join task variables and task pool
RecursiveTask<Matrix> task;
```

```
ArrayList<RecursiveTask<Matrix>> tasks = new ArrayList<RecursiveTask<Matrix>>(parallelism);
        ForkJoinPool pool = new ForkJoinPool(parallelism);

        // The sublist of allForms to be passed to each fork-join task
ArrayList<CompareForm> sublist;

// Create sublists, create tasks, and queue tasks
  for(int i = 0; i < parallelism; ++i) {
        sublist = new ArrayList<CompareForm>();
        sublist.addAll(allDocs.subList(taskOffset[i], taskOffset[i+1]));
        task = new FormDataLoopTask(submissionForms, sublist, distMetric);
        tasks.add(task);
        pool.execute(task);
  }

  // Pre-allocate Matrix row header containing names of allDocs
  ArrayList<String> row = new ArrayList<String>(numDocs);
  for(int i=0; i<numDocs; ++i)
        row.add(null);

  ArrayList<String> resultRow;
  int resultOffset;
  int nextOffset;
  Double srcArr[][];
  Double destArr[][] = new Double[allDocs.size()][numSubmissionForms];
  Matrix resultMatrix = null;
  RecursiveTask<Matrix> resultTask;
  boolean containsNonNull;

  // Combine result matrices from all tasks
  while(true) {
        containsNonNull = false;
        for(int i = 0; i < parallelism; ++i) {

                // Get task and check its status
                resultTask = tasks.get(i);
                if(resultTask != null)                  // task is finished
                        containsNonNull = true;
                else                                    // task is already processed
                        continue;
                if(!resultTask.isDone())                // task is still running
                        continue;

                // Get Matrix returned by task
                resultMatrix = resultTask.join();

                // Indicate that this task has already been processed
                tasks.set(i, null);

                // Get beginning and ending form offsets for this task
                resultOffset = taskOffset[i];
                nextOffset = taskOffset[i+1];

                // Copy this task's row header into the appropriate place in the final row header
                resultRow = resultMatrix.getRow();
                for(int j=resultOffset; j<nextOffset; ++j)
                        row.set(j, resultRow.get(j-resultOffset));

                // Copy results from this task into our final results array
                srcArr = resultMatrix.getMatrixArray();
                for(int j=resultOffset; j<nextOffset; ++j)
                        destArr[j] = srcArr[j-resultOffset];
        }

        // Check if all tasks have been processed
        if(!containsNonNull)
                break;

        // Prevent excessive CPU grinding, especially early on when tasks are still processing
    try{ Thread.sleep(10); } catch(Exception e) {}
  }

  // Row is now constructed; each matrix uses the same row headers
  matrix.setRow(row);

  // Each task's col is the same; each matrix's col is the same
  ArrayList<String> newCol = resultMatrix.getCol();
  matrix.setCol(newCol);

  // Set constructed similarity array for each matrix
  matrix.setMatrixArray(destArr);
```

```
	}



	/**
	 * Perform pairwise comparisons sequentially and without clustering
	 */
	private void compareNone() {
		for(CompareForm s: submissionForms) {

			// Compare each submission form to all forms in database
			for(CompareForm d: databaseForms)
				matrix.add(s.getName(), d.getName(), s.compare(d));

			// Compare each submission form to all other submission forms including self
			for(CompareForm d: submissionForms)
				matrix.add(s.getName(), d.getName(), s.compare(d));
		}
	}



	/**
	 * Sort and output the most similar pairs of compared documents
	 *
	 * @param numPairs_ The number of reported pairs
	 *
	 * @return The pairs of most similar documents in ascending distance order
	 */
	private String getHighestPairs(int numPairs_) {
		ArrayList<DocumentPair> mostSimilarPairs = new ArrayList<DocumentPair>();
		double distance;
		int numAllForms = numSubmissionForms + numDatabaseForms;

		// The distances between and names of compared documents
		Double[][] distArr = matrix.getMatrixArray();
		ArrayList<String> allNames = matrix.getRow();

		// Populate list of document pairs
		for(int i=0; i<numSubmissionForms; ++i) {
			for(int j=0; j<numDatabaseForms; ++j) {
				distance = distArr[j][i];

				mostSimilarPairs.add(new DocumentPair(i, j, distance));
			}

			for(int j=numDatabaseForms+i+1; j<numAllForms; ++j) {
				distance = distArr[j][i];

				mostSimilarPairs.add(new DocumentPair(i, j, distance));
			}
		}

		// Sort document pairs in ascending distance order
		Collections.sort(mostSimilarPairs);

		StringBuffer sb = new StringBuffer();
		DocumentPair sp;

		// MOSS-like output: very odd but can be processed exactly like MOSS' output
		if(OUTPUT_LIKE_MOSS)
			for(int i=0; i<numPairs_; ++i) {
				sp = mostSimilarPairs.get(i);

	sb.append("\"all/").append(allNames.get(sp.left+numDatabaseForms).substring(9).replace("_",
"/")).append(" \",\"all/").append(allNames.get(sp.right).substring(9).replace("_", "/")).append(" \"
").append(sp.distance).append("\n");
			}

		// Normal output (each line: firstDoc secondDoc distance)
		else
			for(int i=0; i<numPairs_; ++i) {
				sp = mostSimilarPairs.get(i);
				sb.append(allNames.get(sp.left+numDatabaseForms)).append("
").append(allNames.get(sp.right)).append(" ").append(sp.distance).append("\n");
			}

		return sb.toString();
	}
```

```
	/**
	 * A sortable pair of compared documents
	 */
	private class DocumentPair implements Comparable<DocumentPair> {
		int left;
		int right;
		double distance;

		/**
		 * Create and populate a document pair
		 *
		 * @param left_ Left or first document's index
		 * @param right_ Right or second document's index
		 * @param distance_ Distance between the documents
		 */
		public DocumentPair(int left_, int right_, double distance_) {
			left = left_;
			right = right_;
			distance = distance_;
		}

		/**
		 * Prints the two documents' indexes and the distance between them
		 *
		 * @return Document indexes and distance, space-separated
		 */
		public String toString() {
			return left + " " + right + " " + distance;
		}

		/**
		 * Compare this DocumentPair to another according to distance
		 *
		 * @param other DocumentPair to compare this pair to
		 *
		 * @return Comparison result
		 */
		@Override
		public int compareTo(DocumentPair other) {
			if(distance > other.distance)
				return 1;
			else if(distance == other.distance)
				return 0;
			else
				return -1;
		}
	}
}
```

12.1.3 CompareForm

```
/**
 * This class represents a document containg n-grams that can be compared to other
 * CompareForms
 *
 * @author Jeremy Iverson
 * @author Tony Ohmann
 * @author Fei Wu
 */
public class CompareForm implements Serializable {

	private static final long serialVersionUID = 8909789759850766837L;

	private String name;
	private File document;

	/**
	 * N-grams associated with their frequencies
	 */
	private HashMap<String, Integer> ngrams;

	private Index index;
	private int distMetric;
```

```
/**
 * For efficiently creating duplicate CompareForms
 *
 * @param name_ Name of document
 * @param index_ Associated index instance
 * @param ngrams_ N-grams with frequencies
 * @param distMetric_ Distance metric
 */
public CompareForm(String name_, Index index_, HashMap<String, Integer> ngrams_, int distMetric_) {
       name = name_;
       index = index_;
       ngrams = ngrams_;
       distMetric = distMetric_;
}



