Code Plagiarism Detector

Om Dalwadi, Ravinder Kaur, Yug Shah

13th December 2022

Contents

List of Figures

1: Abstract

The Code Plagiarism Detector is a tool that can be used to identify instances of code plagiarism in programming assignments. In this project, we address a specific issue of plagiarism in programming assignments for Computer Science courses at the University of Regina. The current method of using Turnitin software for checking code files for plagiarism is inadequate, leaving it up to instructors to manually compare submissions or introduce assessment criteria. This is a time-consuming task, particularly during busy semesters. This project aims to develop a robust and efficient algorithm for detecting code plagiarism, which can be integrated into existing plagiarism detection software or used as a standalone tool. The project uses Levenshtein distance to calculate the edit distance between two files, and then returns the similarity score between these files. We present our research on existing literature and approaches, as well as the methodology and tech stack used in our project. The results and conclusions of our work are also discussed, as well as potential future work. Overall, the Code Plagiarism Detector is a valuable tool for educators and institutions looking to detect and prevent code plagiarism.

2: Introduction and Problem Statement

In our research, we have discovered that there are problems with the Turnitin software when it is used in Computer Science courses at the University of Regina for checking code file submissions for plagiarism. These issues make it difficult for instructors to maintain the integrity of programming assignments, forcing them to either manually compare all submissions or introduce some form of assessment criteria. One example of this is requiring students to include their ID as an input in a function they must implement. The time-sensitive nature of checking these assignments, particularly during busy semesters, makes this a difficult task. Our group project for the CS 375: Database and Information Retrieval course aims to solve this problem by developing a solution to help instructors efficiently detect plagiarism in programming assignments. The rest of the paper is organized as follows: Section 3 discusses the existing literature and references used in our research, Section 4 presents the project description and overview, Section 5 presents the methodology including requirement analysis, ERD's, database normalization, and the tech stack used, Section 6 contains the results and conclusions, and Section 7 presents future work.

3: Literature Review

Here, we will discuss the existing literature and implementations of code plagiarism detectors, including the relevant approaches and algorithms that have been developed to address this problem. During our research, we found a number of implementations of a code plagiarism detector aimed at the use of lexical and syntactical analysis of the text to identify matching code fragments and to improve the accuracy of the detection. The following is a review of the different algorithms and implementations we have encountered so far.

Plagiarism is defined as the use of text, ideas, and results of someone else's work without proper attribution to its source [1]. Plagiarism detection systems can be divided into two main categories: intrinsic and extrinsic. Intrinsic systems rely on the idea that each author has a unique writing style In contrast, extrinsic systems compare a document to a reference collection of documents to identify instances of copied or paraphrased text. [5] [2]. To plagiarize a program, most people rewrite the original version, including part or all of its interface or contents, to avoid easy human identification. If the differences between codes are compared manually, it will be very time-consuming. As the coding scale becomes larger and more complex, the success rate of identification also diminishes.

Jaccard distance, also known as Jaccard similarity coefficient, is a measure of the similarity between two sets. In other words, it is a measure of how many elements two sets have in common, compared to the total number of elements in both sets. This metric is commonly used in natural language processing, data mining, and other

applications where the similarity between sets needs to be measured. For example, it can be used to compare the content of two documents and determine how similar they are. A similar approach is used by document fingerprinting algorithms to identify and authenticate documents. By creating a unique fingerprint for each document, it is possible to quickly and easily compare the content of two documents to determine whether they are the same or not. If the fingerprints of two documents match, it is highly likely that the documents are the same or substantially similar. This allows plagiarism to be detected and prevented, as it is difficult for someone to alter a document in such a way that its fingerprint remains unchanged. This approach has the advantage of being fast and efficient, as it does not require the full text of the documents to be compared. It is also robust against simple modifications such as changing the formatting or rearranging the words.

