Improving Plagiarism Detection

Tomas Votroubek

January 7, 2018

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

BACHELOR THESIS ASSIGNMENT

Student: Votrubek Tomáš

Study programme: Software Technology and Management
Specialisation: Software Engineering

Title of Bachelor Thesis: Improving the plagiarism detection on BRUTE

Guidelines:

1. Learn the principles of plagiarism detection systems for plain text documents and for source codes.
2. Identify and describe the shortcomings of Perl module Text::Similarity, which is currently in use as a plagiarism detector in BRUTE, with the emphasis on simple mechanical code modifications.
3. Test the characteristics of other existing plagiarism detectors.
4. Design and implement plagiarism detector resistant to simple methods of refactoring.
5. Test the system on real data and compare the result with the current state, and with the results of several chosen existing algorithms.

Bibliography/Sources:

[1] S. Schleimer, D. Wilkerson, A. Aiken. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03). ACM, New York, NY, USA, 76-85.
[2] Vítor T. Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz; Plagiarism Detection: A Tool Survey and Comparison. 3rd Symposium on Languages, Applications and Technologies (SLATE:14)
[3] J. Hage, P. Rademaker, N. van Vugt: A comparison of plagiarism detection tools.Technical Report UU-CS-2010-015, June 2010, Department of Information and Computing Sciences, Utrecht University

Bachelor Thesis Supervisor: Ing. Petr Pošík, Ph.D.

Valid until the end of the winter semester of academic year 2018/2019

L.S.

Prague, August 21, 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

. . . . . .

Tomáš Votroubek

Contents

Abstract

In this thesis, we compare the performance of commonly available plagiarism detectors to the solution used in the student task submission system used at *CTU FEE*, named *BRUTE*. We identify the limitation of existing detectors and describe a solution based on enhanced suffix arrays which mitigates them. Along with the detector, we also present multiple visualization tools built on top of its output which simplify the task of plagiarism detection.

Chapter 1

Introduction

Automation of plagiarism detection is important not only for education which prompted the creation of competitions, such as PAN [15], dedicated to testing and development of state-of-the-art detectors. However, these detectors are usually either proprietary or experimental, although their freely available counterparts rely on techniques not updated since the turn of the millennium. In this paper we will describe a simple algorithm for a fast universal plagiarism detector and compare its performance with existing detectors.

1.1 Current Situation at CTU

Currently, the online assignment management system of CTU named BRUTE (Bundle for Reservation, Uploading, Testing and Evaluation) uses a Perl module called Text::Similarity for the purposes of plagiarism detection. This module is, however, not a purpose-built detector and therefore, unlike most plagiarism detectors, no research paper exists testing its performance for this use-case. Furthermore, it is missing desirable features, such as visualization of the detected plagiarized sections of code.

1.2 Plagiarism

Plagiarism is usually defined as using another authors work and presenting it one's own. It is a case of academic dishonesty and an issue of ethics. As such, it is not decidable by a completely automated method.

In this thesis we will be using the similar definition of plagiarism detection to Potthast et al. [15]. We will specifically be dealing with *external plagiarism detection* and it's subtasks *source retrieval* and *text alignment*.

The task of *text alignment*, given a substring s_{src} and a corpus C, is to find a substring s_{plag} and a document $d_{plag} \in C$, where $s_{plag} \in d_{plag}$ and s_{plag} approximates s_{src} as closely as possible.

We define the task of *source retrieval*, given a corpus C and a document d_{plag} containing plagiarized substrings S_d, as the task of retrieving all documents $C_{plag} \subset C$ that contain any of the substrings in S_d, or their close approximations.

External plagiarism detection performs the tasks of plagiarism detection by inspecting other documents in the corpus, in contrast to *Intrinsic plagiarism*

detection which does so without inspecting other documents, usually by means of detecting stylistic changes throughout a document.

1.3 Sections

We explain some of the difficulties of plagiarism detection on concrete examples in section 2. We then introduce commonly used plagiarism detectors in section 3 and test their performance, along with the current solution used in BRUTE, on datasets we have specifically created for this purpose (4.1.1) in section 4.4. Based on the identified issues, we have created fast local and language independent algorithm, which could be used as an alternative to Text::Similarity. This algorithm is presented in section 5.1.1. Finally we implement this algorithm in F# [18] and test its performance on the same datasets in section 5.3.

Chapter 2

Issues of plagiarism detection

In this section we will introduce the issues of plagiarism detection on concrete examples and show the consequences of common limitations of existing detectors. These tests serve partially as proof for the following chapters, and as a more intuitive explanation of the importance of features such as lexical analysis.

We also show the effects of Unicode normalization on the detection performance, and why assuming ASCII input for languages using the Latin script may not be sufficient.

2.1 Lexical analysis

Lexical analysis is the process of converting data into tokens. These token can then be processed based on type. For plagiarism detection it is for example useful to normalize user defined identifiers and to move or completely remove source comments, as these changes are easy to perform even without understanding the subject matter. Simple tokenization by whitespace is not sufficient for plagiarism detection. Without lexical analysis any reformatting or renaming will result in widely different similarity scores. Changing `array [ x + 1 ]` to `array[x+1]` results in a similarity score of 0 %. Examples 1 and 2 prove this point for *Sherlock* and *Text::Similarity*:

```
$ echo "if ( x > 1 )" > long
$ echo "if(x>1)"      > short
$ ./sherlock -z 0 -n 1 -t 0 long short
long and short: 0%
```

Listing 1: Even though *Sherlock* tokenizes by punctuation, it reports 0% similarity on reformatted code.

Figure 2.1 shows this effect on real data. First, a detector was run on a data set with no formatting changes. The results are plotted in red. Secondly, one

```
$ perl text_similarity.pl \
       --type Text::Similarity::Overlaps \
       --string "array [ x + 1 ]" "array[x+1]"
0
```

Listing 2: *Text::Similarity* does not tokenize programming language correctly, which results in a 0% similarity between differently formatted code.

file of every detected pair was reformatted using an IDE, and this modified pair was tested for plagiarism again. The new results are plotted in black.

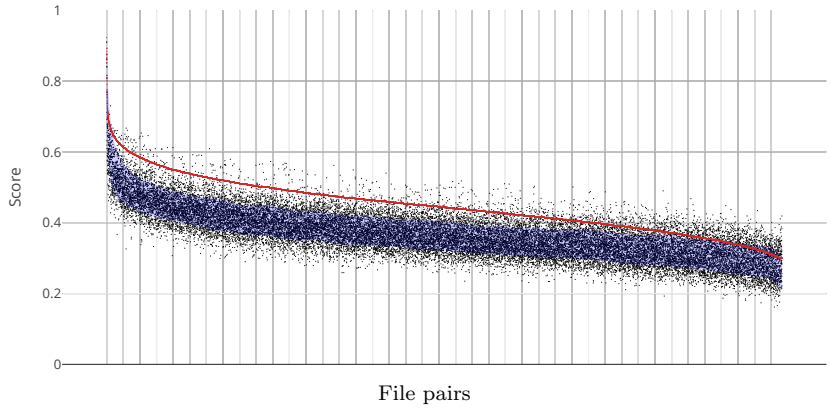


Figure 2.1: The result of running *Text::Similarity*. Incorrect tokenization has produced noise.

2.2 Unicode

2.2.1 Unicode Normalization

The Unicode Standard allows multiple ways to represent the same abstract character. Although these representations have different binary values, they are said to be canonically equivalent. Text must be normalized into one of Unicode Normalization Forms to determine whether any two strings are equivalent. These forms for example dictate the ordering of combining characters. *Canonical Decomposition* is one of these forms.

Changing the Unicode Normalization of a file, while recondite, is one of the easiest ways to influence the result of plagiarism detection. We do not have real data on which to measure its effect, nor do we know, whether this technique is actually used, but the following test might serve as a rough estimate of the seriousness.

Beginning with a Czech dictionary including word frequencies from Czech National Corpus, we have replicated each word by the natural logarithm of its actual frequency. The same process was repeated, but with Canonical Decomposition applied on each word. Similarity of the two files was then computed

using a detector which does not support Unicode and is position independent (Text::Similarity in this instance). Normalization was also applied on a book from the public domain Domácí kuchařka - Magdalena Dobromila Rettigová and it's similarity with the original copy was measured using the same detector.

```
$./detector.pl kucharka_original kucharka_NFD
kucharka_original,0.554672428428119,kucharka_NFD

$./detector.pl czech_corpus_original czech_corpus_NFD
czech_corpus_original,0.449659546165748,czech_corpus_NFD
```

Listing 3: Applying normalization reduced file similarity to around 50 %.

After applying Canonical Decomposition on the two files, their similarities to their original counterparts were reduced to around 50 %. Since the samples were small, the result is not conclusive and only serves as an example of when assuming that the input is ASCII encoded may not be sufficient for correct plagiarism detection.

2.3 Intentionally shared code

Student assignments often contain intentionally shared code. These sections of code will remain mostly unchanged and, when scaling the similarity by the length of the matched substring, could comprise a large portion of the similarity score. If a student changes this shared code, the similarity of the entire file is immediately reduced and the artificially created baseline similarity of the assignment could serve to hide the actually plagiarized sections. Other features, such as idiomatic phrases or obvious parts of the solution, are by definition likely to appear in multiple files but do not indicate plagiarism.

There should exist a mechanism to exclude these features from comparison. Labeling shared code sections is also useful for the purposes of manual plagiarism detection. Since assignment template files are usually preexistent, they may be used directly in detectors that support this feature. However, code might be obtained by attending the same lectures or classes. These cases should be additionally detected and excluded as well.

Figure 2.2 shows an extreme example of the noise on an introductory programming assignment where 60 % of the source code was provided as a starting point. First, JPlag was used to detect similarities between all pairs of files. Secondly, all pairs were sorted by similarity and plotted in red. Thirdly, the same detection was performed with the template file used for exclusion and the new scores were plotted in black in their respective positions along the Y-axis.

2.4 Granularity of detection

Most detectors have an option to customize the granularity of detection, be it by changing the length of n-grams, minimal number of tokens in a match, or by modifying its hashing algorithm. This setting is usually called either *granularity* (Sherlock), *run-length* (Sim), or *sensitivity* (JPlag).

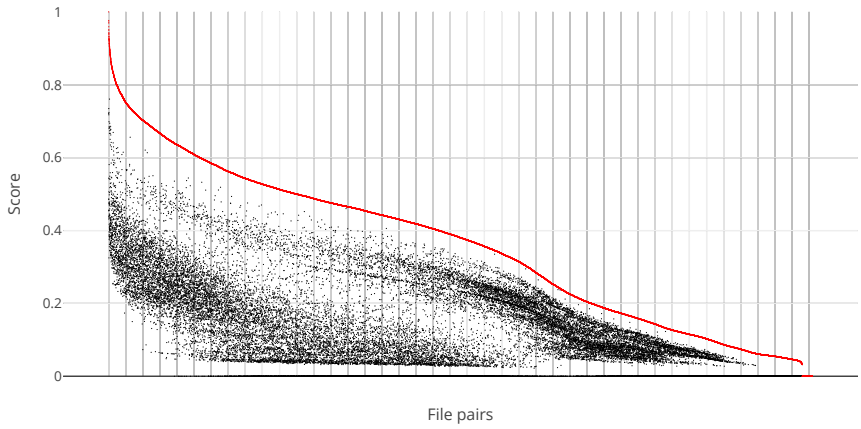
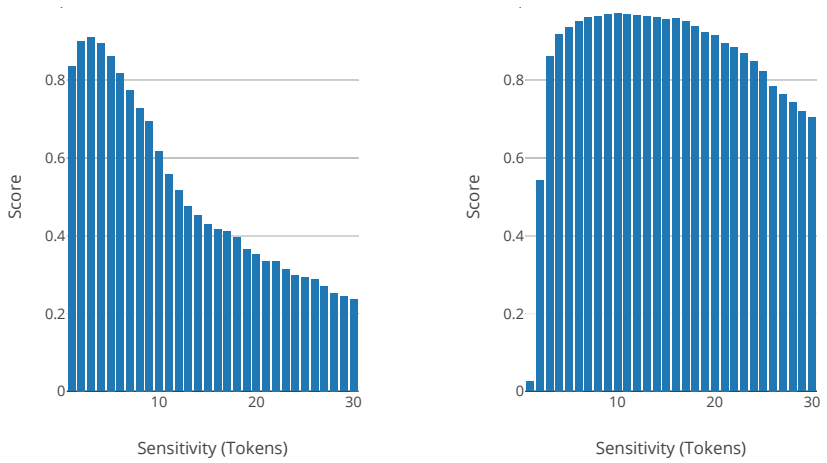


Figure 2.2: An extreme example of noise induced by the failure to exclude template code from the comparison. The correct score is plotted in black, while the score without template code exclusion is plotted in red.

