UNIVERSITY of STIRLING

Division of Computing Science and Mathematics

Faculty of Natural Sciences

University of Stirling

The Potential Applications of a "De-Spinner" in the Role of Plagiarism Detection

Sara Wilson

Dissertation submitted in partial fulfilment for the degree of Master of Science in Big Data

September 2022

Abstract

Problem:
A major issue faced by examiners in academic institutions is the difficulty that can sometimes surround the investigation of plagiarised work. While in some cases, plagiarism detection programs such as Turnitin can identify extracts that could be from another source, it is not always accurate, and can throw up false positives where perhaps no plagiarism has taken place. Submissions that make it past detection can still feature very strange grammar, sentence structure, and general flow which an examiner will need to spend a great deal of time manually investigating, to try and find where it has come from.
Most plagiarism from other papers or sources is performed using online "spinner" tools, which change the sentence structure, flow, and words of a piece of text to try and evade detection but often do a poor job.
By creating our own spinner tool and using it to convert a set of papers and the suspicious text to a root form which is as similar as possible, we can hopefully improve detection of plagiarism and take some pressure away from examiners.
Objectives:

- Design and construct a program which can take text from a field, or loaded in through a PDF file, and "spin" the text, using the support of APIs to convert words into their root form, and then to a synonym.
- Further improve the program by creating a "de-spinner" where the user can compare the similarity of the supplied text and a corpus of sample papers, by converting both to their root forms as far as possible, trimming any formatting differences out, and comparing the similarity of the two using the Levenshtein Distance metric to gauge likelihood of plagiarism.
- Develop the Levenshtein Distance scoring metric, which behaves similarly to other distance metrics, but places greater weight on the number of changes required to take one string to another, hence noticing changes in tense or the use of some synonyms more easily.
- Testing and refinement of the distance metric through repeated testing; taking samples of text from the corpus, spinning it (perhaps more than once), and attempting to compare the similarity of this spun text with the corpus as discussed prior – until the level of accuracy of the "de-spinner" has reached an acceptable point.
- Analysis of the results gained from testing, including what types of spin detection the system is more, or less, capable of, and why.
- Consideration and discussion of the results of the project in terms of what it could mean for the wider academic community, particularly for examiners, if developed further and refined with further natural language processing tools.

Methodology:
The first stage of the project was to design and create the program using Java in conjunction with the Intellij IDE, by implementing the graphical user interface for the user to interact with, as well as the various buttons and their associated functions; namely the ability to load in a file's contents from a PDF using Fontbox, converting said text to a "spun" state with OpenNLP and the Altervista thesaurus, and comparing the spun or supplied text against the corpus using the distance metric to try and detect possible plagiarism.
The next stage was to develop and refine the Levenshtein distance metric using repeated testing of the outcome of the "de-spinning" attempts, to gradually improve its ability to make accurate judgements on whether plagiarism has taken place and reduce the number of false positives generated.

The results gained from testing were then analysed and used to draw further conclusions regarding how the program could be further improved and developed, and what the successes of this project could mean for examiners in the academic field.

Achievements:

The development of a Java program which can accurately simulate the types of changes made by most online text spinning facilities to produce an output of spun text. This spun text can be compared against the contents of academic papers or other sources in conjunction with the use of the Levenshtein distance metric to flag whether the spun or supplied text is likely to have been taken from one of the sources, and how likely it is to be so.

Attestation

I understand the nature of Academic Misconduct as outlined in the University's Academic Integrity Policy and certify that this dissertation contains only original work undertaken by myself during the time of the project.

Signature Sara Wilson **Date** **31/08/2022**

Acknowledgments

I would like to express my gratitude to the following people for all their help in this project, as without them it would not be possible:
Professor Carron Shankland – For constant support, advice, and supervisory work, and for keeping me motivated to keep pushing.
Dr Saemundur Haraldsson – For advice and pointers regarding the types of APIs that could be useful, and what features he would like to see in a de-spinner tool.
Dr Anthony O'Hare – Second marker and excellent feedback and questions throughout demonstration.
GeeksforGeeks.com – With help understanding and implementing the code required for the Levenshtein Distance Metric[1].

