Abstract

As recent studies illustrate, the ever-increasing problem of plagiarism in the academic world is a major predicament to the prompt and fair marking of students’ coursework. To be more specific, the need to detect plagiarism in students’ submitted work facilitated the development of numerous plagiarism detection techniques. Academic institutions and private individuals all over the world are troubled by the high number of students who plagiarize programming code and the use of competent plagiarism detection software is considered essential.

In this report I will elaborate on how Artificial Intelligence can improve the effectiveness of current automatic plagiarism detection techniques. Up to now, most attempts were either to implement an advanced method for detecting similarities in students’ programs or to combine a number of such techniques to achieve a more accurate detection.

My goal is to introduce a new approach in plagiarism detection with the expectation that its future development will endow with impressive results.

1. Introduction

As mentioned above, my intention is neither to implement a comprehensive program that will be used by the end user nor to improve existing techniques for detecting plagiarism in students’ programs. The sole aim of my work is to demonstrate that combining Artificial Intelligence with currently available detection techniques can improve their effectiveness and empower their users.

By confirming that such a combination is possible and by proving that such an approach can award its developers with impressive results, I hope to facilitate advancement in the automatic plagiarism detection.

Further on in this report, the problem of plagiarism and the available solutions will be discussed in detail as well as the way I decided to implement my approach.

2. Related Background Information

Having some experience from an Artificial Intelligence - related module that I attended during my second year of my studies here in Kent University, I was optimistic that my work will aid into solving the problem of plagiarism.

After completing both my research on current plagiarism detection techniques and studying some previous projects provided by my supervisor, Dr Peter Kenny, I was convinced that my approach was not only promising but unique too.

Most programs or applications utilize a single algorithm or a combination of algorithms to detect plagiarism. All of them lacked the ability to evolve through time and usage and to utilize currently available data from previous years. Dr. Peter Kenny, having implemented his own program that can identify plagiarism cases efficiently, encouraged me to proceed with my idea and offered me valuable information from his own experience.

Being given the “green light” by my supervisor, I carried out a research on available Artificial Intelligence algorithms that could be applied in such a situation. Unfortunately, my research indicated that there are many algorithms to choose from but most of them required knowledge on statistics.

After my supervisor’s suggestion, I contacted Dr Alex Freitas who helped me choose an appropriate algorithm, the Naïve Bayse Classifier, and indicated some useful books to help me understand the concept of the algorithm.
3. My Project’s Goal

My venture is to aid the Computing Laboratory staff of my University to detect plagiarism effectively when students admit their programs to be marked. The main purpose that my program is called to deliver is to prove that Artificial Intelligence can be combined with plagiarism detection techniques and this combination will have positive effects on those techniques. This will enable:

- More plagiarism cases to be identified.
- Reduce false positives.
- The development of a generic application that can be used along with any already implemented program to filter its results.
- Further development of this approach.

Plagiarism should be seen as academic dishonesty and thus be treated as a serious and punishable academic offence. I will be joyful to provide the means for fair marking and just treatment of all students who work hard and deserve a good mark.

4. Plagiarism

In simple terms, plagiarism is the act of copying another person’s work and then passing it off as one's own. In our case, students plagiarize code from various sources. Unfortunately, the great growth of the internet “world” makes plagiarism, nowadays, easier than ever and the task of detecting plagiarism excruciating.

Another common way students plagiarize is by copying code from other students, either form the same year or from previous years.

Plagiarism can be considered to be a form of cheating or stealing and can cause serious inconvenience to both the Universities that do not take into consideration such cases and the students who carry out this deceitful act. Universities will loose their reputation and students will end up failing in the exams or even worse, in real life.

4.1 Why students plagiarise?

There might be an infinite number of reasons to why students plagiarize; however, many of those reasons apply for most of those students with the main one being: Because it is so easy!

Regrettably, the people marking students’ assignments not only have to face the fact that students share their work with others or steal work from others but also have to face the massive “library” freely available to everyone; the internet. A student can discover, by doing a simple search, a high number of implementations related to his current assignment. Moreover, there are several sites that offer inappropriate services in exchange for money. They offer copyrighted information that corresponds to the needs of their members and even offer custom-made work for a specific assignment with an additional fee. Detecting such cases of plagiarism is hard, as all these pages are password protected and not check can be applied.

In other words, there are people that turned plagiarism into a profitable business. It seems that there always is a good excuse for students to plagiarise:

- Too busy with other assignments.
- Limited knowledge or skills to cope with the assignment.
- Too lazy to spend the time and effort needed to do the assignment.
- Did not feel like doing the specific assignment.
- Had to go to work.
- Wanted to go out and have fun.
- A friend offered to give them his assignment.
- Had a strong desire to acquire the highest mark possible.
- They like to cheat.
- They believe that the will not get caught.
- Fear of failure.
- They might feel that plagiarism is not important.

On the other hand, there are students who plagiarise without being aware of doing so. Such cases can happen because:

- Students might have not been instructed on how to use other people’s data appropriately.
- New students can be naïve when it comes to plagiarism.
- Different academic institutions can have a different policy regarding plagiarism.
- Students from different cultures may not be familiar with plagiarism regulations as they are set by British colleges and universities.

4.2 How do students plagiarise Java code?

There are numerous “techniques” applied by students who plagiarise Java code. Some of those “techniques” are rather simplistic and easy to identify but others are more complex and need special “treatment”.