Figure 2.3 shows the difference in optimal granularity between source code and text. From the figures, we can see that for JPlag the optimal *sensitivity* for English is around 3 tokens; for Java it is 10 tokens (excluding identifiers). The optimal setting depends on the number of distinct tokens, words, or characters in each language. For simple languages such as *C* which contains only about 32 keywords and 13 special characters the chance for a single token to appear in multiple files is higher than for example for Japanese which contains tens of thousands unique characters, and thus the optimal number of tokens is larger.



(a) Natural language.

(b) Source code.

Figure 2.3: Area under the PR curves of JPlag for sensitivity settings from 1 to 30 tokens.

2.5 Similarity threshold in BRUTE

The plagiarism detection system used in BRUTE can be set to only report matches with similarity above a certain threshold. The following table shows the relation between this threshold and the number of reported true positives and false positives. For this test we used the Corpus of Plagiarised Short Answers. This corpus contains four plagiarism categories: *Near copy*, *Light revision*, *Heavy revision*, *Non-plagiarism*. Each category is composed of 19 submissions.

The tables show that only very few files are correctly reported (true positives) even in the *Near copy* category with a threshold of 70 %, and that false positives do not appear even at the 50 % similarity threshold.

70 % threshold		
	Correctly reported	Incorrectly reported
Near copy	4	0
Light revision	4	0
Heavy revision	2	0
Non-plagiarism	0	0

50 % threshold		
	Correctly reported	Incorrectly reported
Near copy	14	0
Light revision	13	0
Heavy revision	10	0
Non-plagiarism	0	0

30 % threshold (average similarity)		
	Correctly reported	Incorrectly reported
Near copy	18	17
Light revision	18	21
Heavy revision	18	21
Non-plagiarism	0	43

The figure 2.4 shows the similarity distribution of all reported file pairs. We can se that only very few file pairs reach the similarity threshold of 70 % (shown in red). In fact only 9 % of plagiarized files have a score higher than this threshold; their average similarity is only 51,9 %.

2.6 Plagiarism verification

Perhaps the most difficult and the most important issue of plagiarism detection is verifying, whether detected similar sections of text are in fact plagiarized. Even sections with 100% similarity, might not be plagiarisms. Such instances may include obvious parts of a solution to a student assignment, software license files, code from the public domain with correct attribution in appropriate extent, or intentionally shared code. Labeling a student a plagiarizer may have serious ramifications for them. We believe that this step ought not be automated by tools with no notion of ethics and instead be left to a fair trial.

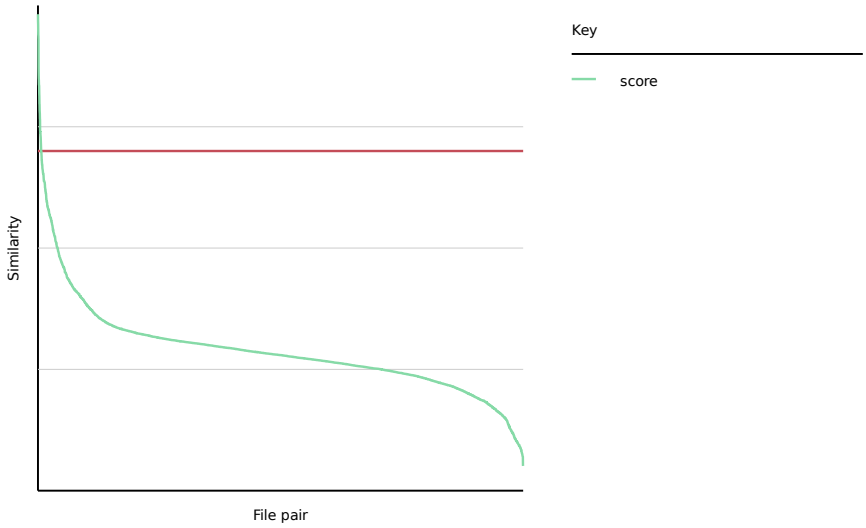


Figure 2.4: Score distribution for the Corpus of short plagiarized answers.

Chapter 3

Existing Detectors

To test the performance of our solution and of *Text::Similarity* we used existing alternative detectors as a benchmark. For the following tests we are mainly interested in local source code detectors. We believe the following detectors are still some of the most commonly used. For example Moss and JPlag can be used as plugins into Moodle.

3.1 JPlag

JPlag [7] is based on a string tiling algorithm. It performs tokenization and compares files pairwise to find similarities. Finally, it outputs the result as a folder of HTML files showing groups of files containing similar token strings. JPlag no longer offers an online detection service, but a program for local comparison is still actively maintained and can be downloaded from their GitHub page. JPlag can detect plagiarism in Java 1.7, C, C++, C# 1.2, Scheme and natural language.

3.2 Moss

Moss [17] performs file comparison using a patented local document fingerprinting algorithm called winnowing. It builds a map of selected fingerprints and file positions. The map is then used to look up all matching fingerprints for each file. Moss is an online-only tool and requires a free email registration. Moss supports the following languages: C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, HCL2.

Since Moss is an online-only detector, using this tool as a plagiarism detector in BRUTE may not be appropriate due to concerns about privacy.

3.3 PMD - CPD

CPD is a Copy/Paste Detector packaged with PMD [14] a cross-language static code analyzer. The main algorithm behind CPD has been improved multiple

times. From a variant of Michael Wise's Greedy String Tiling algorithm to the Karp-Rabin String Matching Algorithm. CPD performs tokenization using custom-written extendable tokenizers. It also supports multiple output formats including XML. CPD currently works with Java, JSP, C, C++, Fortran and PHP code, but support can be extended to other languages.

3.4 Sherlock

Sherlock [11] uses a digital signature generation scheme to find similarities in files. It first splits them by punctuation into tokens and then hashes their n-grams. Many of the hashed values are semi-randomly excluded from the comparison in order to perform a faster and more reliable detection. Sherlock finally outputs a plain text result containing a list of percentual pairwise file overlaps. Sherlock is Language-independent and supports all text files.

3.5 Sim

Sim [5] splits files into tokens using lexical analyzers written in *flex* [10]. It encodes these tokens into bytes, puts them into a hash table and uses it to compare every substring to every other substring. Sim can generate several plain text output formats as well as an intermediate result from its lexical analyzer. Sim currently supports the following languages: C++, C, Java, Pascal, Modula-2, Lisp, Miranda and natural language.

3.6 Other tested detectors

We have previously tested the following two detectors with moderate success, but had to exclude them from this comparison due to difficulty of use, requirements on file structure, and limited language support.

3.6.1 Marble

Marble [6] uses the Unix diff command internally to perform file comparison. It tokenizes files using a custom written tokenizer and restructures each file in an attempt to normalize them. Lastly, it uses the diff command to find similarities. Marble works on Java, C# and similar languages. While we had some success using this tool for plagiarism detection, we found it difficult to use due to its requirements on file structure and it's language support was too limited to be used in our tests.

3.6.2 Plaggie

Plaggie [1] is based on a Running Karp-Rabin Greedy String Tiling Algorithm. Since JPlag offered only an online plagiarism detection service, Plaggie intended to provide the same functionality locally using an open source implementation. Plaggie supports Java 1.5 only.

Chapter 4

Analysis of Existing Detectors

4.1 Source code

4.1.1 Data sets

Even though many datasets have been created for competitions such as PAN (Potthast et al. [15]) and SOCO(Flores et al. [4]) containing both natural-language and source code variants, we could not use these, due to them either not being publicly available anymore or aimed at source retrieval from a large corpus while minimizing retrieval cost. None of the publicly available detectors are suitable for such task. The main difference being that standard plagiarism detectors do not have the luxury of an source code corpus to check against and instead use only the submitted batch as their corpus. As a result most available detectors use algorithms which would run out of memory or time, before finishing such task.

Languages

Due to the similarity of object-oriented languages and the lack of support for other paradigms, only Python and C/C++ submissions were tested. The results, however, are applicable to most OOP languages, thanks to the simplicity of the algorithms underlying plagiarism detection.

Source data sets

The source for our data set generation were 321 real submitted C++ solutions and 140 Python solutions to student assignments, resulting in 270 viable C++ files and 114 Python files after cleanup.

Rather than using the submitted solutions themselves, new data sets were generated from them, in order to avoid the almost certainly impossible task of correctly detecting the directed graphs of plagiarism in the first place. This method also provides greater control over the performed tests, and more granular modifications can be used separately to test the sensitivity of detectors.

Preparation

Each original data set was manually inspected, and all files suspect of being involved in plagiarism or files which were accidentally too similar were removed. Additionally, all files shorter than 60 lines were removed along with files which were not parseable by either Clang or Python. The data sets were finally sanitized by removing difficult escape sequences and transliterating Unicode characters.

Transformations

The list is limited to only those transformations which could be performed either automatically, or by somebody with limited programming knowledge. Each transformation was selected to test the actual worst-case effects of limitations of existing plagiarism detection algorithms. We do not claim that the list is exhaustive, or that it contains all the most commonly employed techniques to avoid plagiarism. Nevertheless, the simpler methods are more likely to be used over the difficult ones and the detection performance for these transformations is most important.

We have created tools to automatically apply these following modification to source code. Each of these modifications resulted in a new data set. Plagiarism detectors were then tested to measure their sensitivity to each modification separately.

Simple Transformations on C++

We tested the following transformations. Their complete description is in section 7.1.1 of the appendix.

- Add comments
- Reformat code
- Rename identifiers
- Add assignment template
- Add original code
- Shuffle functions
- Reverse function parameters
- Extract methods
- Invert if-else branches
- Invert binary operations

While some of the transformations may not be completely lossless, the code will not be far from functional and only a very simple intervention by a human would be necessary. More capable tools exist and may be able to perform these changes perfectly. This, however, is not needed, as none of the tested detectors performs parsing, let alone runs the code. It might, however, be a problem for very inexperienced programmers, who might not be able to fix the code after the more complex transformations. Therefore, we leave the judgment of plausibility of usage of the more difficult obfuscation techniques to the reader.

The examples 4 and 5 show the *Extract method* transformation in action on a random snippet from Github Gists.

```
#define PAGE_SIZE 0x400

void lfsr_scramble(unsigned seed, unsigned page[PAGE_SIZE]) {
  int state = seed;

  unsigned int i;
  for (i = 0; i < PAGE_SIZE; i++) {
    unsigned j;
    for (j = 0; j < 8; j++)
      state = (state >> 1) | (((state^(state >> 1)) & 1) << 14);

    page[i] ^= (unsigned)(state >> 7);
  }
}
```

Listing 4: Original file

```
#define PAGE_SIZE 0x400

void extract0(unsigned int *page, int state, unsigned int i) {
  for (i = 0; i < PAGE_SIZE; i++) {
    unsigned j;
    for (j = 0; j < 8; j++)
      state = (state >> 1) | (((state^(state >> 1)) & 1) << 14);

    page[i] ^= (unsigned)(state >> 7);
  }
}

void lfsr_scramble(unsigned seed, unsigned page[PAGE_SIZE]) {
  int state = seed;

  unsigned int i;
  extract0(page, state, i);
}
```

Listing 5: Transformed file

Complex Transformations on C++

To test the actual performance of detectors and to understand the likelihood of successful detection avoidance, we have combined the previous modifications into complex transformations. The following modifications are supposed to represent more realistic stories of plagiarism attempts.

Sanity-Check The student copies the whole file.

Primitive The student copies the whole file, adds comments, and renames identifiers.

Simple The student copies 50 % of the functions but writes the rest himself. He adds comments and renames identifiers.

Realistic The student copies 50 % of the functions but writes the rest himself. He adds comments, renames identifiers, and reformats the code. An assignment template is provided to the students.

Difficult The student writes 50 % of the code himself but decides to plagiarize the remainder. He extracts blocks of code into methods and shuffles them. He then adds comments and renames identifiers to avoid dictionary based detectors. Finally, to evade string tiling, he reverses function parameter order. A template is provided for this assignment.

Impossible The student writes 50 % of the code himself but decides to plagiarize the remainder. He extracts blocks of code into methods and shuffles them. He then adds comments and renames identifiers to avoid dictionary based detectors. Finally, to evade string tiling, he reverses function parameter order, inverts if-else statements, and switches the operands of binary operations. A template is provided for this assignment.