/**
 * Creates a new CompareForm and initializes n-grams and frequencies
 *
 * @param document_ The file from which to create the CompareForm
 * @param index_ Associated n-gram index
 * @param distMetric_ Distance metric
 */
public CompareForm(File document_, Index index_, int distMetric_) {
       document = document_;
       name = document_.getName();
       index = index_;
       distMetric = distMetric_;
       initializeNGrams();
}



/**
 * Get the db filename of the document without the path
 *
 * @return the filename of the document without the path
 */
public String getName() {
       return name;
}



/**
 * Get the n-gram index
 *
 * @return N-gram index
 */
public Index getIndex() {
       return index;
}



/**
 * Get the document's project name
 *
 * @return The document's project name
 */
public String getProjectName() {
       int start = name.indexOf("_") + 1;
       return name.substring(start, name.indexOf("_", start+1));
}



/**
 * Get the document's original filename
 *
 * @return The document's original filename
 */
public String getFormName() {
       int start = name.indexOf("_", name.indexOf("_", name.indexOf("_")+1)+1) + 1;
       return name.substring(start);
}



/**
```

```
	 * Parse a file and store each of its n-grams and their frequencies
	 */
	public void initializeNGrams() {
		ngrams = new HashMap<String, Integer>();
		int value;

		try {
			Scanner in = new Scanner(document);

			// Traverse the file line by line
			while(in.hasNextLine()) {
				String line = in.nextLine();
				if(!line.startsWith("#")) {								//
Ignore metadata
					value = Integer.parseInt(in.nextLine());  // Get the frequency
					ngrams.put(line, value);
				}
			}
			in.close();

		} catch (FileNotFoundException e) {
			e.printStackTrace();
		}
	}



	/**
	 * Get the total number of n-grams in a document
	 *
	 * @return The total number of n-grams in a document
	 */
	public int totalGrams() {
		int total=0;
		for(String gram: ngrams.keySet())
			total += ngrams.get(gram);
		return total;
	}



	/**
	 * Get the n-grams associated with this document
	 *
	 * @return The n-grams associated with this document
	 */
	public HashMap<String, Integer> getGrams() {
		return ngrams;
	}



	/**
	 * Compute the distance between this document, p, and another, q
	 *
	 * @param qForm_ The document to compare this document to
	 *
	 * @return The calculated distance
	 */
	public double compare(CompareForm qForm_) {
		// The other document's n-grams
		HashMap<String, Integer> qGramsOrig = qForm_.getGrams();

		// A mutable copy of the other document's n-grams
		HashMap<String, Integer> qTable = new HashMap<String, Integer>(qGramsOrig);

		// The vectors representing documents p and q
		int maxNumForms = qTable.size() + ngrams.size();
		double[] p = new double[maxNumForms];
		double[] q = new double[maxNumForms];
		int size = 0;

		// Build the union set of the n-grams contained within p and q, and
		//     assign 0 frequencies to n-grams not appearing in one document.
		// Also perform TF*IDF weighting on the frequencies
		for(String gram: ngrams.keySet()) {
			if(qGramsOrig.containsKey(gram)) {
				p[size] = ngrams.get(gram) * index.idf(gram);
				q[size++] = qGramsOrig.get(gram) * index.idf(gram);
				qTable.remove(gram);
			}
```

```
				else {
					p[size] = ngrams.get(gram) * index.idf(gram);
					q[size++] = 0.0;
				}
			}

			for(String gram: qTable.keySet()) {
				q[size] = qGramsOrig.get(gram) * index.idf(gram);
				p[size++] = 0.0;
			}

			// Calculate the distance using the requested distance metric
			switch(distMetric) {
			case 0:
				return cosine(p, q, size);
			case 1:
				return manhattan(p, q, size);
			case 2:
				return euclidean(p, q, size);
			default:
				return -999.9;
			}
		}



		/**
		 * Calculate distance using the COSINE distance metric
		 *
		 * @param pList_ List of n-gram weights for the first document
		 * @param qList_ List of n-gram weights for the second document
		 * @param size_ The number of unique n-grams between both documents
		 *
		 * @return The calculated distance
		 */
		public double cosine(double[] pList_, double[] qList_, int size_) {
			double x = 0;
			double y = 0;
			double numerator = 0;
			double denominator = 0;
			for(int i=0; i<size_; ++i) {
				numerator += pList_[i] * qList_[i];
				x += pList_[i] * pList_[i];
				y += qList_[i] * qList_[i];
			}
			denominator = Math.sqrt(x * y);

			// Subtract from 1 to convert Cosine similarity into a distance
			Double result = 1.0 - numerator / denominator;
			if(result.isInfinite() || result.isNaN())
				result = Double.POSITIVE_INFINITY;
			return result;
		}



		/**
		 * Calculate distance using the MANHATTAN distance metric
		 *
		 * @param pList_ List of n-gram weights for the first document
		 * @param qList_ List of n-gram weights for the second document
		 * @param size_ The number of unique n-grams between both documents
		 *
		 * @return The calculated distance
		 */
		public double manhattan(double[] pList_, double[] qList_, int size_) {
			double p_num;
			double q_num;
			double summation = 0.0;
			for(int i=0; i<size_; ++i) {
				p_num = pList_[i];
				q_num = qList_[i];

				if(p_num >= q_num)
					summation += p_num - q_num;
				else
					summation += q_num - p_num;
			}

			Double result = summation / size_;
			if(result.isInfinite() || result.isNaN())
```

```
                        result = Double.POSITIVE_INFINITY;
                return result;
        }



        /**
         * Calculate distance using the EUCLIDEAN distance metric
         *
         * @param pList_ List of n-gram weights for the first document
         * @param qList_ List of n-gram weights for the second document
         * @param size_ The number of unique n-grams between both documents
         *
         * @return The calculated distance
         */
        public double euclidean(double[] pList_, double[] qList_, int size_) {
                double difference;
                double summation = 0.0;
                for(int i=0; i<size_; ++i) {
                        difference = pList_[i] - qList_[i];
                        summation += difference * difference;
                }

                summation = Math.sqrt(summation);

                Double result = summation / size_;
                if(result.isInfinite() || result.isNaN())
                        result = Double.POSITIVE_INFINITY;
                return result;
        }



        /**
         * Print the n-grams from a document into a file
         *
         * @param databaseDir_ The destination database directory
         */
        public void toCompareFormFile(File databaseDir_) {
                PrintWriter writer = null;
                File outFile = new File(databaseDir_.getPath() + "/" + name);

                try {
                        // Dont re-output a document into the database
                        if(!outFile.exists()) {
                                writer = new PrintWriter(new FileWriter(outFile,true),true);
                                writer.print("#*" + name + "\n" + toString());
                                writer.close();
                        }
                } catch(Exception e) {
                        e.printStackTrace();
                }
        }



        /**
         * Return string representation of n-grams in this document
         */
        public String toString() {
                StringBuffer sb = new StringBuffer();
                for(String key : ngrams.keySet())
                        sb.append(key).append("\n").append(ngrams.get(key)).append("\n");
                return sb.toString();
        }
}
```

12.1.4 FormDataLoopTask (Loop-based FJTask parallel strategy)

```
/**
 * Fork-join task that compares all submission forms to its sublist of
 * submission+database forms (allForms) using loops
 *
 * @author Tony Ohmann
 * @author Fei Wu
 */
public class FormDataLoopTask extends RecursiveTask<Matrix> {
```

```
	private ArrayList<CompareForm> submissionForms, myForms;
	private Matrix matrix;