We can categorise plagiarising students by their skills, as the more skilled they are the more complex plagiarism techniques they can use. For example, from a first year student we expect to make no change or to
change the variable names and the names of the methods implemented in a java program when he decides to plagiarize. A third year student though, is usually capable to reconstruct the parts of code he copied is several ways to make plagiarism detection harder. Most common of these ways are:
- Making changes in comments.
- Changing the layout of the program. For example, add or remove empty lines.
- Renaming all declared variables and methods.
- Reordering sections of code.
- Modifying control structures. For example, replacing for loops with while loops.
- Changing the data types of variables and methods.
- Changing conditions in loops.
- Copying only small parts of another user’s code.
These are the techniques that the final version of my program will aim to recognise.

4.3 Plagiarism Detection

Different academics have different strategies on how to fight plagiarism. Some of those strategies are aimed to educate and inform students to prevent plagiarism and others are aimed to punish students who were found guilty of plagiarizing. To understand why plagiarism detection is vital, we have to have an idea of the strategies adapted and their success:
- Explain the concept of plagiarism to students and inform them about its implications.
- Develop a clear policy about plagiarism and include it in the syllabus.
- Found an honour code and establish a judicial board to judge plagiarism cases.
- Encourage students to explore and research topics before attempting to tackle an assignment.
- Prefer to devise assignments that enhance freedom of choice and allow students to explore subjects in depth.
- Support students that carry out research.
- Teach students how to document correctly the findings of their research.
- Develop efficient plagiarism detection schemes.
- Require students to provide documentation when they are suspected of committing plagiarism.
- Use the appropriate disciplinary actions for each individual case to achieve the best results.

It is ethical to inform students about plagiarism and the disciplinary actions that can be applied in such cases. However, plagiarism can be seen as the speed limit when driving. We all know that it exists, we know that we will have to pay a fine when we get caught speeding but still most of us exceed the speed limit when we believe that there are no speed cameras.

Plagiarism detection in students’ assignments fulfils the same purpose of having speed cameras monitoring roads. Without the fear that they will get caught, most of the students will plagiarise freely. The Centre for Academic Integrity found that almost 80% of college students admit to cheating at least once. Furthermore, the Psychological Record indicates that 36% of undergraduates have admitted to plagiarizing written material.

After being convinced that plagiarism detection is essential, a question arises: “Is it important to perform efficient and accurate plagiarism detection or the fact that some detection exists will prevent students from plagiarizing?” Answering this question, certain facts indicate that futile plagiarism detection is incompetent to prevent students from plagiarizing. As stated by The National Centre for Policy Analysis: "Too few universities are willing to back up their professors when they catch students cheating, according to academic observers. The schools are simply not willing to expend the effort required to get to the bottom of cheating cases".

Furthermore, the Influence of Honour Codes found that 55% of faculty "would not be willing to devote any real effort to documenting suspected incidents of student cheating". Therefore, concrete feedback, when detecting plagiarism, is crucial to support an instructor’s decision to pursue a plagiarism case.

4.4 Plagiarism Detection Systems

The high increase in plagiarism and the fact that many plagiarism cases are difficult to identify and prove, lead academics to seek for effective plagiarism-detection software. Several programs can be found on the market today.

MyDropBox is a program that uses a single algorithm to find similarities between a huge collection of documents, considered as the Internet Archive, and the assignments submitted by students. This program check files in database systems like WebCT and it claims that it can check student files against over 8 billion articles and produce reports with results in just 1-2 minutes.

Turnitin and iThenticate work in a similar way. They are web-based and they search within various sources, such as the internet, commercial databases and previously submitted articles, to find if they match the student papers submitted. They offer high compatibility with different operating systems, since they are web-based and an efficient way to present similarities.

However, the three above programs are not specialized in identifying plagiarism in students’ programming assignments. MOSS on the other hand, is a System for Detecting Software Plagiarism. In fact,
MOSS stands for Measure Of Software Similarity. It is capable of finding similarities in C, C++, Java, Pascal, Ada, ML, Lisp, or Scheme programs. It detects similarities using a single algorithm that it is considered to be more advanced than most single plagiarism detection algorithms.

Similarly, SID is another system that is designed to find similarities in programs. SID stands for Shared Information Distance or Software Integrity Detection. It computes the shared information between programs to detect possible similarities. It uses a single algorithm that was originally invented to detect how similar genomes are. SID is MOSS’s main rival and claims to be better in that it offers a web interface, approximate matching and maintains account information online.

Concluding, JPlag is another interesting attempt against plagiarism, this time by the University of Karlsruhe in Germany. Similarities in Java, C#, C and C++ languages can be detected by using information based on the structure of each language. It is better than MOSS in that it does not group similar programs and thus it has now limits when presenting the results. Additionally, it allows the final results to be downloaded to the user’s machine.

5. The “Big Dilemma”

As previously discussed, the aim of my work is to prove whether applying Artificial Intelligence to a plagiarism-detection technique will improve its competence in identifying plagiarism correctly. Specifically, my goal was to provide convincing evidence that a supervised classification algorithm can improve an existing plagiarism-detection technique. Thus, the main idea was, in simple terms, to find an existing technique and a suitable algorithm, implement both of them and finally combine them and check the final product. So what is the “Big Dilemma”? I had to decide which was the appropriate technique and algorithm to implement and combine. Wrong decisions could cause failure in proving whether an Artificial Intelligence approach is beneficial for detecting plagiarism.

Before deciding though, I had to carry out a research based on existing methods for detecting plagiarism and existing classification algorithms.