Transformations on Python

The Python dataset contains transformations partly overlapping those of the C++ dataset, in order to verify the generality of the effects of refactoring on plagiarism detectors. However, unlike in the complex C++ dataset, we test the effects of transformations applied directly to copy-pasted code with no original input from the student. We tested the following transformations. Their complete description is in the appendix in section 7.1.2:

- Copy paste
- Reformatting
- Adding comments
- Renaming
- Reordering parameters
- Extracting functions
- Inverting binary operations
- Extracting constants
- Coding style
- All of the above

The examples 6 and 7 show the *Extract constants* and *Invert binary operations* transformations applied on a snippet from Rosetta Code.

```
#!/usr/bin/env python3
import cmath

def quad_discriminating_roots(a, b, c, entier=1e-5):
    """Naive algorithm"""
    discriminant = b * b - 4 * a * c
    a, b, c, d = complex(a), complex(b), complex(c), complex(discriminant)
    root1 = (-b + cmath.sqrt(d)) / 2. / a
    root2 = (-b - cmath.sqrt(d)) / 2. / a
    if abs(discriminant) < entier:
        return "real and equal", abs(root1), abs(root1)
    if discriminant > 0:
        return "real", root1.real, root2.real
    return "complex", root1, root2
```

Listing 6: Original file

```
#!/usr/bin/env python3
import cmath

s_real_and_equal = "real and equal"
s_real = "real"
s_complex = "complex"

def quad_discriminating_roots(a, b, c, entier=1e-5):
    """Naive algorithm"""
    discriminant = b * b - c * a * 4
    a, b, c, d = complex(a), complex(b), complex(c), complex(discriminant)
    root1 = (cmath.sqrt(d) + -b) / 2. / a
    root2 = (-b - cmath.sqrt(d)) / 2. / a
    if entier > abs(discriminant):
        return s_real_and_equal, abs(root1), abs(root1)
    if 0 < discriminant:
        return s_real, root1.real, root2.real
    return s_complex, root1, root2
```

Listing 7: File with extracted constants and inverted binary operations

Data set generation method

The source data set was partitioned into 3 equal groups and the groups were assigned one of the following roles: The source of plagiarism, the actual plagiarism, and clean submissions. Each group acted as every role exactly once.

The sources of plagiarism were copied as-is and were marked "source." The clean files were also copied and marked "clean." The actual plagiarisms were assigned one of the source files each and used code from both the file itself and the provided source file. Transformations were then applied to the plagiarized files only. The plagiarisms were marked with the source file name and "plagiarism."

4.2 Natural language

For testing detector performance on textual assignments, we used an existing Corpus of Plagiarised Short Answers described in Clough and Stevenson [2].

For the following tests we regarded as plagiarism any pair of files from the same task with categories for both files different from *non*, or if one of the files was the *original*.

4.3 Evaluation

To describe the performance of detectors we will be using, among others, the terms *precision* and *recall*. *Precision*, in our case, is the likelihood that a reported case is plagiarized. More specifically, it is the ratio of correctly reported plagiarized file pairs to the number of all reported file pairs. While *recall* is the percentage of plagiarism detected, or more formally, the ratio of the number of correctly reported plagiarized file pairs to the total number of plagiarized file pairs contained in the corpus.

In our view, the most important metric of a plagiarism detector is high precision, especially at low recall values. In other words, at least the highest scoring documents should actually be plagiarized. Nonetheless, precision is important throughout the entire recall range. To measure this we will be using the area under the detectors Precision-Recall (PR) curve (AUC).

We choose the PR curve over the more common Receiver operating characteristic (ROC) due to the large imbalance between the number of plagiarized file pairs and the number of all file pairs in a corpus. This imbalance could result in misleading results if the ROC curve were used as shown by Saito and Rehmsmeier [16].

PR plots (example 4.1) show the relation between precision and recall, with 0% recall on the left and 100% recall on the right, and 0% precision on the bottom and 100% precision on the top.

4.4 Results

Moss and JPlag are the best performing detectors and we recommend they be used whenever possible. The third best performing tool is Sim.

Text::Similarity works well for detecting plagiarism in textual assignments, but it is not suitable for detection in source code. Even the simplest methods

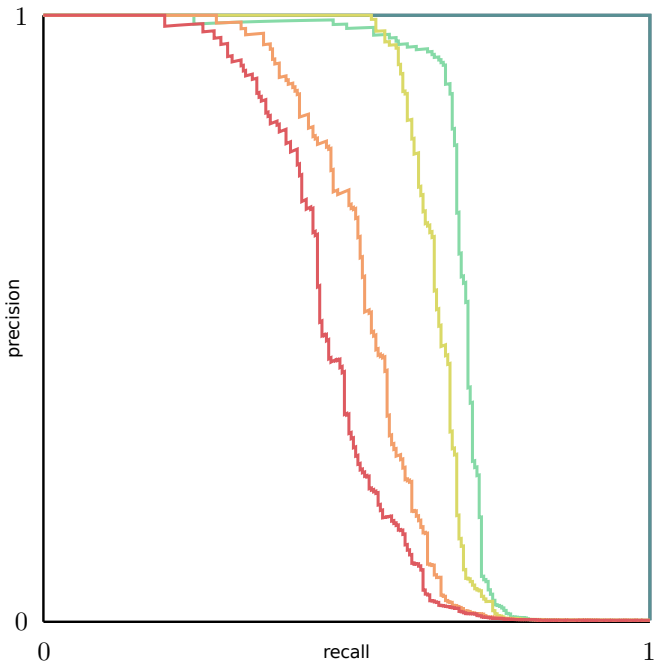


Figure 4.1: Example Precision Recall plot showing multiple PR curves.

for avoiding detection, such as changing comments and renaming, are likely to be successful. If Text::Similarity continues to be used, we recommend using a stopword list and decreasing the detection threshold from 70% to around 50% for textual assignments. For source code we recommend preforming the following preprocessing steps:

- Reformatting using tools such as autopep8 or indent.
- Removing comments.
- Removing intentionally shared code.
- Removing user defined identifiers.

The tests have identified the four most serious shortcomings which inhere in the current methods of plagiarism detection.

Lexical analysis Inappropriate or entirely missing lexical analysis, allows the simplest methods of disguising plagiarism to influence the detection results. This step is language dependent, but provides the largest increase in performance and is thus the most important. We will explain this issue on a concrete example in section 2.1.

Assignment template Matching and reporting similarity in intentionally shared parts of code artificially inflates scores, allows plagiarized code to hide amongst the noise, and might even be harmful as honest short solutions

will be unfairly scored more aggressively. An extreme example of this noise will be shown on a real set of real submissions 2.3.

Granularity of detection Setting the sequence length incorrectly will have a negative effect on the detection performance. Current detectors provide no indication of the appropriate value or lack this setting entirely resulting in diminished performance depending on the language complexity.

Languages Support for languages is generally very limited and even the most popular languages such as python may not be supported. This is partly related to the issue of language-dependent lexical analysis.

4.4.1 CPD

Simple C++ (Figure 7.1) Most simple transformations had no effect on the reported similarities. However, due to incorrect lexical analysis, renaming identifiers resulted in a significant decrease in the number and length of matched code sections and would be sufficient to avoid detection. This result is not applicable to Java, where CPD provides an option to completely ignore identifiers.

Complex C++ (Figure 4.2) CPD was only able to reliably detect copy-pasted source code. Renaming identifiers and adding an assignment template resulted in almost no reported similarities.

Python (Figure 7.2) CPD performed well on simple transformations and even on code with extracted constants, but it performed poorly after renaming identifiers. CPD was unable to detect files with multiple combined transformations.

Text (Figure 4.3) While CPD does not officially support detection of plagiarism in natural language, it's fall-back tokenization option was still able to provide some useful output. With a longer sequence length of 5 tokens the area under the curve reached 0.831. CPD was also able to indicate the locations of duplicated text, but due to the lack of preprocessing, the matched sequences were sparse and short.

4.4.2 JPlag

Simple C++ (Figure 7.3) All tested simple transformations had no effect on reported similarities and JPlag was able to detect plagiarized files perfectly. Even though inverting binary operations and reversing parameters could in some instances result in lower scores, it is not a concern in general and can not alone be used to avoid detection.

Complex C++ (Figure 4.4) JPlag was able to detect a large number of plagiarized files even after applying the Impossible transformation. Detection performance for the more complex transformations could be increased at the cost of a higher number of false positives in the simpler test cases by decreasing the minimal length of reported sequences. All cases of intentionally shared code must be explicitly included as an option in the

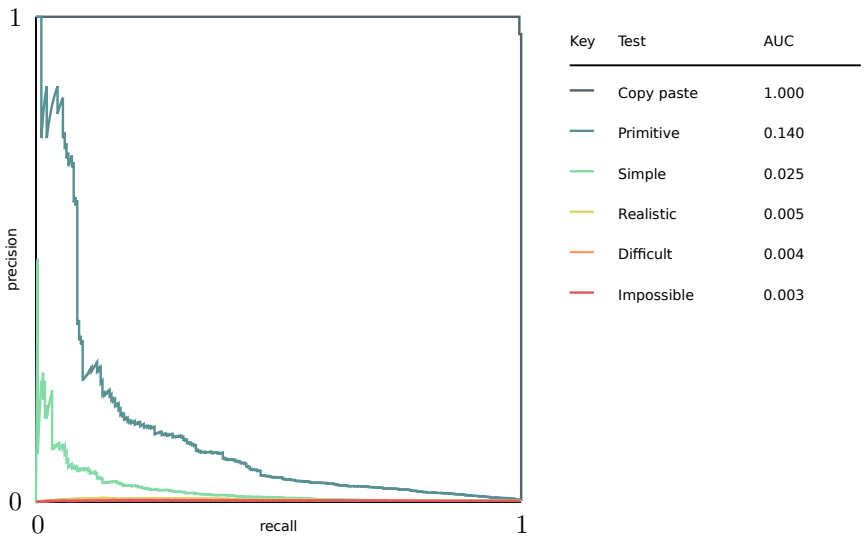


Figure 4.2: Precision recall curve of CPD on the Complex C++ corpus

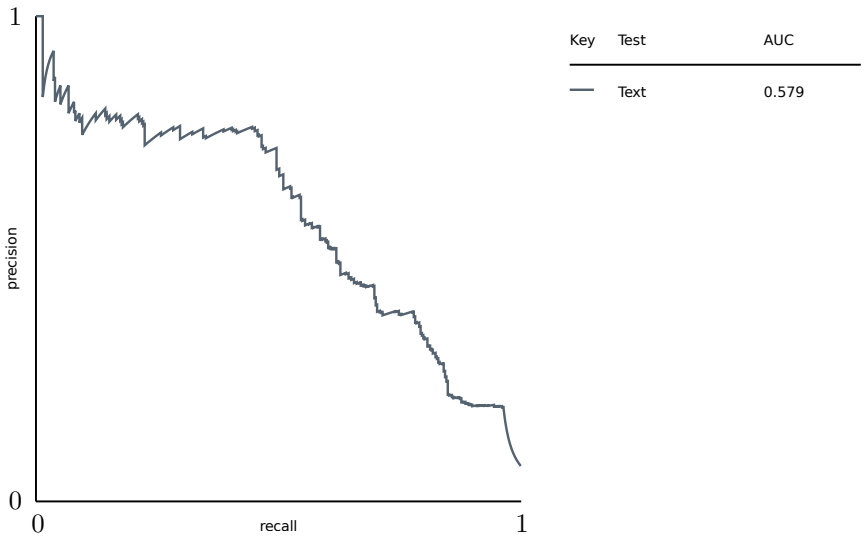


Figure 4.3: Precision recall curve of CPD on the Corpus of short plagiarized answers

query. The assignment template exclusion algorithm performed well after supplying a hand-written file and was able to highlight such code in its visual report.

Text (Figure 4.5) JPlag performs very well on plagiarism in natural language if the minimal reported sequence length is set to around 3 tokens. Even though JPlag offers no specialized preprocessing features it was able to reliably match similar sections of text and avoid most false positives until the end of its output.

The C++ test cases revealed an error in the tokenization stage of detection, where files containing longer escape sequences inside character literals or unusual Unicode characters, were reported as "non-parseable" and excluded from comparison. All such files had to be sanitized prior to detection.

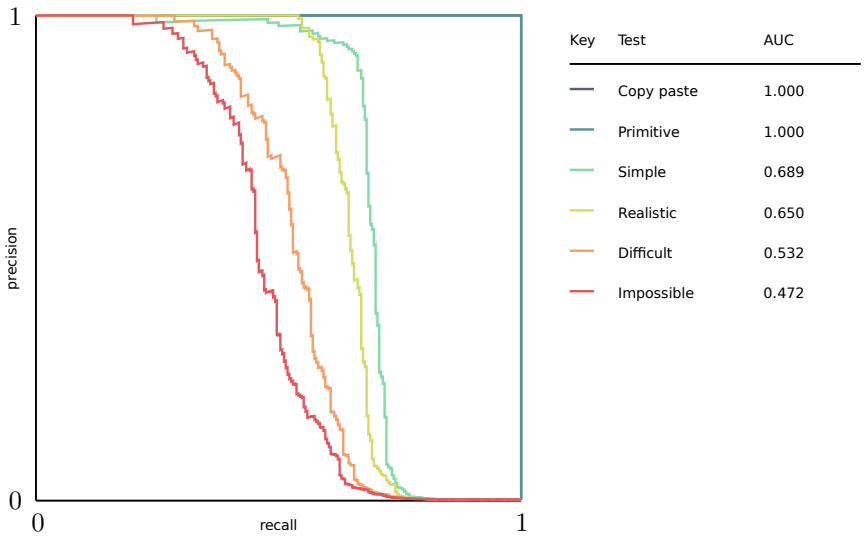


Figure 4.4: Precision recall curve of JPlag on the Complex C++ corpus

4.4.3 Moss

Simple C++ (Figure 7.4) Applying simple transformations had no effect in most cases and Moss was able to detect all plagiarized files. Inverting binary operations had only limited effect, and while it could be sufficient to avoid detection in some special instances, it is not a problem in general.