4: Project Overview

Due to the limited scope of our project, we decided to implement a simple algorithm for plagiarism detector. Levenshtein distance is a string metric that measures the difference between two sequences by counting the minimum number of single-character edits (insertions, deletions, or substitutions) that are required to transform one sequence into the other. It is also known as Edit Distance, and is commonly used in natural language processing, spell checking, and other applications where the similarity between two strings needs to be measured. This algorithm can then be used to compute the difference between two strings, returning the number of edits needed to transform one string into the other. This can then be used for detecting plagiarism in written work, as well as for identifying misspellings in text.

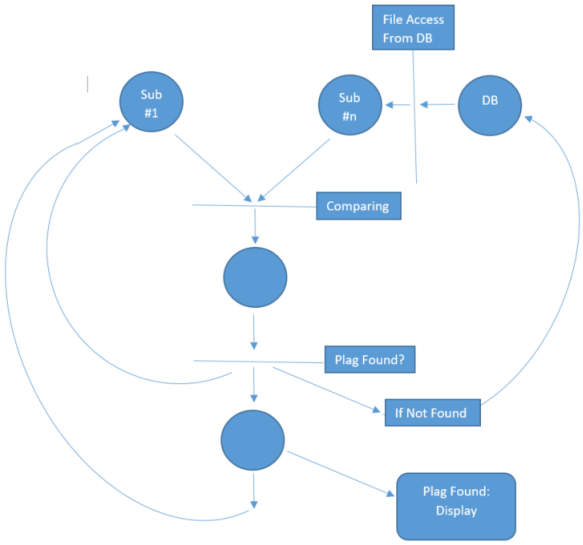

Figure 1: Petri Net Design

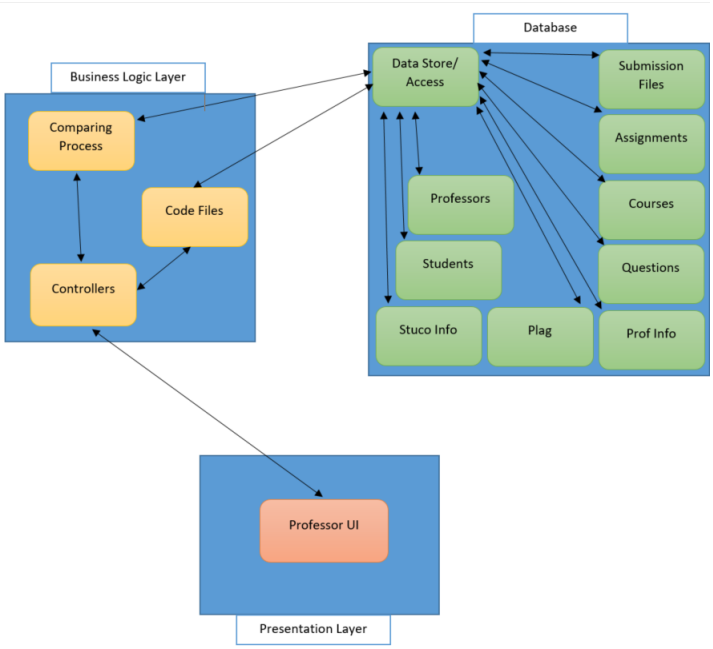

Figure 2: Three Layer Architecture

We have developed a web application that uses the Levenshtein distance calculator for comparing two code files and generate a similarity score based on this comparison. The course instructor/professor will be able to upload code files submitted by students in order to store them in the database. Whenever required, they can compare any file with the other files in the database to check for instances of plagiarism.

5: Methodology

5.1 Requirement Analysis

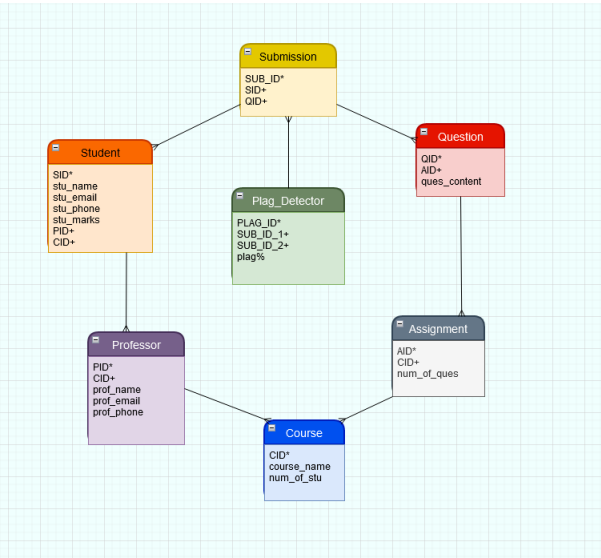

Figure 3: Entity-Relation Diagram (before normalization)