5.1 Existing methods for plagiarism detection

Plagiarism has been causing problems to academics for many years and there were many attempts to detect plagiarism. The evolution of computers aided those seeking to plagiarise but also empowered those determined to identify and punish this dishonest act. All these attempts were the foundation of different methods that can be used to identify plagiarism. Revolutionary techniques, such as Visualisation, Compression, Watermarking and Clustering, were implemented but their effectiveness was not proved. The dominant techniques are those of Attribute-Counting Metric Systems and Structure Metric Systems.

Attribute-Counting Metric Approach is considered to be the earliest method used to detect plagiarism. Attribute-Counting Metrics are also called Linguistic Metrics. Such metrics are used for measuring the properties of programs without actually having to deduce the meaning of those properties. The first attributed-counting systems were based on Halstead's science metrics were programs were classified either as operators or operands. Two popular formulas that utilized those metrics were:

\[ V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2), \]
\[ E = \left[ \eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2) \right] / (2 \eta_2). \]

Where
\[ \eta_1 = \text{number of distinct operators}, \]
\[ \eta_2 = \text{number of distinct operands}, \]
\[ N_1 = \text{total number of operator occurrences} \]
\[ N_2 = \text{total number of operand occurrences} \]

The result of these formulas was the level of similarity between two programs. Many more metrics have been added, as this technique was evolved, to provide for results that are more accurate. The Attribute-Counting technique was promising and its evolved stages proved to be successful in detecting plagiarism. However, this technique has a major flaw. As it does not take into account the structure of programs, it is ineffective for cases were partial plagiarism took place. In other words, when only small parts of code were copied, this method was unable to detect plagiarism.

For correcting that flaw, Structure Metric Systems were brought to surface and they are currently the prevailing systems used for detecting plagiarism. Programs like MOSS, YAP and JPlag are based on different implementations of this technique. Structure Metrics are based on the structural relations of objects within a program. Most variations of this method convert programs into token strings and compare them by using an algorithm. The Greedy-String-Tiling algorithm, for example, is used in YAP3. It compares two strings, the pattern and the text, and searches in the text to find substrings of the pattern. The Structure Metric approach is more complex to implement but succeeds in identifying partial plagiarism too.
5.2 Supervised Classification Algorithms

Every method mentioned above, takes the programs to be checked as an input and after a series of processes, it gives results as an output. Even the most sophisticated method, however, will become ineffective over time as students find new ways to plagiarise and avoid detection. To find out whether Artificial Intelligence can automate this need for constant evolution, I had to research possible machine learning techniques that could be combined with one of the above plagiarism-detection methods. Pattern recognition was the type of machine learning to be considered for my project. Pattern recognition can be defined as the classification of data into predefined categories. This is normally done by using statistical methods. Methods for pattern recognition are generally divided into two categories, supervised and unsupervised learning. Since unsupervised learning follows a rather “loose” classification where classes are not pre-defined, the following supervised classification algorithms were researched:

1. Quadratic classifier
2. Artificial neural network
3. Backpropagation
4. Boosting
5. Bayesian statistics
6. Case-based reasoning
7. Decision tree learning
8. Inductive logic programming
9. Gaussian process regression
10. Minimum message length
11. Naive Bayes classifier
12. Nearest Neighbour Algorithm
13. Probably approximately correct learning (PAC)
14. Support vector machines
15. Random Forests

5.3 Naïve Bayes Classifier

After studying the above algorithms, it became clear to me that the most appropriate algorithm to implement was Naïve Bayes Classifier for two main reasons; it is less complex than most of the other algorithms and it is surprisingly efficient and robust. In fact, it has been proven that Naïve Bayes Classifiers can actually perform as well as more sophisticated supervised classification algorithms when they are tested in complex real-world situations. Naïve Bayes Classifiers are based on Bayes’ theorem, which states that “The probability of an event A conditional on another event B is generally different from the probability of B conditional on A. However, there is a definite relationship between the two.”

Naïve Bayes can be modelled in several different ways:

- Normal Function

\[
\mathcal{N}(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma^2} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right), \quad -\infty < x < \infty, \ \mu, \sigma > 0
\]

- Lognormal Function

\[
\mathcal{LN}(x \mid \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{\log(x^2 \mu^2)}{2\sigma^2}\right), \quad 0 < x < \infty, \mu > 0, \sigma > 0
\]

- Gamma Function

\[
f(x \mid \alpha, \beta) = \frac{x^{\alpha-1} \exp\left(-\frac{x}{\beta}\right)}{\Gamma(\alpha) \beta^\alpha}, \quad 0 \leq x < \infty, \alpha, \beta > 0
\]

- Poisson Function

\[
\mathcal{P}(x \mid \lambda) = \frac{\lambda^x \exp(-\lambda)}{x!}, \quad 0 \leq x < \infty, \lambda > 0, \lambda = 0, 1, 2, ...
\]

In simple terms, the structure of the classifier can be examined in three main levels:

1. Calculation of Prior Probabilities. Prior Probabilities are the probabilities calculated from the training data and thus they indicate past experience. For example, if the training data given to the classifier consisted of 50 non-plagiarised couples and 10 plagiarised couples and there were two classes, Plagiarised and Non-Plagiarised:

Prior Probability for Plagiarised =

Plagiarised Couples / Total Couples = 10/60

Prior Probability for Non-Plagiarised =

Non-Plagiarised Couples / Total Couples = 50/60

Probabilities are to be calculated based on couples of files since two files are compared to be checked for plagiarism.
2. Calculation of Likelihood. At this level, all the necessary probabilities for classifying a new entry are calculated. For example, if the metric Total Number of Lines could return three values, low, medium and high, then the following probabilities have to be calculated:

\[
\begin{align*}
    P(\text{numberOfTotalLines} = \text{low} \mid \text{Plagiarised} = \text{Yes}) \\
    P(\text{numberOfTotalLines} = \text{low} \mid \text{Plagiarised} = \text{No}) \\
    P(\text{numberOfTotalLines} = \text{medium} \mid \text{Plagiarised} = \text{Yes}) \\
    P(\text{numberOfTotalLines} = \text{medium} \mid \text{Plagiarised} = \text{No}) \\
    P(\text{numberOfTotalLines} = \text{high} \mid \text{Plagiarised} = \text{Yes}) \\
    P(\text{numberOfTotalLines} = \text{high} \mid \text{Plagiarised} = \text{No})
\end{align*}
\]

3. Calculation of Posterior Probabilities. This level is a combination of the two previous levels. Thus, if the value of Total Number of Lines metric is low:

\[
\begin{align*}
    \text{Posterior Probability of Couple 1 being Plagiarised} &= (\text{Prior Probability for Plagiarised} \times P(\text{numberOfTotalLines} = \text{low} \mid \text{Plagiarised} = \text{Yes})) \\
    \text{Posterior Probability of Couple 1 being Non-Plagiarised} &= (\text{Prior Probability for Non-Plagiarised} \times P(\text{numberOfTotalLines} = \text{low} \mid \text{Plagiarised} = \text{No}))
\end{align*}
\]

Ending, if the Posterior Probability of Couple 1 being Plagiarised is higher than the Posterior Probability of Couple 1 being Non-Plagiarised, then Couple 1 is classified as Plagiarised and vice versa.

5.4 Decisions for the “Big Dilemma”

The “Big Dilemma” required two decisions to be taken. I had to decide which plagiarism-detection method and which supervised classification algorithm I will implement. As explained above, Naïve Bayes Classifier was the chosen algorithm and since this algorithm requires strong independence assumptions, the plagiarism-detection method most suitable to be implemented was that of Attribute-Counting Metrics.

In addition, although Attribute-Counting Metrics Approaches are unable to detect partial plagiarism, they offered the prospect to clearly show how supervised classification can be used to enhance plagiarism detection. This was because an Attribute-Counting Metrics Approach is less complex that a Structure Metric Approach and thus it offered fewer risks to be wrongly implemented and allowed more time to be spent on the implementation of the classification algorithm.

The implementation of this system, called AutoplagAI, was completed in three main iterations. In the first iteration, twelve Attribute-Counting Metrics have been implemented and formed a base for the second and third iterations. In the second iteration, an approach was implemented for utilizing the Attribute-Counting Metrics created in the first iteration. Finally, in the third iteration, the Naïve Bayes Classifier was implemented and combined with the existing metrics.

AutoplagAI was implemented in three iterations to facilitate time management, testing and evaluation. A detailed analysis of these three iterations is provided further in the report.

6. Attribute-Counting Metrics

In the first iteration of AutoplagAI, twelve attribute-counting metrics have been implemented. The purpose of these metrics is to describe a program by measuring its “characteristics”.

The first metric measures the total number of lines in a program. The functionality of this metric is not affected by the contents of each line. The contents of the java file being checked are parsed and every time a line is being read, a counter is incremented by one.

The second metric measures the number of lines that contain comments. The program iterates through the already parsed code and increments a counter for every line that begins with /**, */ or contains /**, // In this way, lines that contain both code and comments are being included in this metric.

Following, a metric for counting all the primitive variables has been implemented. The program iterates through the parsed code and increments a counter for every time an int, String, char, boolean, double, byte, long, short, or float variable is initialized. The initialized primitive variables are recognised by the following segment of code:

```java
    if ((result[x].contains("int ")) || result[x].contains("String ") || result[x].contains("char ") || result[x].contains("boolean ") || result[x].contains("double ") || result[x].contains("byte ") || result[x].contains("long ") || result[x].contains("short ") || result[x].contains("float ")) &
    (result[x].contains(";"))
    
    i++;
    }
```

The fourth metric counts all the packages or classes being imported in the program. The program iterates through the parsed code and increments a counter for every line that starts with the keyword “import”.

Responsible for counting the number of empty lines, is the metric number five. For every line that contains
no characters, within the parsed code, a counter is incremented by one.

The next metric measures the number of declared methods in the code been parsed. This is done by the following segment of code:

```java
if ( ((tmp.trim().startsWith("public") &&
   tmp.trim().endsWith("y") &&
   (tmp.contains(className) == false)) ||
   ((tmp.trim().startsWith("public") &&
   tmp.trim().endsWith("{") &&
   tmp.contains(")")) &&
   (tmp.contains(className) == false)))
{
   i++;
}
```

Likewise, the next metric counts the total number of constructors in the file. For this, a simpler check was needed:

```java
if(result[x].trim().startsWith("public " + className))
{
   i++;
}
```

Counting Java operators used in a file can help in identifying plagiarism that has been concealed by converting conditions so that different operators will bring the desired outcome. The following part of code is used, in this metric, to identify multiplicative, additive, shift, relational, equality, bitwise logical and ternary operators:

```java
if(result[x].contains("+" || result[x].contains("-") ||
result[x].contains("&") || result[x].contains("<") ||
result[x].contains(">") ||
result[x].contains("==") ||
result[x].contains("!=") ||
result[x].contains("%") ||
result[x].contains("?:"))
{
   i++;
}
```