	/**
	 * Intialize lists of forms and result matrix
	 *
	 * @param submissionForms_ All submission forms
	 * @param allForms_ This fork-join task's sublist of submission forms plus
	 * database forms
	 * @param distMetric_ The distance metric
	 */
	public FormDataLoopTask(ArrayList<CompareForm> submissionForms_, ArrayList<CompareForm> allForms_,
int distMetric_) {
		HashMap<String, Integer> oldNgrams;
		HashMap<String, Integer> newNgrams;

		// NUMA optimization: make a local-memory copy of the index
		Index oldIndex = submissionForms_.get(0).getIndex();
		HashMap<String, Integer> oldCountTable = oldIndex.countTable;
		HashMap<String, Integer> newCountTable = new HashMap<String, Integer>();
		for(String s : oldCountTable.keySet())
			newCountTable.put(new String(s), new Integer(oldCountTable.get(s)));
		Index newIndex = new Index(oldIndex.totalDocuments, newCountTable);

		// NUMA optimization: make a local-memory copy of all submission forms
		submissionForms = new ArrayList<CompareForm>();
		for(CompareForm cf : submissionForms_) {
			oldNgrams = cf.getGrams();
			newNgrams = new HashMap<String, Integer>();
			for(String s : oldNgrams.keySet())
				newNgrams.put(new String(s), new Integer(oldNgrams.get(s)));
			submissionForms.add(new CompareForm(new String(cf.getName()), newIndex, newNgrams,
distMetric_));
		}

		myForms = allForms_;

		matrix = new Matrix(myForms.size(), submissionForms.size());
	}


	/**
	 * Compare all submission forms to this fork-join task's sublist of
	 * submission plus database forms
	 *
	 * @return The matrix containing distance measures between the compared
	 * forms
	 */
	protected Matrix compute() {

		// Compare each submission form to this task's sublist of all forms
		for(CompareForm s: submissionForms)
			for(CompareForm d: myForms)
				matrix.add(s.getName(), d.getName(), s.compare(d));

		return matrix;
	}
}
```

12.1.5 ClusteringAlgorithm

```
/**
 * Interface for a clustering algorithm implementation
 *
 * @author Tony Ohmann
 */
public interface ClusteringAlgorithm {

	/**
	 * Cluster documents into numClusters clusters using PAM
	 *
	 * @param numClusters_ Number of clusters to create
	 */
```

```
	public void cluster(int numClusters_);

	/**
	 * Cluster documents using DBSCAN
	 *
	 * @param epsilon_ Distance for deciding on core and border points
	 * @param minPoints_ Minimum points threshold to label a point as core
	 * (must have minPts points within epsilon)
	 */
	public void cluster(double epsilon_, int minPoints_);

	/**
	 * Find the closest C clusters to a submitted document and return the
	 * documents contained within those clusters in one list
	 *
	 * @param form_ The submitted document to find clusters close to
	 * @param numClosestClusters_ The number of closest clusters to find, or C
	 *
	 * @return The documents contained within the C close clusters
	 */
	public ArrayList<CompareForm> findClosestClusters(CompareForm form_, int numClosestClusters_);
}
```

12.1.6 KMedoids (PAM implementation)

```
/**
 * An implementation of PAM to cluster documents into globular clusters
 * represented by medoids where a point is a database document
 *
 * @author Tony Ohmann
 * @author Chris Roering
 * @author Laura Nierengarten
 */
public class KMedoids implements ClusteringAlgorithm, Serializable {

	private static final long serialVersionUID = 3327354335290624034L;

	private CompareForm[] forms;
	private int[] formToMedoid;
	private double[][] distances;
	private int numForms;
	private Random rand;
	private int[] medoids;
	private ArrayList<ArrayList<CompareForm>> clusters;
	private int numClusters;
	private int medoidOfDataset = -1;



	/**
	 * Private constructor for use in loadClustering()
	 */
	private KMedoids() { }



	/**
	 * Read in documents and compute their distance matrix
	 *
	 * @param forms_ The database documents to read in and later cluster
	 * @param outputPath_ The output directory path
	 */
	public KMedoids(ArrayList<CompareForm> forms_, String outputPath_) {
		numForms = forms_.size();

		// Read in and store the documents
		forms = new CompareForm[numForms];
		for(int formInd=0; formInd<numForms; ++formInd)
			forms[formInd] = forms_.get(formInd);

		formToMedoid = new int[numForms];
		rand = new Random();
		File distFile = new File(outputPath_ + "/km_distances.bin");

		// Compute distance matrix if it doesn't exist
		if(!distFile.exists()) {
			computeDistanceMatrix(distFile);
```

```
		// Load it from file if it does
		} else
			try {
				ObjectInputStream ois = new ObjectInputStream(new FileInputStream(distFile));
				distances = (double[][]) ois.readObject();
				ois.close();

				// Verify that the distance matrix file isn't corrupt
				if(distances.length != numForms || distances[0].length != numForms)
					computeDistanceMatrix(distFile);
			} catch (FileNotFoundException e) {
				e.printStackTrace();
			} catch (IOException e) {
				e.printStackTrace();
			} catch (ClassNotFoundException e) {
				computeDistanceMatrix(distFile);
			}

		// Compute medoid of entire dataset
		cluster(1);
		medoidOfDataset = medoids[0];
	}



	/**
	 * Compute the distance matrix and save it to disk
	 *
	 * @param distFile_ The file to save to
	 */
	private void computeDistanceMatrix(File distFile_) {
		distances = new double[numForms][numForms];

		// Compute distance matrix values
		for(int i=0; i<numForms; ++i)
			for(int j=i; j<numForms; ++j)
				distances[i][j] = distances[j][i] = forms[i].compare(forms[j]);

		// Save distance matrix
		try {
			ObjectOutputStream output = new ObjectOutputStream(new FileOutputStream(distFile_));
			output.writeObject(distances);
			output.close();

		} catch (IOException e) {
			e.printStackTrace();
		}
	}



	/**
	 * Unimplemented version of cluster(). KMedoids uses cluster(int)
	 */
	public void cluster(double epsilon_, int minPoints_) {
		throw new IllegalArgumentException("Wrong number of params. KMedoids uses cluster(int).");
	}



	/**
	 * Cluster documents into numClusters clusters using PAM
	 *
	 * @param numClusters_ Number of clusters to create
	 */
	public void cluster(int numClusters_) {
		numClusters = numClusters_;
		medoids = new int[numClusters];

		int potentialMedoid;

		// Randomly choose k initial medoids
		ChooseInitial:
		for(int numMedoids=0; numMedoids<numClusters; ) {
			potentialMedoid = rand.nextInt(numForms);

			// Verify that this point hasn't already been chosen as an initial
			// medoid
			for(int i=0; i<numMedoids; ++i)
				if(medoids[i] == potentialMedoid)
```

```
                                        continue ChooseInitial;

                                medoids[numMedoids] = potentialMedoid;
                                numMedoids++;
                        }

                        boolean medoidsChanged = true;
                        Double bestDist;
                        double thisDist=-1.0;
                        int bestMedoidIndex;

                        // Loop until medoids don't change or until 10 iterations have passed
                        for(int iteration=0; medoidsChanged && (iteration<10 || iteration<numClusters*2); ++iteration)
{

                                // Assign each non-medoid form to its closest medoid
                                AssignToMedoid:
                                for(int formInd=0; formInd<numForms; ++formInd) {

                                        // Check if the form at formInd is already a medoid
                                        for(int medoidInd=0; medoidInd<numClusters; ++medoidInd)
                                                if(medoids[medoidInd] == formInd) {
                                                        formToMedoid[formInd] = medoidInd;
                                                        continue AssignToMedoid;
                                                }