Complex C++ (Figure 4.6) Moss was able to detect a large percentage of plagiarized files with complex transformations. The combination of inverted If/Else statements, reversed parameters, and inverted binary operations resulted in modifications over a span of code shorter than the hard-coded minimal reported sequence length. The basecode exclusion algorithm was able to automatically remove almost all intentionally shared code from

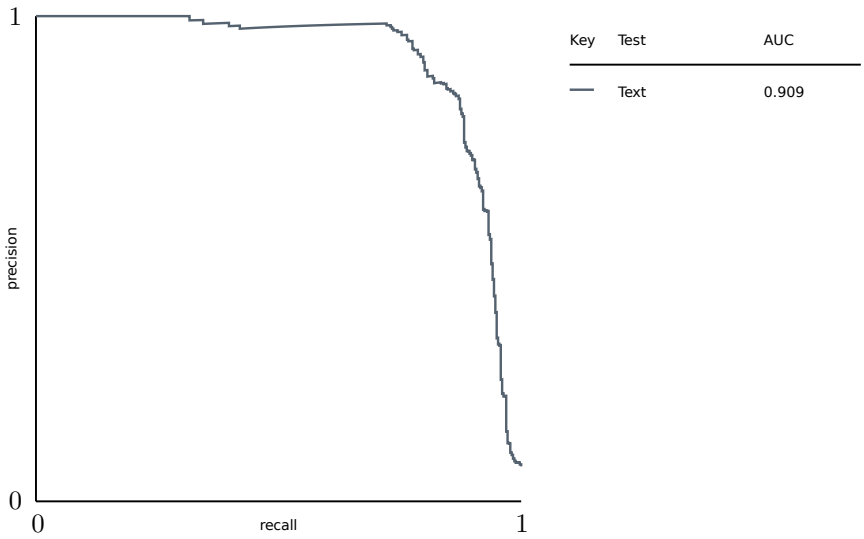


Figure 4.5: Precision recall curve of JPlag on the Corpus of short plagiarized answers

the comparison and providing a hand-written template file had almost no effect.

Python (Figure 7.5) Moss was able to detect most copy-pasted python files. Lexical analysis on Python works well and Reformatting, Comments, and Rename transformations had almost no effect. The more aggressive Extract method transformation could be sufficient to avoid detection. The Invert Binary Operations, and Extract Constant transformations were the most effective and would be sufficient to avoid detection in general.

Text (Figure 4.7) Moss provides no option to set the minimal reported sequence length, and the hard-coded value is the same for natural language as for programming languages. Nearly all items appearing in the result are correctly identified as a plagiarism. However, Moss is overly cautious and a large portion of files are not detected. Furthermore, only very few tokens from each file pair are matched, which might result in additional cases being dismissed for insufficient similarity. The area under the curve is 0.64, the median similarity of all reported file pairs is 13.5 %.

4.4.4 Sherlock

Simple C++ (Figure 7.6) Sherlock was able to perfectly deal with more granular changes to source code, such as inverting if / else statements and extracting methods. Sherlock is also effective on the more difficult transformations, such as inverting binary statements. However, due to its simple tokenization algorithm, reformatting had a large negative impact on its performance. Sherlock forgoes lexical analysis entirely which, coupled

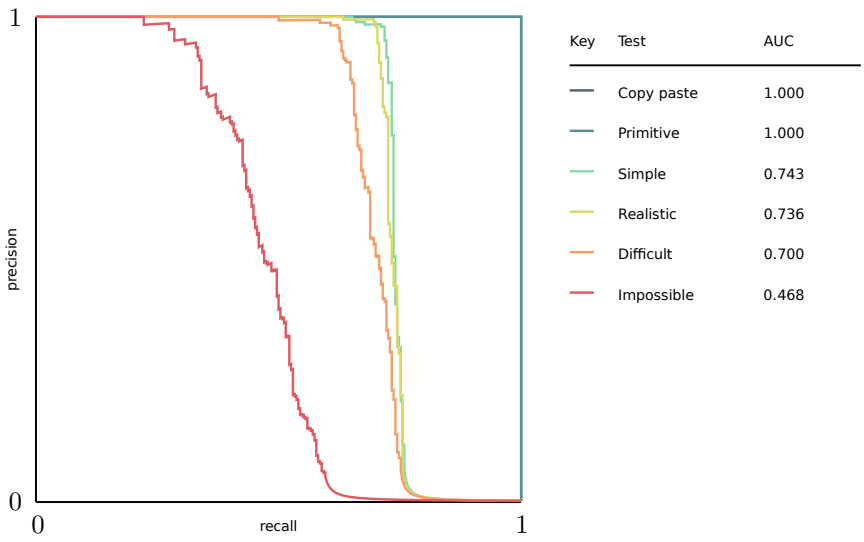


Figure 4.6: Precision recall curve of Moss on the Complex C++ corpus

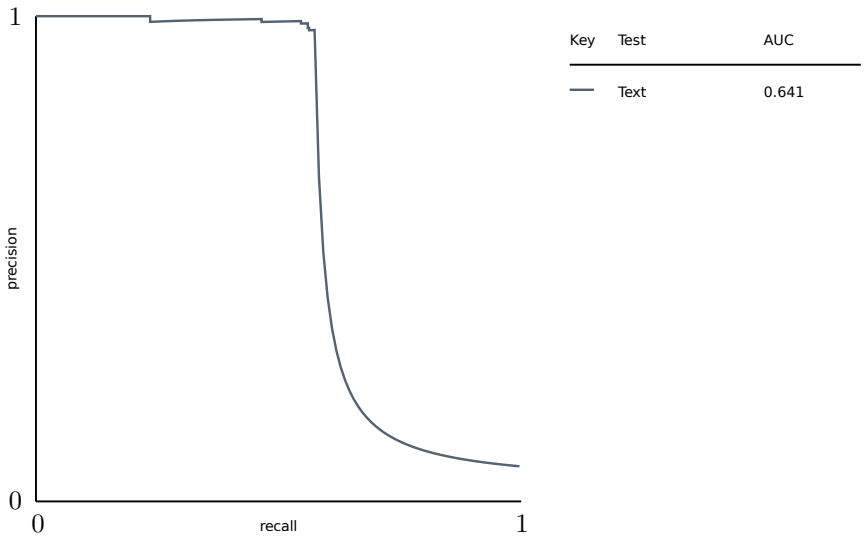


Figure 4.7: Precision recall curve of Moss on the Corpus of short plagiarized answers

with the dictionary based nature of its algorithm, has a result of decreasing the performance for larger n-gram settings in files with renamed identifiers. Renaming therefore had the largest effect on the results and could be sufficient to avoid detection.

Complex C++ (Figure 4.8) The combination of reformatting and renaming identifiers contained in all complex transformations resulted in almost no output for most datasets. Only the fully plagiarized files showed useful results at small recall values.

Python (Figure 7.7) The results for the python dataset are very similar to the C++ dataset. Sherlock detected almost every file with the more difficult transformations, while the performance for reformatting and renaming was not as good. The combination of renaming and reformatting resulted in almost no output.

Text (Figure 4.9) Sherlock performed very well on the text dataset. It detected almost every plagiarism while containing only a few false positives throughout most of it's output.

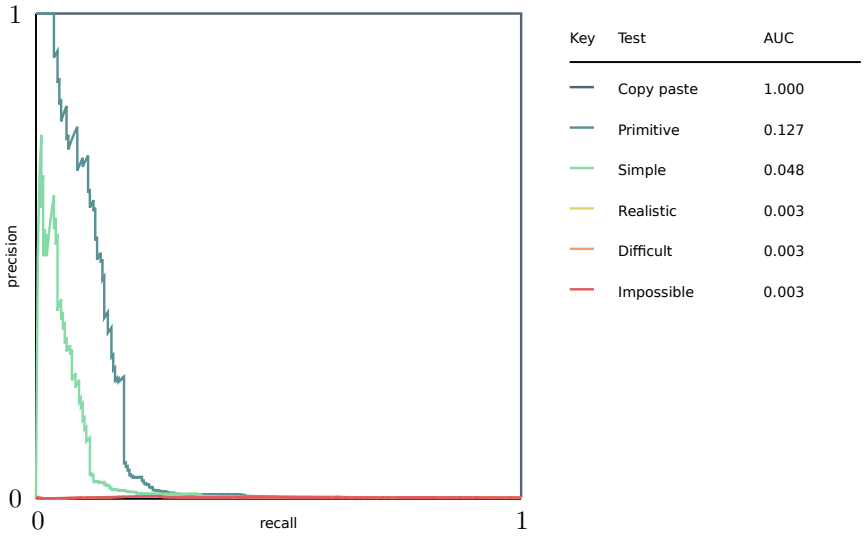


Figure 4.8: Precision recall curve of Sherlock on the Complex C++ corpus

4.4.5 Sim

Simple C++ (Figure 7.8) Sim was able to perfectly detect every file for all but one dataset. The dataset with inverted binary statements contained only a few false positives at high recall values. While this transformation can affect the results, it alone was not sufficient to avoid detection and it is unlikely that this would be an issue in general.

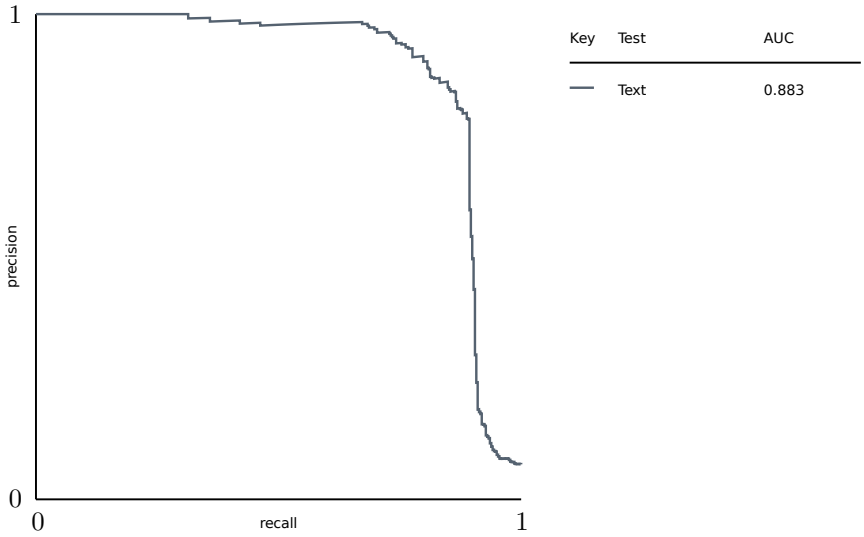


Figure 4.9: Precision recall curve of Sherlock on the Corpus of short plagiarized answers

Complex C++ (Figure 4.10) The partially plagiarized datasets did not score as highly and the output contained false positives. Sim still detected a majority of the files. Adding an assignment template resulted in a high baseline similarity. The ambient noise from applying this transformation resulted in the output being composed mostly of false positives. If the intentionally shared code was removed manually, Sim would likely perform reasonably well.

Text (Figure 4.11) Sim was able to detect most of the plagiarized files while reporting only a small number of false positives. Sim achieved a larger AUC at higher n-gram settings.

4.4.6 Text::Similarity

Simple C++ (Figure 7.9) Text::Similarity performed very well on datasets with simple transformations and was able to detect almost every file with no false positives. Both renaming identifiers and adding comments had some effect, but was not sufficient to avoid detection when done separately.

Complex C++ (Figure 4.12) Text::Similarity detected practically no plagiarism after applying complex transformations on the C++ dataset, mainly due to the lack of lexical analysis, incorrect tokenization, and no facility for excluding intentionally shared code from comparison. A simple combination of changing comments and renaming identifiers is almost certain to successfully avoid detection. Even with only automatic transformations being applied the detection performance was not significantly better than random chance. In this dataset only directly copy-pasted files were detected.