The ninth metric measures the number of times certain keywords appear in the parsed code, which indicate the existence of a loop. These keywords are for, while, if and switch and the portion of code responsible for identifying them is:

```java
if(tokens[x].contains("+=") ||
tokens[x].contains("-=") ||
tokens[x].contains("&=") ||
tokens[x].contains("<=") ||
tokens[x].contains(">=") ||
tokens[x].contains("===") ||
tokens[x].contains("!==") ||
tokens[x].contains("%=") ||
tokens[x].contains("?:="))
{
   i++;
}
```

Word counting is the task carried out by the next metric. Contrasting the tokenization that happened in most of the other metrics, this time the parsed code was tokenized at every white character instead of been tokenized at the end of each line. Every string with more than two characters is considered to be a word and a counter is incremented. This was achieved with the following segment of code:

```java
String[] words = code.split(" ");
int i = 0;
for (int x=0; x<words.length; x++)
{
   if(words[x].length() > 2)
   {
      i++;
   }
}
```

The eleventh metric is used to identify if the file implements an interface. The expected result of this metric is either 0 or 1, since it does not measure how many interfaces it implements. With the code below, the parsed code is checked for a sequence of the keywords public, the name of the class and implement:

```java
if(result[x].trim().startsWith("public " + className)
   && result[x].contains("implements"))
{
   i++;
}
```

The final metric is related to the class name of each java file. Instead of comparing the similarity of each class name by comparing them as strings, the length of each class name is used.

### 6.1 Testing & Evaluation of Attribute-Counting Metrics

For implementing the twelve attribute-counting metrics mentioned above, Iteration 1 passed through several stages were different approaches have been tested. A lot of effort has been placed on using Reflection, a feature unique for the Java language. Reflection enables the manipulation of the internal properties of executed programs. However, the fact that a program has to be compiled and executed before using this feature could cause serious security issues. For
example, if a file being checked contains malicious code, then the computer of the user of AutoplagAI will be infected. Therefore, this approach was dropped and all twelve metrics were implemented to work by iterating on the parsed code.

The final version of Iteration 1 has been tested against files, containing java code, of various sizes. All metrics worked correctly and they were ready to be used in Iteration 2.

7. Simple Utilization of Attribute-Counting Metrics

After completing Iteration 1 and implementing all twelve metrics, I implemented Iteration 2 in various stages. Firstly, I tackled the task of utilizing the twelve attribute-counting metrics in away that will detect plagiarism efficiently. As previously mentioned in section 5.1, many attribute-counting metric systems have been using one of the following formulas to utilize the metrics and detect plagiarism:

\[ V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2), \]
\[ E = [\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)] / (2 \eta_2). \]

Where
- \( \eta_1 \) = number of distinct operators,
- \( \eta_2 \) = number of distinct operands,
- \( N_1 \) = total number of operator occurrences
- \( N_2 \) = total number of operand occurrences

Knowing that by using these two formulas, complex plagiarism techniques can not be detected, I decided to try another approach.

To begin with, I "hard coded" into my program twelve weights, one for each metric. Each weight indicates the maximum contribution of each metric into Plagiarism Percentage. Obviously, the maximum value of Plagiarism Percentage is 100 and it is the final result that indicates the possibility that plagiarism existed when two files are compared. These weights are:

- numberOfTotalLinesPercent = 10;
- numberOfOperatorsPercent = 10;
- numberOfConstructorsPercent = 5;
- numberOfCommentsPercent = 10;
- numberOfImportedPercent = 5;
- numberOfEmptyLinesPercent = 10;
- numberOfVariablesPercent = 10;
- numberOfWordsPercent = 10;
- numberOfDeclaredMethodsPercent = 5;
- numberOfImplementedInterfacesPercent = 5;

The next step was to create my own formula that will use those weights to calculate a Plagiarism Possibility Percentage for every couple of files being checked. Instead of using logarithms, I based my formula on the Difference between the calculated results of each metric. To elaborate, my formula had to produce a result that will increase along with an increase in the similarity of two files and vice versa.

To calculate the similarity of two files I used the following formula:

\[ \frac{MC_a - MC_b}{(MC_a + MC_b) / 2} \]

Where:

- \( MC_a \) = Metric Calculation for file A
- \( MC_b \) = Metric Calculation for file B

The numerator represents the difference between the measurements of a metric for the two files being compared and the denominator represents an average of the metrics measurements. Therefore, the higher the result of this formula the less similar the two files are. For example, in the case of the Number of Total Lines metric, we can assume that file A has 200 lines and file B has 100 lines. So the result of the formula will be:

\[ \frac{200 - 100}{(200 + 100) / 2} = \frac{100}{150} = \frac{10}{15} \approx 4.29 \]

Now that the two files are more similar, the result is lower. The final step for the formula to be complete was to combine the part explained above with the weight of every metric and convert that combination into a percentage.

After several attempts, I found out that by dividing the weight of a metric with the formula above and then converting them into a percentage, the program worked efficiently. Thus, the final formula used for every metric was:

\[ \frac{MW}{MC_a - MC_b} / 100 = MPPP \]

Where:

- \( MW \) = Metric Weight
MCa = Metric Calculation for file A
MCb = Metric Calculation for file B
MPPP = Metric Plagiarism Probability Percentage

After implementing the code for using my formula for each metric, I was able to calculate a Plagiarism Probability Percentage for every couple of files being compared. The second and final stage of Iteration 2 was to enable AutoplagAI to retrieve java files from a specific directory and to compare all files against each other to identify which couples have a high Plagiarism Probability Percentage.