                                        bestDist = Double.POSITIVE_INFINITY;
                                        bestMedoidIndex = -888;

                                        // Find the closest medoid
                                        for(int medoidInd=0; medoidInd<numClusters; ++medoidInd) {
                                                thisDist = distances[formInd][medoids[medoidInd]];
                                                if(thisDist < bestDist) {
                                                        bestDist = thisDist;
                                                        bestMedoidIndex = medoidInd;
                                                }
                                        }

                                        // Store the index of the closest medoid
                                        formToMedoid[formInd] = bestMedoidIndex;
                                }

                                double piToTau,
                                           muToTau,
                                           medoidToTau,
                                           cost,
                                           bestCost = 0.0;
                                int closestMedoidInd,
                                        bestPiInd = -999,
                                        bestMuInd = -999;

                                // Decide if any point mu is a better medoid than current medoid pi
                                //     by comparing the cost of the switch for each other point tau
                                for(int piInd=0; piInd<numClusters; ++piInd) {                     //  pi: current
medoid

                                        MuLoop:
                                        for(int muInd=0; muInd<numForms; ++muInd) {                         //  mu:
new potential medoid

                                                // Verify mu isn't already a medoid
                                                for(int i=0; i<numClusters; ++i)
                                                        if(medoids[i] == muInd)
                                                                continue MuLoop;
                                                cost = 0.0;

                                                TauLoop:
                                                for(int tauInd=0; tauInd<numForms; ++tauInd) {     // tau: other non-
medoid point

                                                        // Verify tau is neither mu nor a medoid
                                                        if(tauInd==muInd)
                                                                continue;
                                                        for(int i=0; i<numClusters; ++i)
                                                                if(medoids[i] == tauInd)
                                                                        continue TauLoop;

                                                        // Retrieve distances that will be required
                                                        piToTau = distances[medoids[piInd]][tauInd];
                                                        muToTau = distances[muInd][tauInd];
                                                        closestMedoidInd = formToMedoid[tauInd];
```

```
								// Pi isn't tau's current closest medoid...
								if(closestMedoidInd != piInd) {
									medoidToTau =
distances[medoids[closestMedoidInd]][tauInd];

									// ...so cost only changes if mu would become tau's
									// new closest medoid
									if(medoidToTau < piToTau)
										if(medoidToTau < muToTau)
											continue;
										else
											cost += muToTau - medoidToTau;

								// Pi is tau's current closest medoid, so cost would
								// change if mu becomes a medoid
								} else
									cost += muToTau - piToTau;
							}

							// This is our best switch so far
							if(cost < bestCost) {
								bestCost = cost;
								bestPiInd = piInd;
								bestMuInd = muInd;
							}
						}
					}

					// Better clustering configuration found, so switch medoids
					if(bestCost < 0.0)
						medoids[bestPiInd] = bestMuInd;

					// The algorithm has converged
					else
						medoidsChanged = false;
				}

				clusters = new ArrayList<ArrayList<CompareForm>>();
				for(int i=0; i<numClusters; ++i)
					clusters.add(new ArrayList<CompareForm>());

				// Construct clusters to make findClosestClusters faster
				for(int formInd=0; formInd<numForms; ++formInd)
					clusters.get(formToMedoid[formInd]).add(forms[formInd]);
			}



			/**
			 * Save the current clustering configuration to a file
			 *
			 * @param filename_ File to save clustering to
			 */
			public void saveClustering(String filename_) {
				try {
					ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(new
File(filename_)));
					oos.writeObject(this);
					oos.close();
				} catch (FileNotFoundException e) {
					e.printStackTrace();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}



			/**
			 * Load the current clustering configuration from a file and return the
			 * loaded KMedoids object
			 *
			 * @param filename_ File to load clustering from
			 */
			public static KMedoids loadClustering(String filename_) {
				ObjectInputStream ois;
				KMedoids loaded = null;

				// Load object from file
				try {
```

```
			ois = new ObjectInputStream(new FileInputStream(new File(filename_)));
			loaded = (KMedoids) ois.readObject();
			ois.close();
		} catch (IOException e) {
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}

		// Grab all fields from loaded object
		CompareForm[] forms_ = loaded.forms;
		int[] formToMedoid_ = loaded.formToMedoid;
		double[][] distances_ = loaded.distances;
		int numForms_ = loaded.numForms;
		int[] medoids_ = loaded.medoids;
		ArrayList<ArrayList<CompareForm>> clusters_ = loaded.clusters;
		int numClusters_ = loaded.numClusters;
		int medoidOfDataset_ = loaded.medoidOfDataset;

		// Create a new fresh KMedoids object
		KMedoids newObject = new KMedoids();
		newObject.forms = forms_;
		newObject.formToMedoid = formToMedoid_;
		newObject.distances = distances_;
		newObject.numForms = numForms_;
		newObject.medoids = medoids_;
		newObject.clusters = clusters_;
		newObject.numClusters = numClusters_;
		newObject.medoidOfDataset = medoidOfDataset_;

		return newObject;
	}



	/**
	 * Find the closest C clusters to a submitted document and return the
	 * documents contained within those clusters in one list
	 *
	 * @param form_ The submitted document to find clusters close to
	 * @param numClosestClusters_ The number of closest clusters to find, or C
	 *
	 * @return The documents contained within the C close clusters
	 */
	public ArrayList<CompareForm> findClosestClusters(CompareForm form_, int numClosestClusters_) {
		ArrayList<Integer> closestClusters = new ArrayList<Integer>(numClosestClusters_);
		ArrayList<Double> closestDistances = new ArrayList<Double>(numClosestClusters_);
		double worstDist = Double.NEGATIVE_INFINITY;
		double distance;
		double tempDist;
		int worstInd = -1;

		// Find the requested number of closest clusters
		for(int medoidInd=0; medoidInd<numClusters; ++medoidInd) {
			distance = form_.compare(forms[medoids[medoidInd]]);
			if(medoidInd < numClosestClusters_ || distance < worstDist) {

				// Remove the least close (worst) of the closest clusters if the
				//     ArrayList is full (i.e. after numClosestClusters_ iterations)
				if(medoidInd >= numClosestClusters_) {
					closestClusters.set(worstInd, medoidInd);
					closestDistances.set(worstInd, distance);

				// List isn't full yet, so this cluster is tentatively "close"
				} else {
					closestClusters.add(medoidInd);
					closestDistances.add(distance);
				}

				// Recalculate the least close (worst) of the closest clusters
				worstDist = Double.NEGATIVE_INFINITY;
				for(int j=0; j<closestDistances.size(); ++j) {
					tempDist = closestDistances.get(j);
					if(tempDist > worstDist) {
						worstDist = tempDist;
						worstInd = j;
					}
				}
			}
		}
```