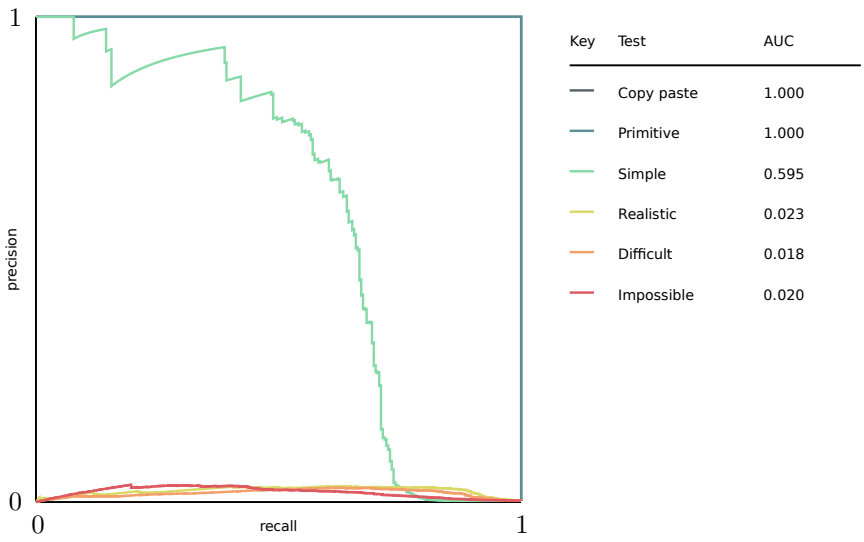


Figure 4.10: Precision recall curve of Sim on the Complex C++ corpus

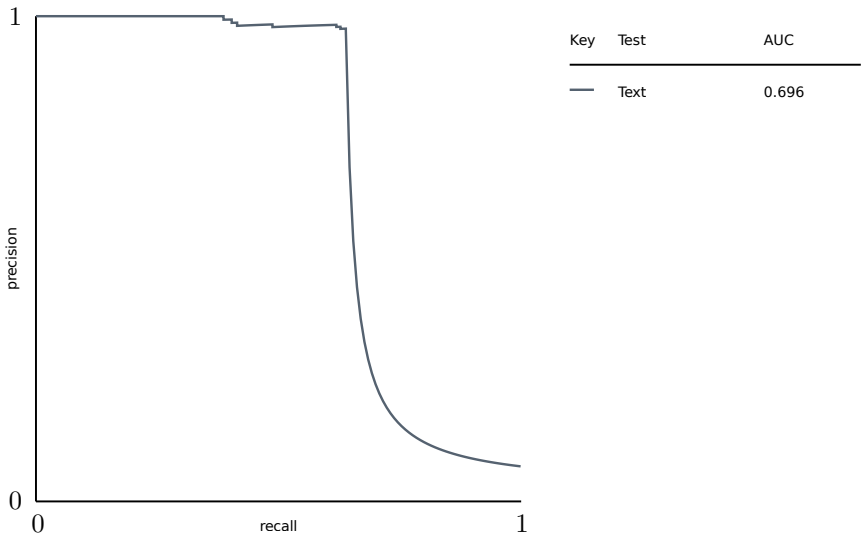


Figure 4.11: Precision recall curve of Sim on the Corpus of short plagiarized answers

Python (Figure 7.10) Performance on the python dataset was very similar to the C++ dataset. Applying transformations separately had only limited effect, while combining multiple transformations resulted in only a small number of correctly identified files. Even in the case of completely plagiarized files, simply renaming and changing comments will usually be sufficient to avoid detection.

Text (Figure 4.13) Text::Similarity detected most plagiarism in the natural language dataset and it's performance further improved after providing a stop-word list. Text::Similarity outputs several measures of similarity, but apart from *Lesk* and the F_1 *score* other measures serve mostly for debug purposes. The tests indicated that files, which are high in both of these measures are more likely to be plagiarisms than those high in only one.

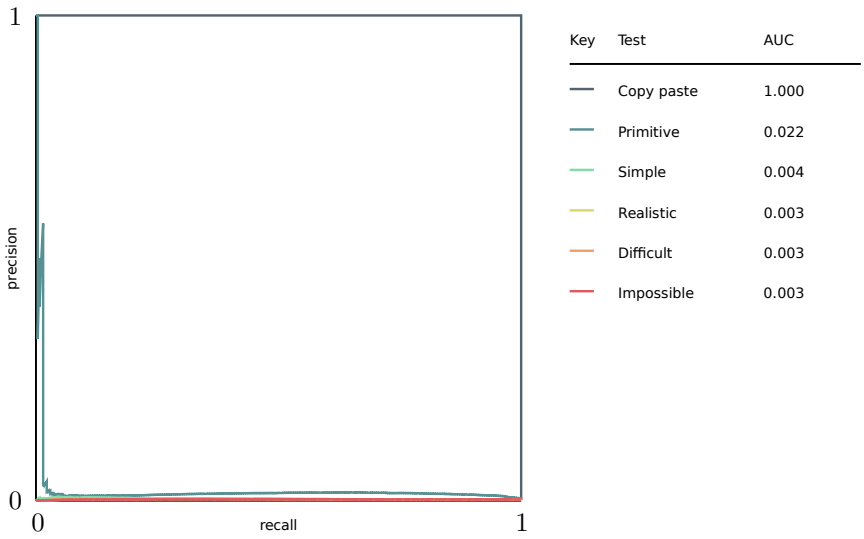


Figure 4.12: Precision recall curve of Text::Similarity on the Complex C++ corpus

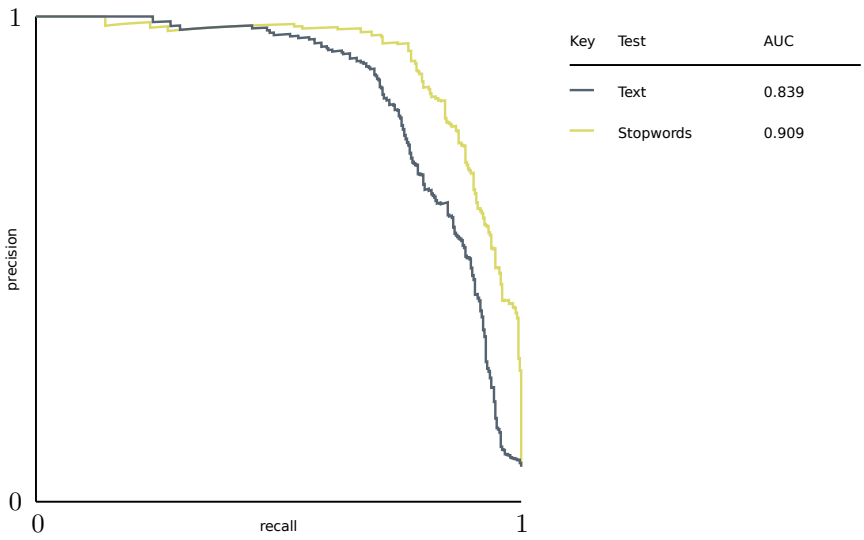


Figure 4.13: Precision recall curve of Text::Similarity on the Corpus of short plagiarized answers

Chapter 5

Our Solution

Our solution is a fast, local, language independent detector, with improved performance compared with Text::Similarity on most of our tests.

5.1 Our Solution

Our detector uses enhanced suffix arrays to detect overlaps in a collection of documents and scores their importance using document frequency. It uses a simple heuristic approach to exclude comments and user defined identifiers from source code based on their frequency in the corpus.

5.1.1 The Algorithm

Our algorithm detects *matches* in *documents*. For our purposes a *document* is synonymous with a string of tokens from a finite alphabet, while a *match* is a set of equivalent token substrings. We use the simplification that any two substrings are equivalent only if all of their tokens are equivalent and in the same order. We deffer the responsibility of defining tokens, equivalence of tokens, and the normalization of their order to language-dependent preprocessors. Nevertheless, even without reordering and using token equality based on their hashes, gives reasonable results.

With our simplification a set of substrings can form a match only if their first tokens are equivalent. Therefore, if we lexicographically sort all suffixes of all documents we need only compare the prefixes of consecutive suffixes to find every possible match. A data structure called *Suffix Array* (SA) can perform exactly this sorting in linear time [12, 13]. This array can be enhanced with information of the length of the longest prefix of every pair of consecutive suffixes in linear time [3] as well. Additionally, for source retrieval we need information about the document of origin of each suffix this is a task of the Document Array (DA).

For a given match containing substrings $\{(n_1, m_1); (n_2, m_2); \ldots\}$ matches containing $(n_1, m_1 - 1)$ down to single token matches (n_1, n_1) also exist. These token substrings can be divided into sets, named *substring classes*, based on the number of documents they occur in (document frequency) [20] in $\mathcal{O}(n \log n)$ time.

We can use the information of document frequency to automatically exclude common substrings, such as obvious parts of the solutions or assignment templates (section 2.3). This information is also useful for scaling the importance of matches, or excluding them completely for speed, since more common substrings are less likely to be plagiarisms.

Preprocessing

The first stage (listing 8) is a preprocessing step. We use very simple preprocessing based on the hashing trick [19] which should be sufficiently general for most languages. We simply tokenize all files into words, keeping track of their offsets, hash their values, and concatenate them.

This step could also be performed based on *mutual information* and *ridf*. Such approach would then be applicable even to Japanese with a technique similar to Kashioka et al. [8]. The cost for generality would, however, be slower detection, complex preprocessing, and higher memory requirements.

We recommend the use of more complex preprocessors as this is likely to greatly improve detection performance. Such preprocessors can for example reorder binary statements so that constants are on the left-hand side, or use stemming, or replace words with a synonym with the lowest alphabetical order. Other features, such as stop word exclusion, are to some extent performed automatically by successive steps of our algorithm.

```
type Token = { value: int; offset: unsigned int }

let preprocess (files: string array) =

  (* Tokenize each file                                    *)
  (* Lower-case each token                                 *)
  (* Apply NFKC Unicode normalization                      *)
  (* Use the hashing trick                                 *)
  (* Join the tokens using a unique token                  *)

  let features =
    tokenize files
    |> lowercase
    |> normalize "NFKC"
    |> vectorize (hash mod N)
  let concatenated = Join features by unique_token

  return concatenated
```

Listing 8: Pseudocode for preprocessing a corpus.

Exclusion of comments and identifiers

The second stage (listing 9) of preprocessing is used only for programming languages. We use a simple heuristic based on document frequency of substrings to determine which parts of files are likely to be names and comments. After

this step, the only tokens remaining in the corpus are those that describe the general structure. These tokens include `for` loops and `return` statements.

We use an algorithm for iterating over suffix classes and computing their document frequencies described by Yamamoto and Church [20]. The general pseudocode is as follows.

```
let exclude (tokens: Token array) =

  (* Use an enhanced Suffix Array Construction Algorithm      *)
  (* sa : Suffix Array                                        *)
  (* da : Document Array                                      *)
  (* lcp: Longest Common Prefix Array                         *)

  let sa, da, lcp = Enhanced.SACA (tokens.values)

  (* Substring classes can overlap                            *)
  (* We create a mask array and use it to filter the corpus   *)

  let mask = Array of false with length of tokens.length

  (* Keep only those tokens, that are                         *)
  (* sufficiently frequent to not be comments or identifiers  *)

  for suffix_class<i, j> in classes of sa
    if document frequency of suffix_class > Threshold
    then for token in tokens of suffix_class
           mask[token.offset] <- true

  let tokens_to_keep = keep tokens where mask[index] is true

  return tokens_to_keep
```

Listing 9: Approximate algorithm for heuristically excluding user defined identifiers and source comments.

An inverse of this step, which would exclude common tokens, could be used to improve detection speed on natural language by removing likely stop-words.

Detecting matches

In the third stage (listing 10) we recompute the suffix array for the filtered corpus and iterate over all suffixes. When a suffix occurs in at most *Threshold* files we add it to a collection of matches for that file group.

Output

The final step scores the importance of all matched suffixes using $length/df^2$. Since we have the array containing offsets for each token we can also output where the matches occur by setting the `start = offsets[sa[i]]` and setting

```
let detect (filtered: Token array) =

  (* Use an enhanced Suffix Array Construction Algorithm     *)
  (* sa : Suffix Array                                       *)
  (* da : Document Array                                     *)
  (* lcp: Longest Common Prefix Array                        *)

  let sa, da, lcp = Enhanced.SACA (filtered.values)

  (* Keep only those matches                                 *)
  (* which occur in a small number of files                  *)

  let similarities: dictionary<Document * Document, match> =
    dictionary ()

  for suffix_class<i, j> in classes sa
    if document frequency of suffix_class < Threshold
    then similarities[documents i j] += suffix_class

  return similarities
```

Listing 10: Pseudocode for detecting suspicious substrings.

the `end = offsets[sa[i] + length]` (where *offsets* is the offset array, *sa* is the suffix array, i is the start of the suffix, and *length* is the length of the suffix).

Used implementations

The implementation of the Suffix Array Construction Algorithm (SACA) for collections gSACA-K is described in Louza, Gog, and Telles [12]. This implementation also builds the Document array and the LCP array. But the exact algorithm is not important and any linear time SACA implementation would work well. For the purposes of plagiarism detection a fast LCP construction algorithm is necessary. Fischer [3] shows that while even a naive algorithm is fast on most types of data, it is slow on texts with long repeated phrases such as English.