During this phase, we analysed the requirements of what data needs to be stored in order to build a working application. Initially, we created 7 different tables to store all the important information in the database. Figure 3 shows the original ER diagram generated during our first design stint. While building the application, we realized that we needed a few more tables to store the relevant information and to make it easy for the web application to retrieve the data. Figure 4 shows the new ERD diagram which consists of 9 tables interconnected through foreign keys. Our database model has a few assumptions as listed below:

- A professor can only teach a single course.

- A course can be taught by multiple professors simultaneously
- A student can register for multiple classes under different professors simultaneously
- A professor can have multiple assignments and each assignment can have multiple questions
- There are no sub-questions to a given question

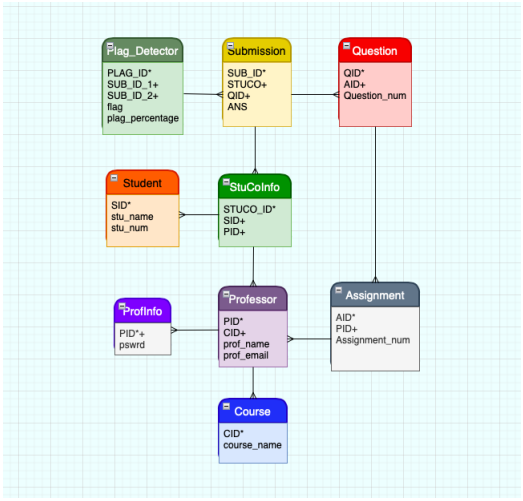

Figure 4: Entity-Relation Diagram (after normalization)

As a result of these assumptions, we have updated our previous ER diagram. You can notice that the foreign key for most of the tables has shifted from CID to PID since the professors table plays an important role in our database design. The following is a breakdown of all the tables and an overview of the data being stored in them:

- **Course[primary key: CID]** includes the names of the existing courses.
- **Professor[primary key: PID]** stores the name and email of professors.

- **ProfInfo[primary key: Prof_id]** contains the login credentials for instructors/professors who have access to the plagiarism detector.
- **Student[primary key: SID]** stores the identifying information for a student such as their names and student numbers.
- **StuCoInfo[primary key: STUCO_ID]** has emerged from the idea that a single student can have multiple courses at a time. It stores the SID and PID for the respective courses and links to the professor and student table.
- **Assignment[primary key: AID]** stores the assignment number for a specific course under a single professor.
- **Question[primary key: QID]** which contains the question numbers of their assignments.
- **Submission[primary key: SUB_ID]** stores the submissions by students, uploaded by the professor for a specific question.
- **Plag_Detector[primary key: PLAG_ID]** stores two submissions id's (foreign keys) and a unique PID. It also includes a flag (1 if plagiarised and 0 otherwise) and how much percentage of code is plagiarised.

5.2 Normalization

Our Current Tables are in 3NF as they satisfy all the above given conditions. The tables Course, Student, Professor, Assignment, Submission, Question, PlagDetector, ProfInfo and StuCoInfo consist of one primary key and every non-prime attribute of the table depends on it. In the case of the table StuCoInfo the column stumarks: let's call it Z, Z depends on the candidate keys: SID as X and PID as Y. We can say XY→Z as no partial and transitive dependency exists for Z. It fully depends on both XY. Thus it can be concluded that StuCoInfo also follows 3NF. The new assumption that was mentioned in the ERD sector also impacted the other tables such as the number of students which was previously in Course was now inserted in Professor which makes sense as each professor will have a unique number of students. The addition of the new attributes such as Assignment_num in Assignment and Question _num in Question makes it more precise and clear as it refers to the specific question of the specific assignment. It also reduces data redundancy. Figure 5 shows the old database model and Figure 6 shows the new model with some imaginary data values. We can see how the new model differs from the old after converting to 3NF:

Course

CID	course_name	num_of_stu
1	375	50

Professor

PID	CID	prof_name	prof_email	prof_phone
1	1	Henry	H@ur.ca	-

Student

SID	stu_name	stu_phone	stu_marks	PID	CID	stu_email
1	Albus	78372832	75	1	1	a@ur.ca
2	Lily	-	80	1	1	l@ur.ca

Submission

SUB_ID	SID	QID
1	1	1
2	2	1

Question

QID	AID	ques_content
1	1	whatever

Assignment

AID	CID	num_of_ques
1	1	3

Plag_Detector

PLAG_ID	SUB_ID_1	SUB_ID_2	plag%
1	1	2	15

Figure 5: Original Model with 7 tables

The original model, based on the assumption that a single professor can teach multiple courses at a time, had a few flaws which have been corrected in the following model.

Course

CID	course_name
1	375
2	350

Professor

PID	CID	Prof_id	prof_name	prof_email	prof_phone	num_of_stu
1	1	1	A	A@u.ca	-	20
2	1	2	B	B@u.ca	-	21
3	2	3	C	C@u.ca	-	29

ProfInfo

Prof_id	pswrd	prof_email
1	hello	A@u.ca
2	hello_world	B@u.ca
3	jhsjhs	C@u.ca

Assignment

AID	PID	Assignment_num	num_of_ques
1	1	1	-
2	1	2	-

Student

SID	stu_name	stu_email	stu_phone
1	X	X@u.ca	-
2	Y	Y@u.ca	-

StuCoInfo

STUCO_ID	SID	PID	stu_marks
1	1	1	98
2	1	3	99
3	2	1	100

Question

QID	AID	Question_num	ques_content
1	1	1	shahsa
2	1	2	kasjkajsaj

Submission

SUB_ID	STUCO	QID
1	1	1
2	3	1
3	3	2

Plag_Detector

PLAG__ID	SUB_ID_1	SUB_ID_2	flag	plag_percentage
1	1	2	1	13

Figure 6: Updated Model with 9 tables

6: Results & Conclusions

6.1 Implemented Functionalities

- The user can upload one or more code files to the plagiarism detector. The file input handles different programming languages for example, .txt, .cpp, .js and others. Only .pdf file types will not be accepted by this since Turnitin already has functionality for .pdf files.
- The detector will pre-process the uploaded code files to remove irrelevant information and to make them easier to compare. This includes standardizing indentation, removing irrelevant spaces, escaping single and double quotes, since they could cause errors when sending data over the server into the database.
- The detector can divide the code files into smaller chunks, such as individual functions or blocks of code. This allows for a more fine-grained comparison of the code.
- It calculates the Levenshtein distance between pairs of code chunks and between code files. One file is uploaded by the user, and the other comes from the answers stored in the database.
- Using this Levenshtein distance, a similarity score is generated where a small value of the distance indicates that the code files have higher similarity, while a high value for the distance indicates that they are different.
- It will then display the results to the user in a format that makes it easy to see

which code files are similar and which student had submitted that file.

- The user can then use these results to address any instances of code plagiarism in the code files. This can include reviewing the code manually if they desire, taking appropriate actions to report this, or any other action they desire.
- Overall, our product provides a user-friendly interface for comparing code files and identifying instances of plagiarism in programming assignments. Using advanced techniques to accurately detect similarities in the code, makes it a valuable tool for identifying plagiarism in code files.