```
		// Add all forms from the closest clusters to one list
		ArrayList<CompareForm> closestForms = new ArrayList<CompareForm>();
		for(int i : closestClusters)
			closestForms.addAll(clusters.get(i));

		return closestForms;
	}



	/**
	 * Within-cluster sum of squares, cluster cohesion. Smaller is better.
	 *
	 * @return WSS value
	 */
	public double getWSS() {
		double wss = 0.0;
		double dist;
		int medoidFormInd;

		for(int medoidInd=0; medoidInd<numClusters; ++medoidInd) {
			medoidFormInd = medoids[medoidInd];
			for(int formInd=0; formInd<numForms; ++formInd)
				if(formToMedoid[formInd] == medoidInd) {
					dist = 1.0 - distances[medoidFormInd][formInd];
					wss += dist*dist;
				}
		}

		return wss;
	}



	/**
	 * Between cluster sum of squares, cluster separation. Larger is better.
	 *
	 * @return BSS value
	 */
	public double getBSS() {
		double bss = 0.0;
		double dist;
		int medoidFormInd;

		for(int medoidInd=0; medoidInd<numClusters; ++medoidInd) {
			medoidFormInd = medoids[medoidInd];
			dist = 1.0 - distances[medoidFormInd][medoidOfDataset];
			bss += clusters.get(medoidInd).size() * dist*dist;
		}

		return bss;
	}
}
```

12.1.7 DBSCAN

```
/**
 * An implementation of DBSCAN to cluster documents into density-based clusters
 * given a distance epsilon and a minimum threshold minPts where a point is a
 * database document
 *
 * @author Tony Ohmann
 * @author Chris Roering
 * @author Laura Nierengarten
 */
public class DBSCAN implements ClusteringAlgorithm {
	private double epsilon = 0;
	private int minPoints = 0;
	private int numDocs = 0;

	private static double delta;

	CompareForm[] inputDocs;
	double[][] distMatrix;
	int[] pointLabels;

	// Constants to represent point labels
```

```
	private static int NOISE = 0,
					BORDER = 1,
					CORE = 2,
					NOISEBORDER = 3;

	ArrayList<ArrayList<CompareForm>> clusters;
	ArrayList<CompareForm> cluster_centroids;
	ArrayList<TreeSet<Integer>> finalClusterIndices;
	ArrayList<ArrayList<Integer>> clusterIndexPoints;

	CompareForm overall_centroid;

	double BSS, WSS;



	/**
	 * Read in documents and compute their distance matrix
	 *
	 * @param inputDocs_ The database documents to read in and later cluster
	 */
	public DBSCAN(ArrayList<CompareForm> inputDocs_) {
		numDocs = inputDocs_.size();
		inputDocs = new CompareForm[numDocs];

		// Read in and store the documents
		for(int formInd=0; formInd<numDocs; ++formInd)
			inputDocs[formInd] = inputDocs_.get(formInd);

		// Compute distance matrix values
		distMatrix = new double[numDocs][numDocs];
		for(int i=0; i<numDocs; ++i)
			for(int j=i; j<numDocs; ++j)
				distMatrix[i][j] = distMatrix[j][i] = inputDocs[i].compare(inputDocs[j]);
	}



	/**
	 * Unimplemented version of cluster(). DBSCAN uses cluster(double,int)
	 */
	public void cluster(int numClusters_) {
		throw new IllegalArgumentException("Wrong number of params. DBSCAN uses
cluster(double,int).");
	}



	/**
	 * Cluster documents using DBSCAN
	 *
	 * @param epsilon_ Distance for deciding on core and border points
	 * @param minPoints_ Minimum points threshold to label a point as core
	 * (must have minPts points within epsilon)
	 */
	public void cluster(double epsilon_, int minpoints_) {
		pointLabels = new int[numDocs];
		epsilon = epsilon_;
		// Parameter for computing index points
		delta = 2*epsilon;
		minPoints = minpoints_;

		LabelPoints();
		FormClusters();
	}



	/**
	 * Label each point as core, border, or noise
	 */
	public void LabelPoints() {
		// Initially classify everything as noise
		for(int i = 0; i < numDocs; ++i)
			pointLabels[i] = NOISE;

		// Find and label core points
		for(int i = 0; i < numDocs; i++) {
			int ptsWithinEpsilon = 0;

			// Find how many points are within epsilon of this
```

```
                for(int j = 0; j < numDocs; j++)
                        if (distMatrix[i][j] <= epsilon)
                                ptsWithinEpsilon++;
                if(ptsWithinEpsilon >= minPoints)
                        pointLabels[i] = CORE;
        }

        // Find and label border points
        for(int k = 0; k < numDocs; k++)
                if(pointLabels[k] == CORE)
                        for(int l = 0; l < numDocs; l++)
                                if(distMatrix[k][l] <= epsilon && pointLabels[l] != CORE)
                                        pointLabels[l] = BORDER;
}



/**
 * Create clusters from core/border points that are within epsilon of a
 * core point
 */
public void FormClusters() {
        ArrayList<TreeSet<Integer>> tentativeClusterIndices = new ArrayList<TreeSet<Integer>>();

        // For each core point, generate a list of core points within epsilon
        // of this core point and create for it a tentative cluster
        for(int i = 0; i < numDocs; i++)
                if(pointLabels[i] == CORE) {
                        TreeSet<Integer> core_cluster = new TreeSet<Integer>();
                        for(int j = 0; j < numDocs; j++)
                                if(pointLabels[j] == CORE && distMatrix[i][j] <= epsilon)
                                        core_cluster.add(j);
                        tentativeClusterIndices.add(core_cluster);
                }

        TreeSet<Integer> currentCluster;
        TreeSet<Integer> thisTentCluster, otherTentCluster;
        int tentativeClusterIndicesSize = tentativeClusterIndices.size();

        boolean finishedMerging;
        ArrayList<Integer> alreadyMerged = new ArrayList<Integer>();
        finalClusterIndices = new ArrayList<TreeSet<Integer>>();

        // Merge tentative clusters within epsilon of one another
        for (int i = 0; i < tentativeClusterIndicesSize; ++i) {

                // Each tentative cluster can only go through the merging process
                // once
                if(alreadyMerged.contains(i))
                        continue;

                // Mark this tentative cluster as merged (it's about to be)
                thisTentCluster = tentativeClusterIndices.get(i);
                alreadyMerged.add(i);

                // To store the cluster as it's merged
                currentCluster = new TreeSet<Integer>();
                currentCluster.addAll(thisTentCluster);

                finishedMerging = false;

                // Merge other tentative clusters with this one until we complete a
                // loop without any merges occurring
                while(!finishedMerging) {
                        finishedMerging = true;
                        for(int j = 0; j < tentativeClusterIndicesSize; ++j) {

                                // Do not re-process merged clusters
                                if(alreadyMerged.contains(j))
                                        continue;

                                // Check if another tentative cluster contains at least one
                                // of the same core points, i.e., intersects this one
                                otherTentCluster = tentativeClusterIndices.get(j);
                                if(Intersects(currentCluster, otherTentCluster)) {

                                        // There is at least one common core point, so these
                                        // tentative clusters should now be the same cluster
                                        currentCluster.addAll(otherTentCluster);
                                        alreadyMerged.add(j);
```

```
                        // A merge occurred
                        finishedMerging = false;
                    }
                }
            }

            finalClusterIndices.add(currentCluster);
        }

        // Assign border points to their correct clusters
        for (int m = 0; m < numDocs; m++)
            if (pointLabels[m] == BORDER) {

                // We know at least one core point is within epsilon of this
                // border point since it got labeled as a border point
                boolean foundCoreAlready = false;
                double minDistance = Double.MAX_VALUE;
                int indexOfSmallest = -1;

                // Find which core point it is closest to
                for(int n = 0; n < numDocs; n++) {
                    double distance = distMatrix[m][n];

                    // Only look at core points
                    if(pointLabels[n] == CORE) {

                        // This is the first core point found, so it's the
                        // closest so far
                        if(!foundCoreAlready) {
                            foundCoreAlready = true;
                            minDistance = distance;
                            indexOfSmallest = n;

                        // Check if this is the new closest core point
                        } else if(distance < minDistance) {
                            minDistance = distance;
                            indexOfSmallest = n;
                        }
                    }
                }