This stage was necessary as it allowed more efficient use and testing of the program for both the second and third Iterations. After the implementation of this final stage, AutoplagAI required as a parameter the path of the main directory that contains the files to be checked. Based on the way that most lecturers at the University of Kent structure the modules’ directories for submitting assignments, my program was designed to retrieve all the subfolders of the given directory and retrieve only the java files from each subfolder. A tree representation of the structure of a module’s directory is given below:

CO520
  Assignment1
    kk57
      autoplagAI.java
    vs36
      autoplag.java
    qmp21
      plag.java

In this case, the user of AutoplagAI will give …/CO520/Assignment1 as a parameter and the program will retrieve autoplagAI.java, autoplag.java and plag.java and will compare them against each other. Every comparison that yields a result of more than 60% is printed to the user to notify him for possible plagiarism.

7.1 Testing & Evaluation of Metrics Utilization

The final version of Iteration 2 was checked against the plagiarism techniques mentioned in the section 4.2. Eight Test Cases were developed to test whether my “Simple Utilization of Attribute-Counting Metrics” Approach was working and being able to detect plagiarism efficiently. In Test Case 1, the copy of a file was submitted after changing: the file name, class name, method names and the variable names, author name. When AutoplagAI checked the files, plagiarism was successfully detected.

Under Test Case 2, another plagiarised file was submitted where only the file name and the author name were changed. AutoplagAI identified both plagiarism cases. For Test Case 3, the file been modified in Test Case 1 was changed even more. All empty lines where removed. The program managed to correctly identify all plagiarised couples again.

Test Case 4, however, was the program’s first failure. The file that was modified in Test Cases 1 and 3 was changed again. This time all comments were removed too and AutoplagAI failed to detect it as plagiarised.

In Test Case 5, a large file was added in unclassified data and variable and method names have been changed in an identical copy. The program identified this case of plagiarism successfully but it was still failing to detect the file modified in Test Case 4. When all empty lines and comments were removed from the large file too, under Test Case 6, AutoplagAI failed to identify it as plagiarised.

Nevertheless, AutoplagAI succeeded to identify plagiarism when parts of code were reordered in Test Case 7 and when loops and equality operators were changed in Test Case 8.

Iteration 2 of AutoplagAI has been surprisingly successful. The combination of the twelve metrics with the plagiarism-calculation formula and the retrieval of files from directories produced positive results. The program failed to identify plagiarism only when the following changes took place in the same file: The file name, class name, method names, variable names and author name were changed and all empty lines and comments were removed.

This indicates that AutoplagAI – Iteration 2 can efficiently identify many plagiarism techniques but it will fail if many techniques are combined to modify a plagiarised file. Iteration 3 has the mission of identifying these more complex, sophisticated techniques by introducing an Artificial Intelligence approach.

8. The Artificial Intelligence Approach

In section 5.3 it was explained that Naïve Bayes Classifier is the algorithm chosen to be implemented in Iteration 3. Naïve Bayes Classifier is a learner algorithm, which is why it is also known as Naïve Bayes Learner. Its purpose, in AutoplagAI, is to facilitate the implementation of an Attribute-Counting Metrics System which will be able to be trained and evolve.

Before implementing Iteration 3, a plan was necessary to manage to implement the different aspects of the system in the correct order. The different parts of the system are listed below in the order they had to be implemented:

1. Twelve Attribute-Counting Metrics
2. A program that calculates the average difference for every metric
3. A program that imports both training and unclassified (unchecked) data.
4. A program that implements Couples as objects, with a number of required characteristics.
5. A program that calculates the necessary probabilities.
6. A program that classifies the unclassified data.

The first part was implemented in Iteration 1, so I proceeded immediately in the implementation of part two. Iteration 2 was not useful in any way at this stage.

8.1 Calculating Metrics’ Average Differences

For two files to be compared, twelve metrics have been implemented. After the metrics are calculated for each file separately, the results of each metric have to be compared and their difference is saved. The higher their difference, the less similar they are.

In this part of the program, the following segment of code is used to calculate that difference and convert it into a percentage:

\[
\text{difference} = a.\text{numberOfTotalLines} - b.\text{numberOfTotalLines}; \\
\text{average} = (a.\text{numberOfTotalLines} + b.\text{numberOfTotalLines}) / 2; \\
\text{metricResult} = ((\text{difference/average})*100);
\]

This is repeated for every metric and the results are saved into an array as they are later used in other stages of the program.

8.2 Importing Training and Unclassified Data

Two main kinds of data have to be imported into the system; Training data and Unclassified data. However, training data has to be given in a special way by the user, it is necessary for the program to know which combinations of files (couples) are plagiarised and which are non-plagiarised.

Non-plagiarised data could be obtained in a similar manner as when acquiring data for Iteration 2. Thus, the segment of code responsible for importing non-plagiarised data was implemented in a way that the program takes the path of a directory as a parameter. It searches and gets any java files from subdirectories one level deep. Those files are compared against each other and a number of details are saved into arrays so that they will be used when creating Couples objects. These details are the user owning file one, the name of file one, the name of file two and the twelve average differences calculated when the two files were compared. Files that belong to the same user, which are found in the same subfolder, are not compared against each other. The files imported with this method are attached with a Boolean with false as its value. This is done to indicate to the program that the couples created by these files are to be treated as non-plagiarised.