6.2 Testing Database Queries

Following are screenshots of testing some[1] of the SQL queries developed to interact with the database:

```
-- selecting a row from the plag_detector table if the flag is set to 1

SELECT *
FROM Plag_Detector
WHERE flag = 1;
```

Result Grid | Filter Rows: | Edit:

PLAG_ID	SUB_ID_1	SUB_ID_2	flag	plag_percentage
1	1	2	1	65
NULL	NULL	NULL	NULL	NULL

Figure 7: Query to return all the instances of plagiarism

```
-- inserting submissions into the database

CREATE VIEW sid AS
SELECT SID
FROM Student
WHERE stu_num = 12345;

CREATE VIEW stucoid AS
SELECT STUCO_ID
FROM StuCoInfo, sid
WHERE stucoinfo.SID = sid.SID
AND PID = 1;

CREATE VIEW aid AS
SELECT AID
FROM Assignment
WHERE Assignment_num = 1
AND PID = 1;

CREATE VIEW qid AS
SELECT QID
FROM Question, aid
WHERE Question_num = 1
AND Question.AID = aid.AID;

INSERT INTO Submission(STUCO_ID, QID, ANS) VALUES (stucoid.STUCO_ID, qid.QID, 'hello');
```

Result

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell C

SUB_ID	STUCO_ID	QID	ANS
1	1	1	#include<iostream>#include<cstring>//tousestringcompare#include<string>...
2	3	3	#include<iostream>#include<cstring>//tousestringcompare#include<string>...
3	1	3	DROPTABLEIFEXISTSCourse;DROPTABLEIFEXISTSProfessor;DROPTABLEIFEXI...
4	1	1	hello
NULL	NULL	NULL	NULL

Figure 8: Query to insert submissions into the database

[1]In order to conform with the length of the project report, we decided to omit screenshots for a few queries and have only shown the important ones. We have submitted all the queries in a .sql file

```sql
-- retrieving all passwords, emails and names from the professor table,
-- given the email by joining table Professor and ProfInfo
SELECT p.prof_name, p.prof_email, pri.pswrd
FROM Professor p
JOIN ProfInfo pri
ON p.PID = pri.PID
WHERE p.prof_email = 'a@uregina.ca';
```

Result Grid | Filter Rows:

prof_name	prof_email	pswrd
A	a@uregina.ca	database1

Figure 9: Query to retrieve pswrd, prof_email and prof_name

```sql
-- finding all the submissions and their corresponding student numbers
-- given their course id, proffessor id, assignment number and question number
SELECT sub.ANS, s.stu_num
FROM Course c
INNER JOIN Professor p ON c.CID = p.CID
INNER JOIN Assignment a ON p.PID = a.PID
INNER JOIN Question q ON a.AID = q.AID
INNER JOIN StuCoInfo sci ON p.PID = sci.PID
INNER JOIN Submission sub ON q.QID = sub.QID AND sci.STUCO_ID = sub.STUCO_ID
INNER JOIN Student s ON sci.SID = s.SID
WHERE c.CID = 1
AND p.PID = 1
AND q.Question_num = 1
AND a.Assignment num = 1;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ANS	stu_num
#include<iostream>#include<cstring>//tousestringcompare#include<string>...	12345
hello	12345

Figure 10: Query to retrieve all the submissions from database for comparison

```
-- insert query for when we detect plagiarism, to update the plag_detector table with the corresponding values