                // Assign this border point to the cluster containing the
                // closest core point
                for(TreeSet<Integer> clust : finalClusterIndices) {
                    if(clust.contains(indexOfSmallest)) {
                        clust.add(indexOfSmallest);
                        break;
                    }
                }
            }

        // Assign each of the noise point to a cluster
        // NOTE: This is not a typical DBSCAN step but prevents db documents
        //       from becoming useless after being classified as noise
        for(int m = 0; m < numDocs; ++m)
            if(pointLabels[m] == NOISE) {

                // There is no core point within epsilon, but a core point
                // might still be the closest point to this one
                boolean foundCoreBorderAlready = false;
                double minDistance = Double.MAX_VALUE;
                int indexOfClosest = -1;

                // Find which point it is closest to
                for(int n = 0; n < numDocs; n++)

                    // Only look at core and border points, as they are part of
                    // a cluster
                    if(pointLabels[n] == CORE || pointLabels[n] == BORDER) {
                        double distance = distMatrix[m][n];

                        // This is the first core/border point found, so it's
                        // the closest so far
                        if(!foundCoreBorderAlready) {
                            foundCoreBorderAlready = true;
                            minDistance = distance;
                            indexOfClosest = n;

                        // Check if this is the new closest core/border point
                        } else if(distance < minDistance) {
                            minDistance = distance;
```

```
                                                        indexOfClosest = n;
                                                }
                                        }

                                // Assign this noise point to the cluster containing the
                                // closest core/border point and label it as a "noiseborder"
                                // point, meaning a noise point which is acting as a border
                                // point, since it was assigned to a cluster
                                for(TreeSet<Integer> clust : finalClusterIndices)
                                        if(clust.contains(indexOfClosest)) {
                                                clust.add(m);
                                                pointLabels[m] = NOISEBORDER;
                                                break;
                                        }
                }


        TreeSet<Integer> master = new TreeSet<Integer>();

        // Form actual clusters
        clusters = new ArrayList<ArrayList<CompareForm>>();
        for (TreeSet<Integer> finalClust : finalClusterIndices) {
                ArrayList<CompareForm> cluster = new ArrayList<CompareForm>();
                master.addAll(finalClust);
                for(Integer index : finalClust)
                        cluster.add(inputDocs[index]);
                clusters.add(cluster);
        }
}



/**
 * Get number of clusters
 *
 * @return Number of clusters
 */
public int getNumberOfClusters() {
        return clusters.size();
}



/**
 * Get number of noise points
 *
 * @return Number of noise points
 */
public int getNumberOfNoisePoints() {
        int num_noise_points = 0;
        for(int i : pointLabels)
                if(i==NOISE)
                        num_noise_points++;
        return num_noise_points;
}



/**
 * Get number of documents assigned to any cluster
 *
 * @return Number of documents assigned to clusters
 */
public int GetFormsInClusters() {
        int num = 0;
        Forms:
        for(int i=0; i<numDocs; ++i)
                for(TreeSet<Integer> clust : finalClusterIndices)
                        if(clust.contains(i)) {
                                num++;
                                continue Forms;
                        }
        return num;
}



/**
 * Compute the index points of each cluster
 */
public void ComputeIndexPoints() {
```

```
        clusterIndexPoints = new ArrayList<ArrayList<Integer>>();
        ArrayList<Integer> indexPoints;
        int numClusters = getNumberOfClusters();

        // For every cluster
        for(int i = 0; i < numClusters; ++i) {

                // Retrieve the cluster itself
                TreeSet<Integer> cluster = finalClusterIndices.get(i);
                indexPoints = new ArrayList<Integer>();

                // Every point in the cluster is a potential index point
                FindIndexPoints:
                for(Integer index : cluster) {

                        // Points originally labeled as noise cannot be index points
                        if(pointLabels[index] == NOISEBORDER)
                                continue;

                        // Check if this point is within delta of an existing index
                        // point
                        for(Integer otherIndex : indexPoints)
                                if(distMatrix[index][otherIndex] <= delta)
                                        continue FindIndexPoints;

                        // If we get here, there are no other index points within delta
                        indexPoints.add(index);
                }

                // If everything is within delta of each other, indexPoints will be
                // empty, so simply add the first point in the cluster
                if(indexPoints.size() == 0)
                        indexPoints.add(cluster.first());

                // Store this cluster's calculated index points
                clusterIndexPoints.add(indexPoints);
        }
}



/**
 * Prints epsilon, minPts, and the number of clusters
 */
@Override
public String toString() {
        String cluster_string = "\nEpsilon: " + epsilon +
                        "\nMinPts: " + minPoints +
                        "\nNumber of Clusters: " + getNumberOfClusters();
        return cluster_string;
}



/**
 * Returns whether or not two TreeSets intersect set-wise
 *
 * @param left_ The first set
 * @param right_ The second set
 *
 * @return Whether the sets intersect
 */
public static boolean Intersects(TreeSet<Integer> left_, TreeSet<Integer> right_) {
        for (Integer i : left_)
                if (right_.contains(i))
                        return true;

        return false;
}



/**
 * Find the closest C clusters to a submitted document and return the
 * documents contained within those clusters in one list
 *
 * @param form_ The submitted document to find clusters close to
 * @param numClosestClusters_ The number of closest clusters to find, or C
 *
 * @return The documents contained within the C close clusters
```

```
	 */
	public ArrayList<CompareForm> findClosestClusters(CompareForm form_, int numClosestClusters_) {

		ArrayList<Integer> closestClusters = new ArrayList<Integer>(numClosestClusters_);
		ArrayList<Double> closestDistances = new ArrayList<Double>(numClosestClusters_);
		clusterIndexPoints = new ArrayList<ArrayList<Integer>>();
		double worstDist = Double.NEGATIVE_INFINITY;
		double localDist;
		double closestLocalDistance;
		double tempDist;
		int worstInd = -1;
		int numClusters = getNumberOfClusters();

		// Find the requested number of closest clusters
		for(int clusterIndex=0; clusterIndex<numClusters; ++clusterIndex) {

			// Find the smallest distance from one of the index points to the
			// submitted form
			closestLocalDistance = Double.MAX_VALUE;
			if(clusterIndexPoints.size() > 0)
				for(Integer index_point : clusterIndexPoints.get(clusterIndex)) {

					// Distance to this index point
					localDist = inputDocs[index_point].compare(form_);

					// Keep looking for the closest of this cluster's index
					// points
					if(localDist < closestLocalDistance)
						closestLocalDistance = localDist;
				}

			// If the list isn't full or this one is closer than one of the
			// other clusters
			if (clusterIndex < numClosestClusters_ || closestLocalDistance < worstDist) {

				// List isn't full yet, so this cluster is tentatively "close"
				if(clusterIndex <= numClosestClusters_) {
					closestClusters.add(clusterIndex);
					closestDistances.add(closestLocalDistance);

				// This cluster is closer than the least close one currently
				} else {
					closestClusters.set(worstInd, clusterIndex);
					closestDistances.set(worstInd, closestLocalDistance);
				}

				// Recalculate the least close (worst) of the closest clusters
				worstDist = Double.NEGATIVE_INFINITY;
				for(int j = 0; j < closestDistances.size(); ++j) {
					tempDist = closestDistances.get(j);
					if(tempDist > worstDist) {
						worstDist = tempDist;
						worstInd = j;
					}
				}
			}
		}

		// Add all forms from the closest clusters to one list
		ArrayList<CompareForm> closestForms = new ArrayList<CompareForm>();
		for(int i : closestClusters)
			closestForms.addAll(clusters.get(i));

		return closestForms;
	}
}
```

12.1.8 ClusterTask (Combined approach)