Unclassified data is obtained at the same way since the user does not have to specify whether the files are to be treated as plagiarised or non-plagiarised. The difficulty was to decide how plagiarised files are to be imported. The arising issue is that the user has to specify which two files form a plagiarised couple. Instead of forcing the user to type into the program the names of the files that consist in each couple, I decided that it will be more convenient to acquire the plagiarised couples in a similar manner to the non-plagiarised and the unclassified files. However, this time some limitations are imposed. For the program to identify which files form a plagiarised couple, the user will give as a parameter the path of a directory that is structured in a special way. This directory can only contain two java files in each subfolder, the two files that form a plagiarised couple. After both the training data and the unclassified data are imported, some more calculations take place to return the number of unclassified couples, the number of plagiarised training couples and the number of non-plagiarised training couples. These values are needed later to calculate some of the probabilities.

For this part of the program to finish, the next part has to be implemented too. This is because this part uses the information saved into arrays to create objects of type Couples. Details about these objects are given at the next section.

8.3 Implementing Couples

The previous part of the program gathers some information about the files imported and then uses this part to create new objects of type Couples for every two files being compared. Couples are objects with six characteristics: Name of User 1, Name of File 1, Name of User 2, Name of File 2, an array containing the twelve average differences and a Boolean that indicates whether the couple is plagiarised or non-plagiarised.

At this part of the program, another important process takes place. For Naïve Bayes Classifier to work, the Metrics’ Average Differences have to be classified into categories. Thus, here three categories are used, low, medium and high, to represent these differences. Differences less or equal to 39% are classified as high, between 40% and 69% medium and above 69% low.
After calculating the average differences as percentages, importing plagiarised, non-plagiarised and unclassified data and creating Couples objects every time two files are compared, all the necessary probabilities have to be calculated. This is done in the next part.

### 8.4 Probabilities

To classify the unclassified couples either as plagiarised or non-plagiarised, a number of probabilities has to be calculated. Probabilities can be divided in three categories, Prior Probabilities, Likelihood Probabilities and Posterior Probabilities.

In this part of the program Prior and Likelihood Probabilities are calculated. Prior Probabilities are only two, since the unclassified couples can be classified only as plagiarised or non-plagiarised. An intermediate class could be used, called Suspicious, but as we will observe later this would have caused a great increase in the amount Likelihood Probabilities that have to be calculated. Prior Probabilities are simple to understand and to implement as they are based on the total of all couples, the total of plagiarised couples and the total of non-plagiarised couples. The code implementing them is the following:

```java
plagiarisedCouplesPriorProbability = ((double)plagiarisedCouplesNum() / (double)totalCouplesNum());
nonPlagiarisedCouplesPriorProbability = ((double)nonPlagiarisedCouplesNum() / (double)totalCouplesNum());
```

Likelihood Probabilities, however, are calculated using the values of all the average differences of metrics and the two classes of Plagiarised and Non-plagiarised as well. The combination of all these values results in having to calculate six different probabilities for every metric, seventy-two in total. As explained above, each metric’s average difference calculation can result in being high, medium or low. In addition, we know that couples are classified into Plagiarised and Non-plagiarised. Therefore, there are six possible combinations for every individual metric. To elaborate on this, we can consider the case of the Number of Empty Lines metric. The average difference of Empty Lines can be high when calculated for a plagiarised couple or high when calculated for a non-plagiarised couple. Likewise, it can be medium for a plagiarised couple or medium when calculated for a non-plagiarised couple and low for a plagiarised couple or low for a non-plagiarised couple. This can be represented with the following probabilities:

- \( P(\text{NumberOfEmptyLines} = \text{low} | \text{Plagiarised} = \text{Yes}) \)
- \( P(\text{NumberOfEmptyLines} = \text{low} | \text{Plagiarised} = \text{No}) \)
- \( P(\text{NumberOfEmptyLines} = \text{medium} | \text{Plagiarised} = \text{Yes}) \)
- \( P(\text{NumberOfEmptyLines} = \text{medium} | \text{Plagiarised} = \text{No}) \)
- \( P(\text{NumberOfEmptyLines} = \text{high} | \text{Plagiarised} = \text{Yes}) \)
- \( P(\text{NumberOfEmptyLines} = \text{high} | \text{Plagiarised} = \text{No}) \)

Consequently, having an intermediate class would have caused the couples to be classified into Plagiarised, Suspicious and Non-plagiarised and nine probabilities would be needed for every metric:

- \( P(\text{NumberOfEmptyLines} = \text{low} | \text{Plagiarised} = \text{Yes}) \)
- \( P(\text{NumberOfEmptyLines} = \text{low} | \text{Plagiarised} = \text{No}) \)
- \( P(\text{NumberOfEmptyLines} = \text{low} | \text{Suspicious}) \)
- \( P(\text{NumberOfEmptyLines} = \text{medium} | \text{Plagiarised} = \text{Yes}) \)
- \( P(\text{NumberOfEmptyLines} = \text{medium} | \text{Plagiarised} = \text{No}) \)
- \( P(\text{NumberOfEmptyLines} = \text{medium} | \text{Suspicious}) \)
- \( P(\text{NumberOfEmptyLines} = \text{high} | \text{Plagiarised} = \text{Yes}) \)
- \( P(\text{NumberOfEmptyLines} = \text{high} | \text{Plagiarised} = \text{No}) \)
- \( P(\text{NumberOfEmptyLines} = \text{high} | \text{Suspicious}) \)

The translation of probability “\( P(\text{NumberOfEmptyLines} = \text{low} | \text{Plagiarised} = \text{Yes}) \)” into code is:

```java
numEmptyLinesLowPlagiarisedYesProbability = ((double)y / (double)(plagiarisedCouplesNum()));
```

Where \( y \) = number of times the Empty Lines Difference was low when it was calculated for all plagiarised couples.