5.2 Our Implementation

Our implementation, along with the visualization interface, can be obtained from GitLab. It is written in F# [18] and runs on Windows using the .NET Framework and on GNU/Linux using Mono or .NET Core.

5.3 Our results

Our algorithm finds matches based on document frequency and is able to automatically exclude intentionally shared code from comparison which largely obviates the need to provide a hand-written assignment template. Setting the

appropriate granularity of detection (Section 2.4) is also automatically solved by scoring matches based on their document frequency. Our solution is thus mostly invariant to different language complexities. The speed of detection has also improved greatly compared with Text::Similarity from around 10 minutes for 300 files to less than 3 seconds.

In spite of the fact that our simple algorithm does not use any prior knowledge about languages, it was able to perform at least as well as purpose built detectors (Compare JPlag 4.4 versus our 5.2). As a result it also solves the issue of limited language support.

Simple C++ (Figure 5.1) Our detector performed well on all simple datasets. This suggests that our heuristic solution for excluding identifiers and source comments may be a viable alternative to creating bespoke lexical analyzers for each language.

Complex C++ (Figure 5.2) A large percentage of files was detected in all complex dataset. Unlike with language-specific detectors, the *Primitive* dataset (comprising renaming and adding comments) had some limited effect.

Python (Figure 5.3) The detection performance for the simple changes was similar to the simple C++ dataset. The *style* and *full* datasets still contained numerous similar substrings which our algorithm was able to detect.

Text (Figure 5.4) Due to the large number of almost exact copies of each file, we had to turn off the document frequency scaling (We had to set the same option for Moss). With this setting turned off, our detector performed well and detected most plagiarized files, with only a few false positives until high recall values.

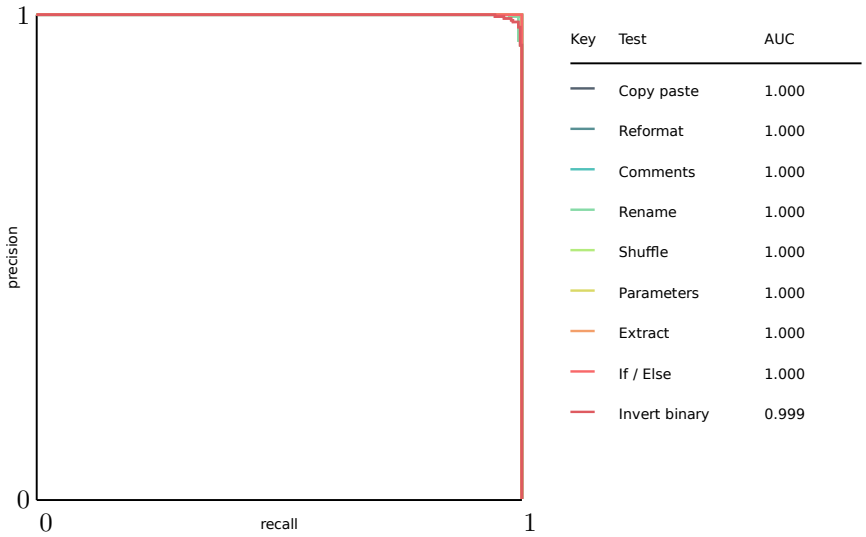


Figure 5.1: Precision recall curve of our detector on the Simple C++ corpus

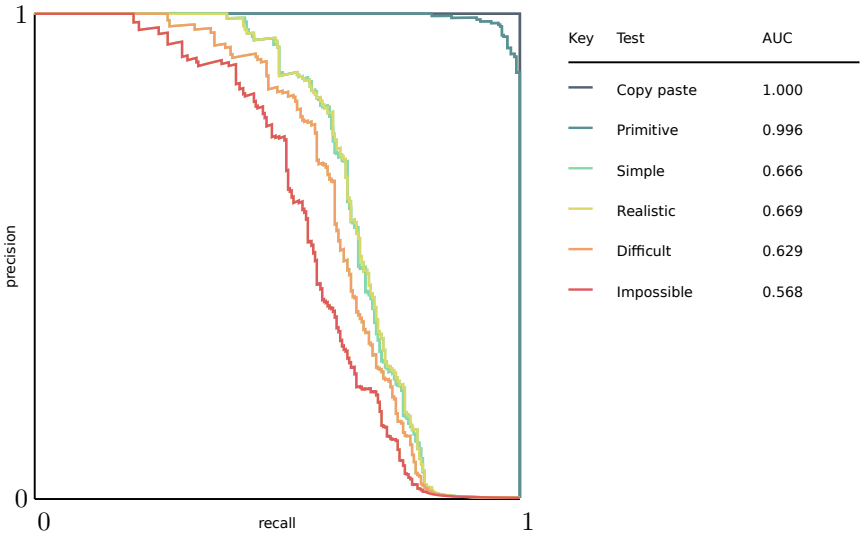


Figure 5.2: Precision recall curve of our detector on the Complex C++ corpus

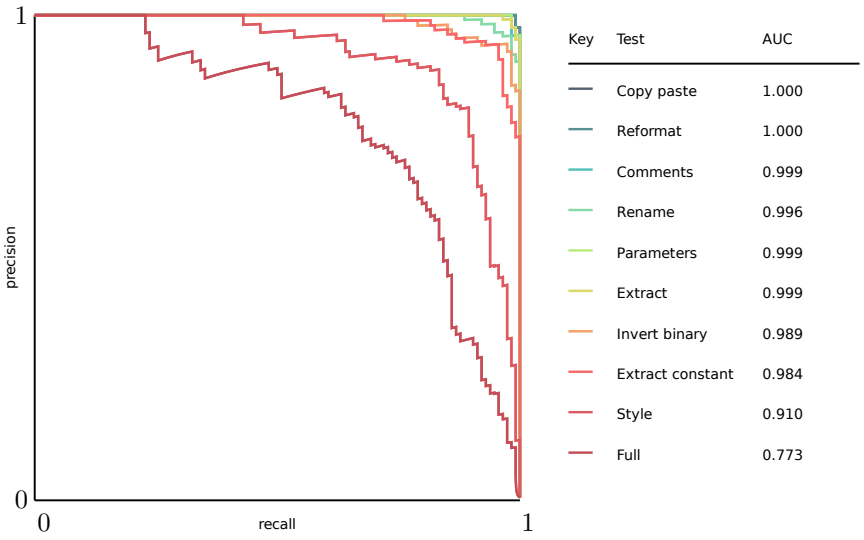


Figure 5.3: Precision recall curve of our detector on the Python corpus

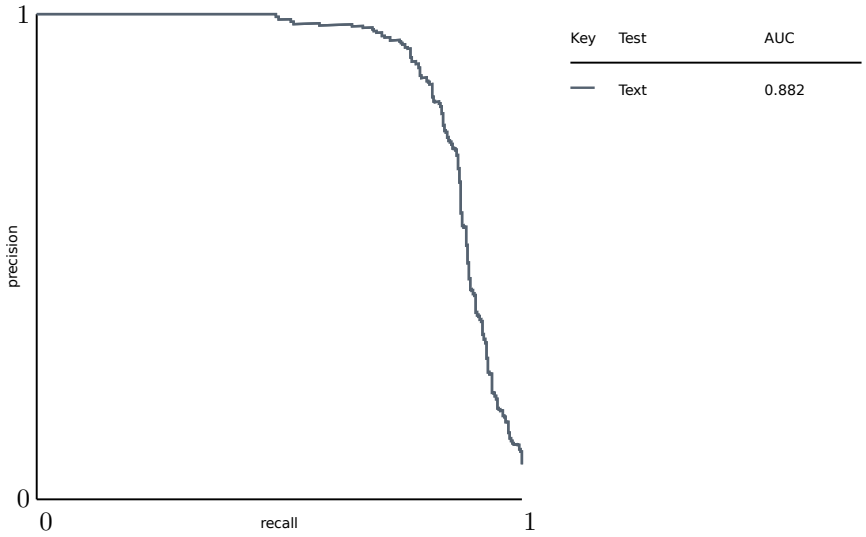


Figure 5.4: Precision recall curve of our detector on the Corpus of short plagiarized answers

5.4 Visualization

We have built two visualization tools on top of the detection output.

5.4.1 Match plot

This tool plots all matched sequences among two files. It is useful for quickly judging the validity of detection. Example output is shown in figure 5.5.

5.4.2 Side-by-Side comparison

We have also built an experimental tool to visualize the matches between two files. The tool shows both of the files side by side and color-codes each of the matched sequences. These sequences can be double-clicked to scroll into view the sections in both files. Example output is shown in figure 5.6.

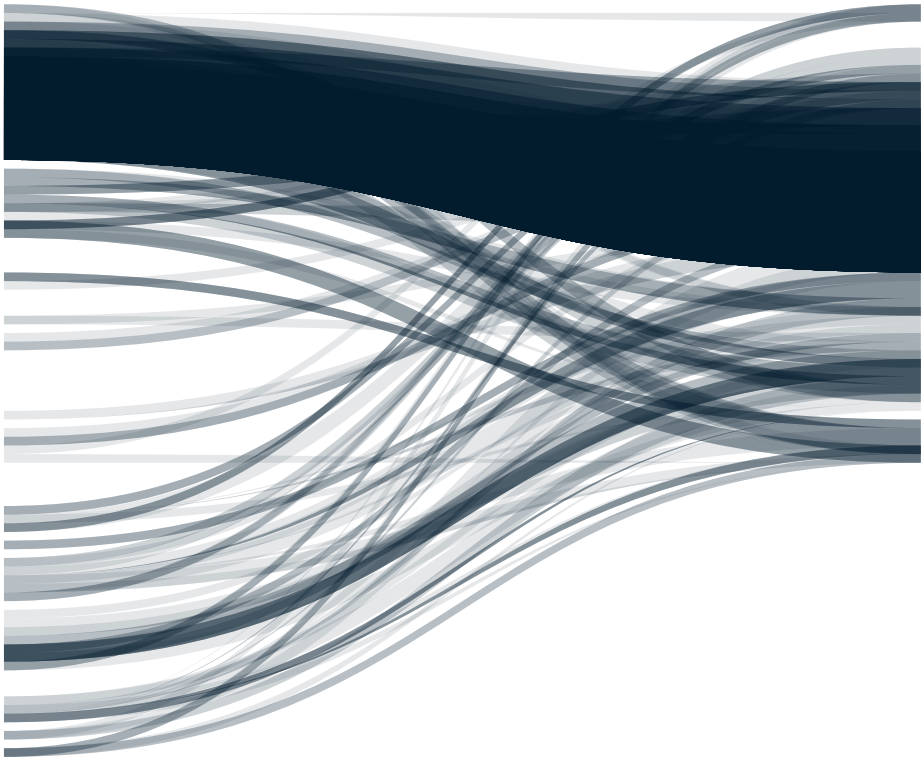


Figure 5.5: Overview image showing that the beginning of the two files is plagiarized.

1. In object-oriented programming, inheritance is a way to form new classes (instances of which are called objects) using classes that have already been defined. The inheritance concept was invented in 1967 for Simula. The new classes, known as derived classes, take over (or inherit) attributes and behavior of the pre-existing classes, which are referred to as base classes (or ancestor classes). It is intended to help reuse existing code with little or no modification. Inheritance provides the support for representation by categorization in computer languages. Categorization is a powerful mechanism number of information processing, crucial to human learning by means of generalization (what is known about specific entities is applied to a wider group given a belongs relation can be established) and cognitive economy (less information needs to be stored about each specific entity, only its particularities). Inheritance is also sometimes called generalization, because the is-a relationships represent a hierarchy between classes of objects. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple is-a fruit), apples may naturally inherit all the properties common to all fruit, such as being a fleshy container for the seed of a plant. An advantage of inheritance is that modules with sufficiently similar interfaces can share a lot of code, reducing the complexity of the program. Inheritance therefore has another view, a dual, called polymorphism, which describes many pieces of code being controlled by shared control code. Inheritance is typically accomplished either by overriding (replacing) one or more methods exposed by ancestor, or by adding new methods to those exposed by an ancestor. Complex inheritance, or inheritance used within a design that is not sufficiently mature, may lead to the Yo-yo problem.

1. In object-oriented programming, inheritance is a way to form new classes (instances of which are called objects) using classes that have already been defined. Inheritance is also sometimes called generalization, because the is-a relationships represent a hierarchy between classes of objects. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple is-a fruit), apples may naturally inherit all the properties common to all ruit, such as being a fleshy container for the seed of a plant. Inheritance is typically accomplished either by overriding (replacing) one or more methods exposed by ancestor, or by adding new methods to those exposed by an ancestor.

Figure 5.6: Side by side view of plagiarized files including color coded matches.