```
/**
 * Fork-join task that compares all submission forms to its sublist of
 * submission+database forms (allForms) using loops (for after clustering)
 *
 * @author Tony Ohmann
 */
public class ClusterTask extends RecursiveTask<Matrix> {
```

```
	ArrayList<CompareForm> submissionForms;
	ArrayList<CompareForm> myForms;
	private ArrayList<ArrayList<CompareForm>> myCloseFormLists;
	private Matrix matrix;



	/**
	 * Intialize lists of forms and result matrices
	 *
	 * @param startIndex_ The start of which submission forms this task is
	 * responsible for
	 * @param endIndex_ The end of which submission forms this task is
	 * responsible for
	 * @param submissionForms_ All submission forms
	 * @param closeFormLists_ Lists of forms from the clusters closest to this
	 * task's sublist of submission forms
	 * @param distMetric_ The distance metric
	 * @param row_ The document names for the matrix header
	 */
	public ClusterTask(int startIndex_, int endIndex_, ArrayList<CompareForm> submissionForms_,
ArrayList<ArrayList<CompareForm>> closeFormLists_, int distMetric_, ArrayList<String> row_) {
		HashMap<String, Integer> oldNgrams;
		HashMap<String, Integer> newNgrams;
		submissionForms = new ArrayList<CompareForm>();
		myForms = new ArrayList<CompareForm>();
		myCloseFormLists = new ArrayList<ArrayList<CompareForm>>();

		// NUMA optimization: make a local-memory copy of the index
		Index oldIndex = submissionForms_.get(0).getIndex();
		HashMap<String, Integer> oldCountTable = oldIndex.countTable;
		HashMap<String, Integer> newCountTable = new HashMap<String, Integer>();
		for(String s : oldCountTable.keySet())
			newCountTable.put(new String(s), new Integer(oldCountTable.get(s)));
		Index newIndex = new Index(oldIndex.totalDocuments, newCountTable);

		// NUMA optimization: make a local-memory copy of all submission forms
		for(CompareForm form : submissionForms_) {
			oldNgrams = form.getGrams();
			newNgrams = new HashMap<String, Integer>();
			for(String s : oldNgrams.keySet())
				newNgrams.put(new String(s), new Integer(oldNgrams.get(s)));
			submissionForms.add(new CompareForm(new String(form.getName()), newIndex, newNgrams,
distMetric_));
		}

		// Get this task's submission forms and corresponding close clusters
		for(int i=startIndex_; i<=endIndex_; ++i) {
			myForms.add(submissionForms.get(i));
			myCloseFormLists.add( closeFormLists_.get(i) );
		}

		// Make a local-memory copy of the matrix row names
		ArrayList<String> row = new ArrayList<String>(row_.size());
		for(String s : row_)
			row.add(new String(s));

		// Col and row reversed for efficient result combination in NGCE
		matrix = new Matrix(myForms.size(), row_.size());
		matrix.setCol(row);
	}



	/**
	 * Compare this submission form to all submission forms and to our sublist
	 * of database forms
	 *
	 * @return The matrix containing distance measures between the compared
	 * forms
	 */
	protected Matrix compute() {
		CompareForm s;

		for(int i=0; i<myForms.size(); ++i) {
			s = myForms.get(i);

			// Again, col and row reversed for efficiency in NGCE
			// Compare to closest-cluster database forms
			for(CompareForm d : myCloseFormLists.get(i))
				matrix.add(d.getName(), s.getName(), s.compare(d));
```

```
                        // Compare to all other submission forms
                        for(CompareForm d : submissionForms)
                                matrix.add(d.getName(), s.getName(), s.compare(d));
                }

                return matrix;
        }
}
```

12.1.9 MossConverter (Calculating accuracy)

```
/**
 * MOSS and IPPDC results converter. Generates f1-score and other statistics
 * given a list of most similar pairs of documents.
 *
 * @author Tony Ohmann
 */
public class MossConverter {

        /**
         * Run the MOSS and IPPDC results converter. Generates f1-score and other
         * statistics given a list of most similar pairs of documents.
         *
         * @param args To do MOSS style conversion, pass "MOSS". To do clustering
         * conversion, pass "kmedoids [k]" or "dbscan [eps] [minPts]". Anything
         * else does normal IPPDC results conversion.
         *
         * @throws Exception
         */
        private void run(String[] args) throws Exception {

                int NUM_PAIRS = 200;

                String outputDir = "./output";
                Scanner in = null;
                PrintStream out = null;

                int k = 2;
                String eps = "";
                String minPts = "";
                int numClusters = 1;

                // Set clustering type
                String clustering = "";
                if(args.length > 0)
                        clustering = args[0];

                if(clustering.equals("kmedoids")) {

                        // Get k, set up base as "kmedoids_[k]"
                        k = Integer.parseInt(args[1]);
                        String clusteringBase = clustering + "_" + args[1];

                        // Set up input and output
                        in = new Scanner(new
File(outputDir+"/results_"+clusteringBase+"_"+numClusters+".txt"));
                        out = new PrintStream(new
FileOutputStream(outputDir+"/results_"+clusteringBase+"_STATS.csv"));
                        out.println("k, numClosestClusters, runtime, TP, P, R, f1, f1 (I), TP (I), f1 (II), TP
(II), f1 (III), TP (III), f1 (IV), TP (IV), f1 (V), TP (V), f1 (mult), TP (mult)");

                } else if(clustering.equals("dbscan")) {

                        // Get eps and minPts, set up base as "dbscan_[eps]_[minPts]"
                        eps = args[1];
                        minPts = args[2];
                        String clusteringBase = clustering + "_" + args[1] + "_" + args[2];

                        // Set up input and output using base name
                        in = new Scanner(new
File(outputDir+"/results_"+clusteringBase+"_"+numClusters+".txt"));
                        out = new PrintStream(new
FileOutputStream(outputDir+"/results_"+clusteringBase+"_STATS.csv"));
                        out.println("eps, minpts, numClosestClusters, runtime, TP, P, R, f1, f1 (I), TP (I),
f1 (II), TP (II), f1 (III), TP (III), f1 (IV), TP (IV), f1 (V), TP (V), f1 (mult), TP (mult)");

                } else {
```

```
			clustering = "";
			in = new Scanner(new File(outputDir+"/results.txt"));
			out = new PrintStream(new FileOutputStream(outputDir+"/results_STATS.txt"));
			out.println("TP, P, R, f1, f1 (I), TP (I), f1 (II), TP (II), f1 (III), TP (III), f1 
(IV), TP (IV), f1 (V), TP (V), f1 (mult), TP (mult)");
		}

		// Get runtime if using clustering input
		double runtime = 0.0;
		if(in.hasNextDouble()) {
			runtime = in.nextDouble();
			in.nextLine();
		}

		String line, left_proj=null, right_proj=null, left_form = null, right_form = null;
		int[] tps_i = new int[1000];
		int[] tps_ii = new int[1000];
		int[] tps_iii = new int[1000];
		int[] tps_iv = new int[1000];
		int[] tps_v = new int[1000];
		int[] tps_m = new int[1000];
		int[] fps_i = new int[1000];
		int[] fps_ii = new int[1000];
		int[] fps_iii = new int[1000];
		int[] fps_iv = new int[1000];
		int[] fps_v = new int[1000];
		int[] fps_m = new int[1000];

		// Parse most similar pairs to calculate TPs and FPs
		for(int i=0; in.hasNextLine(); ++i) {
			line = in.nextLine();