CREATE VIEW sid1 AS SELECT SID FROM Student WHERE stu_num = stu_num1;
CREATE VIEW sid2 AS SELECT SID FROM Student WHERE stu_num = stu_num2;
CREATE VIEW stucoid1 AS SELECT STUCO_ID FROM StuCoInfo WHERE SID = sid1.SID AND PID = currentPID;
CREATE VIEW stucoid2 AS SELECT STUCO_ID FROM StuCoInfo WHERE SID = sid2.SID AND PID = currentPID;
CREATE VIEW aid AS SELECT AID FROM Assignment WHERE Assignment_num = assignmentNum AND PID = currentPID;
CREATE VIEW qid AS SELECT QID FROM Question WHERE Question_num = questionNum AND AID = aid.AID;
CREATE VIEW subid1 AS SELECT SUB_ID FROM Submission WHERE STUCO_ID = stucoid1.STUCO_ID AND QID = qid.QID;
CREATE VIEW subid2 AS SELECT SUB_ID FROM Submission WHERE STUCO_ID = stucoid2.STUCO_ID AND QID = qid.QID;
CREATE VIEW plagid AS SELECT PLAG_ID FROM Plag_Detector WHERE SUB_ID_1 = subid1.SUB_ID AND SUB_ID_2 = subid2.SUB_ID;
UPDATE Plag_Detector
SET plag_percentage = pp,
   flag = CASE
          WHEN plag_percentage > 0 THEN 1
          ELSE 0
          END
  WHERE SUB_ID_1 <> SUB_ID_2
  AND PLAG_ID = plagid.PLAG_ID;
```

Figure 11: Query to update the plag_detector table after comparison

6.3 Concluding Remarks

We have addressed the issue of code plagiarism in programming assignments for CS courses at the University of Regina. The current method of using Turnitin software for detecting plagiarism is inadequate, leaving it up to instructors to manually compare submissions or introduce assessment criteria. The aim was to develop a robust and efficient algorithm for detecting code plagiarism, which can be integrated into existing plagiarism detection software or used as a standalone tool. The implemented project uses Levenshtein distance to calculate the similarity score between two files.[2] The results and conclusions of the research are discussed, as well as potential future work. Overall, the Code Plagiarism Detector is a valuable tool for educators and institutions looking to detect and prevent code plagiarism.

[2]When we were creating the database on the Hercules server, we realized that we did not have admin access to the University's database. So, we had to create our own local database server to make our project functional. We were not able to implement any encryption of the data that with the limited functionality of our local database.

7: Future Work

Due to the limited scope of our project and requirement for the CS 375: Database and Information Retrieval course, we could only implement some of the functionalities for the Code Plagiarism Detector. The current version is simple and only focuses on the task of comparing two code files and storing them in the database. There is a lot of further work that can be done on this product and upgrade it to industry standards. It can then be used as a standalone tool or integrated with online course management modules within the university for their use. Following are a few upgrades our group will be working on in the future:

- The current relation between the course and professor tables is 1-to-many. We can modify it to many-to-many and allow a professor to teach many courses.
- A function can be implemented to filter out common code fragments and keywords which should not count as being plagiarized. This could involve using a database of known, open-source code libraries or frameworks, and excluding code fragments that match these libraries from the comparisons.
- A function to account for differences in variable names and other minor changes that could be made to the code. This involves creating abstract syntax trees or code normalization to compare the underlying structure of the code, rather than just specific words and symbols used. [3] [4]
- After the comparison is done, it could generate similarity reports or summaries of the results. This involves displaying the matching code fragments in a user-

friendly format, along with the statistic about the overall similarity.

- Currently, the application is designed to be used only by instructors and professors, but it can be redesigned so that it can be used by students.
- A major limitation we realized is that the files are being compared only with the database. Functionality can be implemented by using web-crawlers to check for plagiarism over the internet.
- Machine learning algorithms can be used to analyze the comparison results and identify syntactical and lexical patterns to flag potential plagiarism cases with the files stored in the database.

References

[1] Teddi Fishman. "we know it when we see it" is not good enough: Toward a standard definition of plagiarism that transcends theft, fraud, and copyright, 2009.

[2] Tomáš Foltỳnek, Norman Meuschke, and Bela Gipp. Academic plagiarism detection: a systematic literature review. *ACM Computing Surveys (CSUR)*, 52(6):1–42, 2019.

[3] Jong Yih Kuo and Fu Chu Huang. Code analyzer for an online course management system. *Journal of Systems and Software*, 83(12):2478–2486, 2010.

[4] Jeong-Woo Son, Tae-Gil Noh, Hyun-Je Song, and Seong-Bae Park. An application for plagiarized source code detection based on a parse tree kernel. *Engineering Applications of Artificial Intelligence*, 26(8):1911–1918, 2013.

[5] K Vani and Deepa Gupta. Detection of idea plagiarism using syntax–semantic concept extractions with genetic algorithm. *Expert Systems with Applications*, 73:11–26, 2017.