Chapter 6

Conclusions

We have created a simple, fast and general algorithm for plagiarism detection in most programming and natural languages (5.1.1). This algorithm can easily be modified to improve its performance for specific languages by implementing custom preprocessors. Additionally, we have implemented the algorithm in F# [18] and compared its performance to other available detectors in section 5.3. Compared with the current solution used in BRUTE, its performance is significantly improved for all our source code datasets, while only slightly worse on the Corpus of Plagiarised Short Answers. However, in this special case, we had to turn off document frequency scaling, which is the most powerful feature of our detector.

6.1 Future work

We believe the performance of our detector can be improved by using more complex preprocessors, while still remaining general for use on most languages. Although our results are encouraging, it was tested only on a small dataset and the results are not comparable to the results from the source retrieval task of the PAN competition [9]. Extending our software with an algorithm to filter candidate files, would allow it to work on much larger datasets. This would enable us to evaluate the performance of our solution in contrast to the state-of-the art.

Chapter 7

Appendix - Testing in Detail

This chapter completes the important details of testing which were too detailed to be included in the Source code and Results chapters.

7.1 Description of Transformations

7.1.1 Simple Transformations on C++ in detail

Add comments Twenty lines of randomly generated ASCII comments were interspersed throughout the file. Each line is at most 120 characters long.

Reformat code Every space preceding an operator, a bracket, or a comma, and spaces appearing inside brackets were removed.

Rename identifiers All user-defined identifiers, including define macros, were renamed to sanitized versions of random words from the Aspell dictionary.

Add assignment template The same 40 line function and 2 variables were appended to every file, simulating an assignment template, or a part of code intentionally shared during a lecture. This code was also saved as a file and passed to every detector, which supports shared-code exclusion.

Add original code Two roughly equal files were concatenated into one, simulating a situation where a student finds themselves unable to solve the assignment completely and decides to plagiarize the remaining code.

Shuffle functions All function definitions were randomly permuted throughout the file, with no regard to the functional dependency graph.

Reverse function parameters The order of parameters of every user-defined function was reversed.

Extract methods Every top-level statement which itself contains at least one compound statement was extracted from a method into a smaller method. Each of these extracted parts was then placed before the original method. An attempt was made to infer appropriate function headers.

Invert if-else branches Branches of if-else statements were switched and an ! operator was prepended to the condition.

Invert binary operations Operands of binary operations were switched and the operators were replaced to maintain the original behavior.

7.1.2 Transformations on Python in detail

Copy paste The entire solution is copied and pasted.

Reformat Each file was reformatted using autopep8 with the following arguments `autopep8 --aggressive --aggressive --in-place file.py`

Comments Randomly generated ASCII comments with an average length of 8 lines were added to each file.

Rename All user-defined identifiers were renamed using a Caesar cipher.

Parameters The order of function parameters was reversed and default values were inserted where necessary.

Extract All For loops, While loops, and If/Else statements were repeatedly extracted into functions, until further extraction was not possible or all functions contained fewer than 2 top-level statements.

Invert binary Operands of binary expressions were switched and the operators were replaced to maintain the original behavior.

Extract constant Every string constant and every numerical constant with at least 2 characters was extracted into global scope. This refactoring is usually called Introduce Explaining Variable.

Full All the previous transformations were applied in turn on the copied and pasted solutions.

Style Combination of the Comments, Rename, Extract constant, and Reformat transformations.

7.2 Testing - Parameter Setting

The minimal sequence length for the natural-language dataset was set to 3 tokens, while for the source code datasets the length was 10 tokens. The explanation for why this is needed is in section 2.4.

Due to the number of almost exact copies of each file in the natural-language dataset, detectors would identify the copy-pasted sections as a natural feature of the dataset and exclude such matches. As a result, any detectors which support automatic assignment template detection had to have this feature turned off.

7.2.1 CPD

All tests were performed with the following parameters only changing the language and minimum-tokens appropriately.

```
./run.sh cpd --format xml     \
             --language text   \
             --minimum-tokens 3 \
             --files Files/*
```

7.2.2 JPlag

```
java -jar jplag-2.11.8-SNAPSHOT-jar-with-dependencies.jar \
                -l text -t 3 Files/
```

7.2.3 Moss

Tests on the natural-language dataset were performed setting the maximum number of times a given passage may appear before it is ignored to 1000000 as suggested in the usage instructions to keep all matches. This setting was needed due to the large number of copies of each file. For other datasets this parameter was not used.

```
./moss -l ascii -m 1000000 -n 1000 Dataset/*
```

7.2.4 Sherlock

All tests were performed with the following parameters. Sherlock is language-independent and only the sequence length needed to be changed for each dataset.

```
./sherlock -t 0 -z 0 -n 3 Files/*
```

Sim

```
./sim_text -p -e -s -T -r3 Files/*
```

7.2.5 Text::similarity

The sample stopword list included int the downloaded perl module was used for the natural-language dataset. Stop-word lists available on-line worked just as well.

```
perl ./text_similarity.pl --type=Text::Similarity::Overlaps \
                          --stoplist=../samples/stoplist.txt \
                          Files/*
```

7.2.6 Our detector

Source Code

```
mono detector.exe --data=code --output=scores \
                  $(find $FILES -type f)
```

Text

The same setting of `--group` and `-m` for *Moss* was used. However, in our case, setting this to around 20 would significantly improve our detection performance (To around 0.92 AUC).

```
mono detector.exe --data=text --frequency=false --output=scores \
                  --group=1000000 $(find $FILES -type f)
```