			// Parsing MOSS's results
			if(args.length > 0 && args[0].equalsIgnoreCase("moss")) {
				Scanner left_sc;
				Scanner right_sc;
				int start;

				left_sc = new Scanner( line.substring(1, line.indexOf(" ")) );
				left_sc.useDelimiter("/");

				start = line.indexOf(",\"all")+2;
				right_sc = new Scanner( line.substring(start, line.indexOf(" ", start)) );
				right_sc.useDelimiter("/");

				left_sc.next();
				left_proj = left_sc.next();
				while(left_sc.hasNext())
					left_form = left_sc.next();

				right_sc.next();
				right_proj = right_sc.next();
				while(right_sc.hasNext())
					right_form = right_sc.next();
			}

			// Parsing IPPDC's results
			else {
				Scanner sc;

				sc = new Scanner(line);
				sc.useDelimiter("[_ ]");

				sc.next();
				left_proj = sc.next();
				sc.next();
				left_form = sc.next();

				sc.next();
				right_proj = sc.next();
				sc.next();
				right_form = sc.next();
			}

			// Mark if this pair is a TP or a FP of one of the plagiarism types
			if(right_proj.equals("plagtypei") || left_proj.equals("plagtypei")) {
				if((right_proj.equals("orig") || left_proj.equals("orig")) && 
left_form.equals(right_form))
					tps_i[i] = 1;
				else
					fps_i[i] = 1;
```

```
				}
				if(right_proj.equals("plagtypeii") || left_proj.equals("plagtypeii")) {
					if((right_proj.equals("orig") || left_proj.equals("orig")) &&
left_form.equals(right_form))
						tps_ii[i] = 1;
					else
						fps_ii[i] = 1;
				}
				if(right_proj.equals("plagtypeiii") || left_proj.equals("plagtypeiii")) {
					if((right_proj.equals("orig") || left_proj.equals("orig")) &&
left_form.equals(right_form))
						tps_iii[i] = 1;
					else
						fps_iii[i] = 1;
				}
				if(right_proj.equals("plagtypeiv") || left_proj.equals("plagtypeiv")) {
					if((right_proj.equals("orig") || left_proj.equals("orig")) &&
left_form.equals(right_form))
						tps_iv[i] = 1;
					else
						fps_iv[i] = 1;
				}
				if(right_proj.equals("plagtypev") || left_proj.equals("plagtypev")) {
					if((right_proj.equals("orig") || left_proj.equals("orig")) &&
left_form.equals(right_form))
						tps_v[i] = 1;
					else
						fps_v[i] = 1;
				}
				if(right_proj.equals("plagmixed") || left_proj.equals("plagmixed")) {
					if((right_proj.equals("orig") || left_proj.equals("orig")) &&
left_form.equals(right_form))
						tps_m[i] = 1;
					else
						fps_m[i] = 1;
				}
			}
			in.close();

			StringBuffer sb = new StringBuffer();
			int num_tps_total, num_tps_i, num_tps_ii, num_tps_iii, num_tps_iv, num_tps_v, num_tps_m;
			int num_fps_total, num_fps_i, num_fps_ii, num_fps_iii, num_fps_iv, num_fps_v, num_fps_m;
			int tp, fp, fn;

			// Sum the TPs for each type of plagiarism
			num_tps_i = 0;
			num_tps_ii = 0;
			num_tps_iii = 0;
			num_tps_iv = 0;
			num_tps_v = 0;
			num_tps_m = 0;
			for(int pair=0; pair<NUM_PAIRS; ++pair) {
				num_tps_i += tps_i[pair];
				num_tps_ii += tps_ii[pair];
				num_tps_iii += tps_iii[pair];
				num_tps_iv += tps_iv[pair];
				num_tps_v += tps_v[pair];
				num_tps_m += tps_m[pair];
			}
			num_tps_total = num_tps_i + num_tps_ii + num_tps_iii + num_tps_iv + num_tps_v + num_tps_m;

			// Sum the FPs for each type of plagiarism
			num_fps_i = 0;
			num_fps_ii = 0;
			num_fps_iii = 0;
			num_fps_iv = 0;
			num_fps_v = 0;
			num_fps_m = 0;
			for(int pair=0; pair<NUM_PAIRS; ++pair) {
				num_fps_i += fps_i[pair];
				num_fps_ii += fps_ii[pair];
				num_fps_iii += fps_iii[pair];
				num_fps_iv += fps_iv[pair];
				num_fps_v += fps_v[pair];
				num_fps_m += fps_m[pair];
			}
			num_fps_total = num_fps_i + num_fps_ii + num_fps_iii + num_fps_iv + num_fps_v + num_fps_m;


			// The total TPs, FPs, and FNs for all plagiarism types
			tp = num_tps_total;
```

```
		fp = num_fps_total;
		fn = 180 - num_tps_total;

		// Print clustering-specific stats
		if(clustering.equals("kmedoids"))
			sb.append(k + ", " + numClusters + ", " + runtime + ", ");
		else if(clustering.equals("dbscan"))
			sb.append(eps + ", " + minPts + ", " + numClusters + ", " + runtime + ", ");

		// TPs, P, R, f1-score
		sb.append(tp + ", " + (1.0*tp/(tp+fp)) + ", " + (1.0*tp/(tp+fn)) + ", " + get_fscore(tp, fp,
fn, 1) + ", ");

		// Print for Type I plagiarism: f1-score, TPs
		tp = num_tps_i;
		fp = num_fps_i;
		fn = 30 - num_tps_i;
		sb.append(get_fscore(tp, fp, fn, 1) + ", " + tp + ", ");

		// Print for Type II plagiarism: f1-score, TPs
		tp = num_tps_ii;
		fp = num_fps_ii;
		fn = 30 - num_tps_ii;
		sb.append(get_fscore(tp, fp, fn, 1) + ", " + tp + ", ");

		// Print for Type III plagiarism: f1-score, TPs
		tp = num_tps_iii;
		fp = num_fps_iii;
		fn = 30 - num_tps_iii;
		sb.append(get_fscore(tp, fp, fn, 1) + ", " + tp + ", ");

		// Print for Type IV plagiarism: f1-score, TPs
		tp = num_tps_iv;
		fp = num_fps_iv;
		fn = 30 - num_tps_iv;
		sb.append(get_fscore(tp, fp, fn, 1) + ", " + tp + ", ");

		// Print for Type V plagiarism: f1-score, TPs
		tp = num_tps_v;
		fp = num_fps_v;
		fn = 30 - num_tps_v;
		sb.append(get_fscore(tp, fp, fn, 1) + ", " + tp + ", ");

		// Print for Multiple Type plagiarism: f1-score, TPs
		tp = num_tps_m;
		fp = num_fps_m;
		fn = 30 - num_tps_m;
		sb.append(get_fscore(tp, fp, fn, 1) + ", " + tp + "\n");

		out.print(sb);
		out.close();
	}

	/**
	 * Calculates the f-score for a given beta (e.g. beta=1 is f1-score)
	 *
	 * @param tp_ True positives
	 * @param fp_ False positives
	 * @param fn_ False negatives
	 * @param beta Parameter for f-score. If 1, this is f1-score
	 *
	 * @return The f-score value
	 */
	private double get_fscore(int tp_, int fp_, int fn_, int beta) {
		return ((1.0+beta*beta)*tp_) / ((1.0+beta*beta)*tp_ + (beta*beta)*fn_ + fp_);
	}

	/**
	 * (see run())
	 *
	 * @param args (see run())
	 * @throws Exception
	 */
	public static void main(String[] args) throws Exception {
		new MossConverter().run(args);
	}
}
```