### 8.5 Classification

This is the final part of Iteration 3. In this part the unclassified couples are being classified into either Plagiarised or Non-plagiarised. This classification is done by calculating the Posterior Probabilities. Two Posterior Probabilities exist for every couple, the Posterior Probability that the couple is plagiarised and the Posterior Probability that the couple is not plagiarised. If the Posterior Probability of the couple being plagiarised is higher than the Posterior Probability of the couple not being plagiarised, the couple is classified as plagiarised and vice versa.

Posterior Probabilities are calculated by combining Prior Probabilities with Likelihood Probabilities. To be more specific:
Posterior Probability for Plagiarised = Prior Probability for Plagiarised * Likelihood Probabilities

Posterior Probability for Non-Plagiarised = Prior Probability for Non-Plagiarised * Likelihood Probabilities

As there are many Likelihood Probabilities, the appropriate ones have to be selected for each case. For instance, when a new couple is being classified the average difference for each metric is checked so that the correct probability will be chosen. If the number of loops difference, for example, is low then the values of two probabilities are selected. The value of “numLoopsLowPlagiarisedYesProbability” and the “numLoopsLowPlagiarisedNoProbability”, as both the Posterior Probability for being Plagiarised and the Posterior Probability for not being plagiarised have to be calculated. This selection has to be done for every metric.

8.6 Testing & Evaluation of Artificial Intelligence approach

Previously, in section 7.1, we have seen an evaluation of the second Iteration of AutoplagAI after a number of tests were applied. Iteration 2 was evaluated as capable of detecting rather straightforward plagiarism and failed when files were modified in a more complex manner. Specifically, the program failed to identify plagiarism when the following modifications were applied to a java file: The file name, class name, method names, variable names and author name were changed and all empty lines and comments were removed.

Iteration 3 was implemented to overcome such limitations. The concept behind implementation was that if the program could be trained by the user, then it can be prepared to detect any plagiarism case known to the user. Over time, the program will evolve and become more powerful in detecting plagiarism as more training data will be available to be given to the program.

The final version of Iteration 3 has been tested to prove whether my approach was correct or not. For this purpose, seventeen Test Cases were build that tested various aspects of the program as it is important for my final version of AutoplagAI to be efficient in detecting plagiarism, robust, reliable and be able to process files at a reasonable speed. Completing the necessary tests, I was pleased to find that AutoplagAI satisfied all the above aspects.

The program was tested against the following plagiarism techniques:

- Changing the file name, class name, method names, variable names and author name.
- Changing only the file name and author name.
- Changing the file name, class name, method names, variable names and author name and removing all empty lines.
- Changing the file name, class name, method names, variable names and author name, removing all empty lines and removing all comments.
- Reordering segments of code.
- Changing loops and operators.
- Changing the file name, class name, method names, variable names and author name, removing all empty lines, removing all comments, reordering segments of code, changing loops and changing operators in a single file.

AutoplagAI succeeded in identifying all of the above cases after the necessary training data was given to the program. The program was also tested in the following:

- Increasing training data to see if it affects previously identified plagiarism cases.
- Plagiarism Detection without using training appropriate training data.
- Its ability to processes files with different size from that of the training data.
- Its ability to process multiple files per user.
- Its speed and stability when processing a large number of files.

The only limitation found, was that AutoplagAI has to be given the appropriate training data to identify the corresponding plagiarism cases.

9. Conclusion

AutoplagAI proved to be accurate when given the appropriate training data, robust as it will operate correctly even in cases where some of the metrics fail to be calculated and fast as it can process 35 java files in a couple of seconds. It outperformed Iteration 2, which was the implementation of a regular attribute-counting metrics system, in every possible aspect and proved that the combination of Artificial Intelligence with plagiarism detection techniques can offer astonishing results.

9.1 Possible Improvements

Having almost no experience with statistics and complex supervised classification algorithms in the past, I chose to implement my program in more analytical
way so that I can detect bugs and malfunctions caused by misapprehension of the statistical concepts. This approach though caused parts of code to be implemented in a less than optimal way. A necessary improvement, now that evidence exists for the success of this approach, is to review the code and remove code duplications to make it more compact.

9.2 Further Development

There are two main areas of development for AutoplagAI. The first is to improve the way that current metrics work and add more metrics that might help plagiarism detection. In addition, a more sophisticated training approach can be implemented so that the program can be trained by multiple resources and save every given database to create a global database and evolve every time it is used. A sophisticated GUI can be implemented to obtain the necessary parameters from the user and provide an output appropriate to be presented in front of a committee. In this way a fully operational, automated system can exist.

The second area of development is to continue researching this approach. There are many plagiarism-detection methods and many supervised classification algorithms that can be implemented and tested to provide evidence for the optimal combination. More sophisticated implementations can be developed by students or academics with more time available and more experience on the aspects elaborated throughout this report.

10. Acknowledgements

First of all, I would like to show appreciation to my supervisor, Dr Peter Kenny for his valuable support and guidance throughout the project. Several drawbacks existed during the development of this project but he helped me find an optimal solution and move forward.

Special thanks are also given to Dr Alex Freitas and Dr Andrew Runnalls for their help in understanding the statistical concepts surrounding this project.

11. Bibliography


[5] Project Research Documents