7.3 Figures

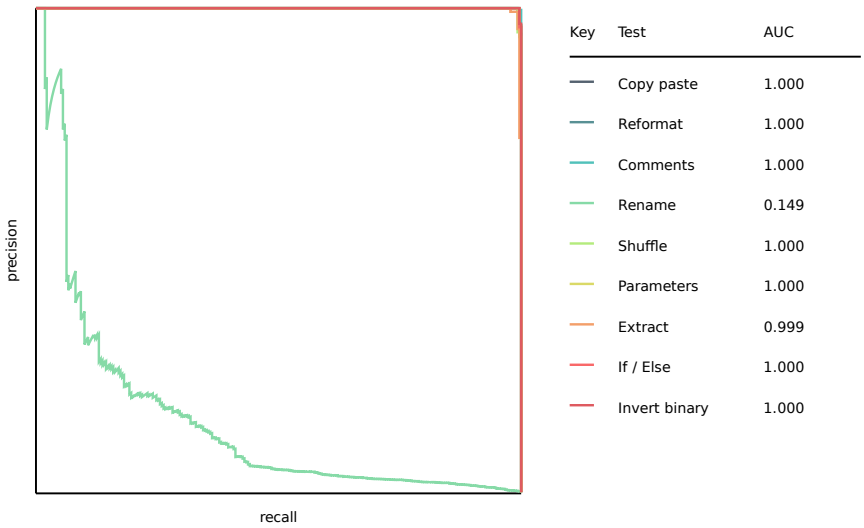


Figure 7.1: Precision recall curve of CPD on the Simple C++ corpus

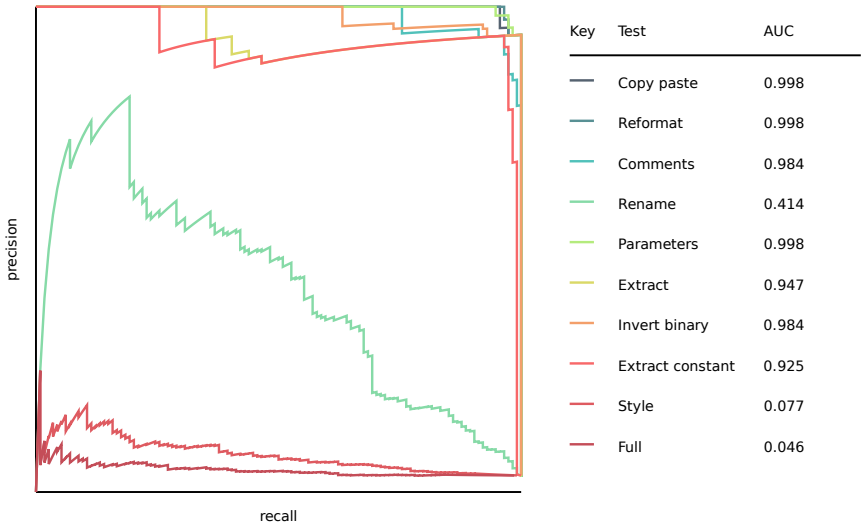


Figure 7.2: Precision recall curve of CPD on the Python corpus

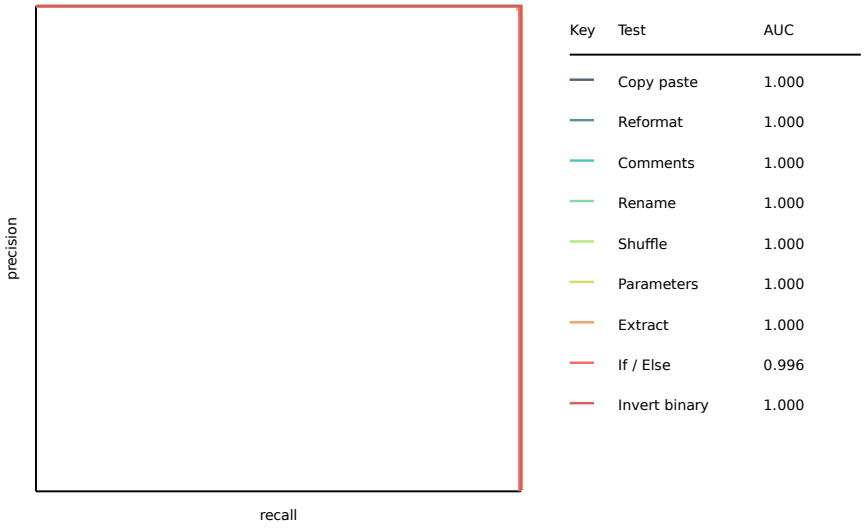


Figure 7.3: Precision recall curve of JPlag on the Simple C++ corpus

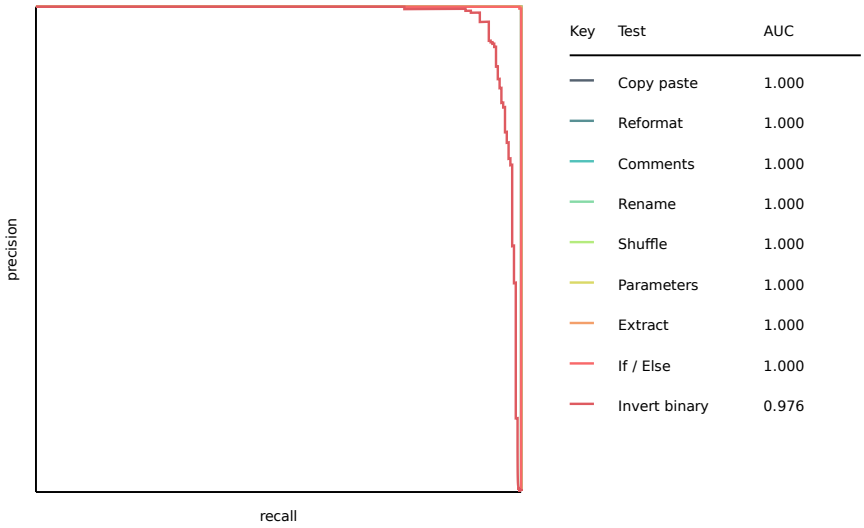


Figure 7.4: Precision recall curve of Moss on the Simple C++ corpus

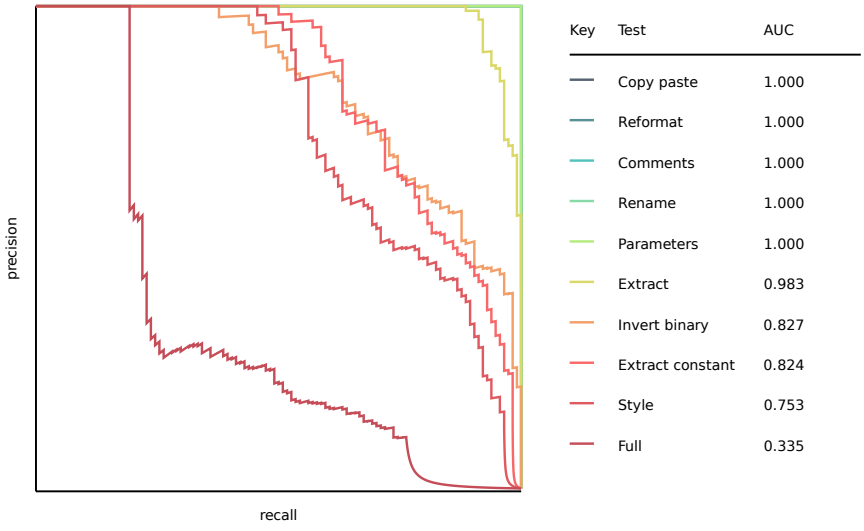


Figure 7.5: Precision recall curve of Moss on the Python corpus

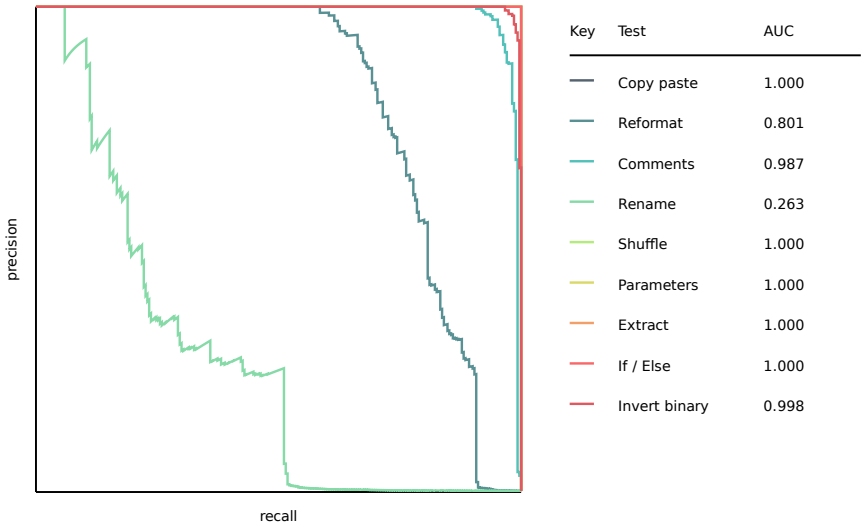


Figure 7.6: Precision recall curve of Sherlock on the Simple C++ corpus

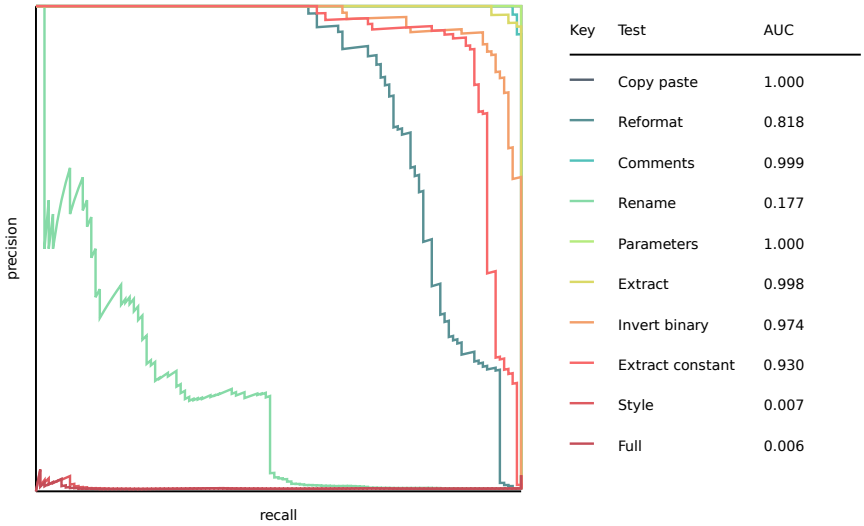


Figure 7.7: Precision recall curve of Sherlock on the Python corpus

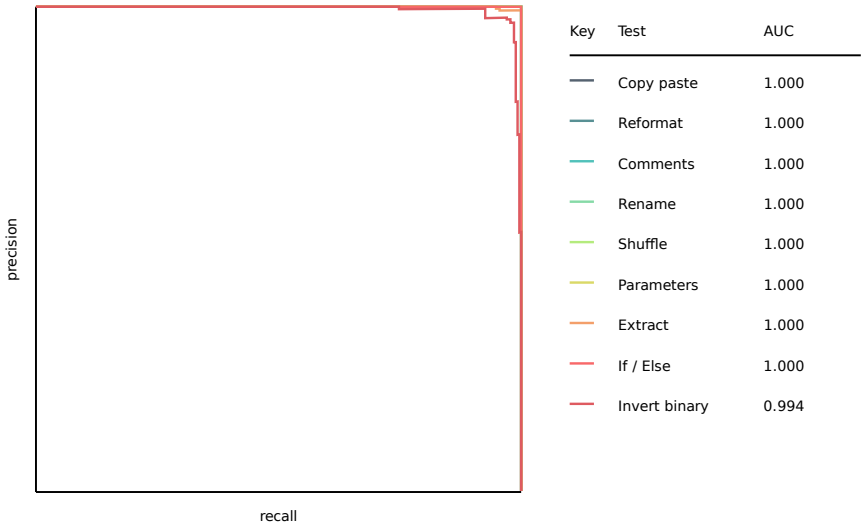


Figure 7.8: Precision recall curve of Sim on the Simple C++ corpus

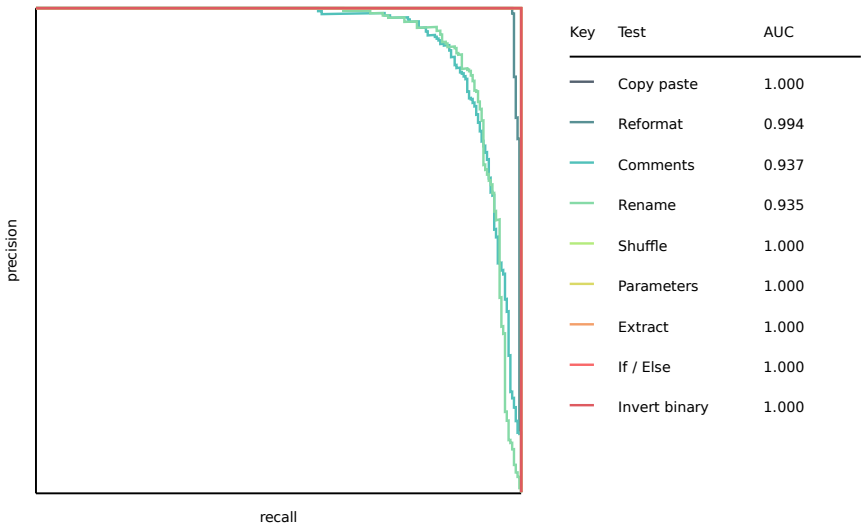


Figure 7.9: Precision recall curve of Text::Similarity on the Simple C++ corpus

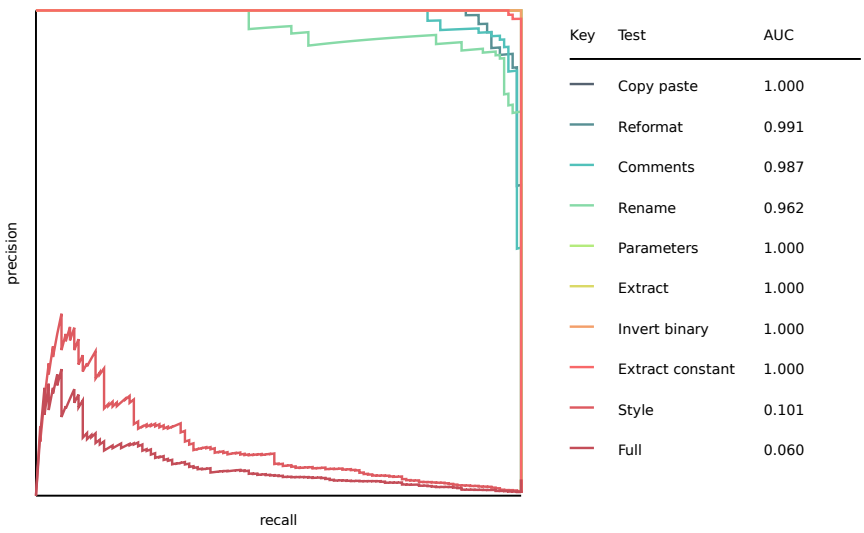


Figure 7.10: Precision recall curve of Text::Similarity on the Python corpus

Bibliography

[1] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. "Plaggie: GNU-licensed Source Code Plagiarism Detection Engine for Java Exercises". In: *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*. Baltic Sea '06. Uppsala, Sweden: ACM, 2006, pp. 141–142. DOI: 10.1145/1315803.1315831. URL: http://doi.acm.org/10.1145/1315803.1315831.

[2] Paul Clough and Mark Stevenson. "Developing a corpus of plagiarised short answers". In: *Language Resources and Evaluation* 45.1 (Mar. 2011), pp. 5–24. ISSN: 1574-0218. DOI: 10.1007/s10579-009-9112-1. URL: https://doi.org/10.1007/s10579-009-9112-1.

[3] Johannes Fischer. "Inducing the LCP-Array". In: *Algorithms and Data Structures: 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*. Ed. by Frank Dehne, John Iacono, and Jörg-Rüdiger Sack. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 374–385. ISBN: 978-3-642-22300-6. DOI: 10.1007/978-3-642-22300-6_32. URL: https://doi.org/10.1007/978-3-642-22300-6_32.

[4] Enrique Flores et al. "On the Detection of SOurce COde Re-use". In: *Proceedings of the Forum for Information Retrieval Evaluation*. FIRE '14. Bangalore, India: ACM, 2015, pp. 21–30. ISBN: 978-1-4503-3755-7. DOI: 10.1145/2824864.2824878. URL: http://doi.acm.org/10.1145/2824864.2824878.

[5] Dick Grune and Matty Huntjens. "Het detecteren van kopie:en bij informatica-practica". In: *Informatie* (Nov. 1989).

[6] Jurriaan Hage, Peter Rademaker, and Nikè van Vugt. *A comparison of plagiarism detection tools*. Tech. rep. UU-CS-2010-015. Department of Information and Computing Sciences, Utrecht University, 2010.

[7] Institute for Program Structures and Data Organization. *JPlag*. Version v2.11.9-SNAPSHOT. Dec. 20, 2017. URL: https://jplag.ipd.kit.edu/.

[8] Hideki Kashioka et al. "Use of mutual information based character clusters in dictionary-less morphological analysis of Japanese". In: *In Proceedings of COLING-ACL '98*. 1998, pp. 658–662.

[9] Leilei Kong et al. "Source Retrieval and Text Alignment Corpus Construction for Plagiarism Detection—Notebook for PAN at CLEF 2015". In: *CLEF 2015 Evaluation Labs and Workshop – Working Notes Papers, 8-11 September, Toulouse, France*. Ed. by Linda Cappellato et al. CEUR-WS.org, Sept. 2015.

[10] J. Levine. *flex & bison: Text Processing Tools*. O'Reilly Media, 2009. ISBN: 9781449379278. URL: https://books.google.cz/books?id=3Sr1V5J9%5C_qMC.

[11] Loki. *Sherlock*. Dec. 20, 2017. URL: http://www.cs.usyd.edu.au/~scilect/sherlock/.

[12] Felipe Alves da Louza, Simon Gog, and Guilherme P. Telles. "Inducing enhanced suffix arrays for string collections". In: *Theor. Comput. Sci.* 678 (2017), pp. 22–39. DOI: 10.1016/j.tcs.2017.03.039. URL: http://www.sciencedirect.com/science/article/pii/S0304397517302621.

[13] Ge Nong, Sen Zhang, and Daricks Wai Hong Chan. "Linear Suffix Array Construction by Almost Pure Induced-Sorting". In: (Mar. 2009), pp. 193–202.

[14] *PMD Applied*. Centennial Books, 2005. ISBN: 0976221411. URL: https://www.amazon.com/PMD-Applied/dp/0976221411?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0976221411.

[15] Martin Potthast et al. "Overview of the 2nd International Competition on Plagiarism Detection". In: *Working Notes Papers of the CLEF 2010 Evaluation Labs*. Ed. by Martin Braschler, Donna Harman, and Emanuele Pianta. Sept. 2010. ISBN: 978-88-904810-2-4. URL: http://www.clef-initiative.eu/publication/working-notes.

[16] Takaya Saito and Marc Rehmsmeier. "The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets". In: *PLOS ONE* 10.3 (Mar. 2015), pp. 1–21. DOI: 10.1371/journal.pone.0118432. URL: https://doi.org/10.1371/journal.pone.0118432.

[17] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. "Winnowing: Local Algorithms for Document Fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 76–85. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872770. URL: http://doi.acm.org/10.1145/872757.872770.

[18] Don Syme et al. "The F# 4.0 Language Specification". In: (2005).

[19] Kilian Weinberger et al. "Feature Hashing for Large Scale Multitask Learning". In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. Montreal, Quebec, Canada: ACM, 2009, pp. 1113–1120. ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553516. URL: http://doi.acm.org/10.1145/1553374.1553516.

[20] Mikio Yamamoto and Kenneth W. Church. "Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus". In: *Comput. Linguist.* 27.1 (Mar. 2001), pp. 1–30. ISSN: 0891-2017. DOI: 10.1162/089120101300346787. URL: https://doi.org/10.1162/089120101300346787.

Chapter 8

Přílohy

Součástí této práce je:

Implementace Implementace algoritmu pro detekci plagiarismu.

Zobrazovací program Program pro zobrazení výstupu detekce plagiarismu.