Table of Contents

List of Figures

1 Introduction

Plagiarism has been an ongoing issue in both the academic community and wider society for a long time yet has only grown easier to perform with the advent and growth of technology. It is generally aided using “spinner” tools, which claim to be able to offer inspiration to users by rewording or restructuring their content into another form but are ultimately used to get around plagiarism detection methods such as Turnitin. These tools compare the submitted text against other submitted work and academic papers to try and detect where content has been lifted from another source. Unfortunately, these tools are not always accurate, and in some cases, plagiarism can be missed entirely. At this point it is up to the marker to investigate any suspicious content that they find, and manually search for where the content came from, which can take an extremely long time.

Hence, it is important to investigate how best we can streamline the plagiarism detection process, and one such way of doing this is through the creation of a “de-spinner” program. This program can be developed to simulate the behaviour of common spinning tools, and by adding a facility by which we can spin a set of source texts and suspicious text into their root forms and compare their similarities, we can hopefully pick up on stolen content more easily, understand how spinners work in detail, and analyse whether a “de-spinner” could be used to offset some of the workload for examiners regarding manual searching for plagiarised text.

1.1 Background and Context

Plagiarism is the act of taking the work of another and presenting it as if it were one’s own. This can include lifting phrases, sentences, or entire paragraphs of written work, as well as copying the work of another in the sense of code, graphic designs, scripts, and so on. It is distinct from similar terms like paraphrasing and quoting in that credit has been given in these cases, usually in the form of a references page in a paper or similar. Quoting and paraphrasing typically also deals with smaller chunks of work and in the case of paraphrasing, demonstrates a level of understanding of the source material.

According to plagiarism.org, a survey conducted by Rutgers University in the United States found that around 36% of undergraduates and around 24% of graduates admitted to taking content from an online source without giving proper credit or reference, with 38% and 25% respectively taking written content without giving credit.[2] Many students surveyed appear to be driven by a fear of failing a class, disappointing their parents, or because of poor time management skills. Plagiarism in the academic workplace is considered a gross violation of academic integrity, and includes other behaviours such as contract cheating, where a person pays someone to have them do work for them, typically writing an essay, falsification of results or information for gain, cheating on examinations by various means, and assisting others in breaching academic integrity.

According to Our World in Data, the number of people with internet access has risen from less than 10% in 2000, to over 45% by 2016.[3] As a result the opportunity for plagiarism to take place has been growing, partly in thanks to the libraries of free academic and research papers across the internet, but also because of the growing popularity of so called “article spinner” tools (popular ones include WordAI and SpinnerChief), marketed to improve creative thinking by taking a submitted text and rewording it in another manner. The reality is, however, that these tools are almost exclusively used for evading the tools used to automatically detect plagiarism by substituting words for others and shuffling the order of sentences around to slip under the radar of these programs. Detection software such as Turnitin calculate a similarity score by comparing sections of text from a submitted piece against a massive corpus of sources to try and notice content that is similar in terms of flow, structure, and word choice. These are not infallible, however, and can miss plagiarised content if care has been taken to mask it through sufficient changes in sentence structure or wording. When this happens, it falls to the markers themselves to identify suspicious parts of text – but the difficulty for markers lies in *proving* that plagiarism has taken place, and with many papers to mark they may simply not have enough time, and perhaps cannot simply perform a web search and pull up the uncredited source if it has been masked well enough.

Given this growing ease in performing plagiarism, it would be useful to develop a program by which we can investigate how article spinners typically transform data and imitate their behaviours, while also developing a tool which can take this spun text and compare it against those of a sample of sources to detect if any content has been plagiarised. Given the heavy load already placed on academic examiners, a tool such as this could reduce the overhead required in searching each submission for suspicious parts of text that may not have been picked up by plagiarism checking tools and trawling through the internet trying to prove that plagiarism has indeed taken place; hence, I consider this tool to be of importance and relevance to current issues, particularly as work has moved increasingly online in not just the tech sector, given the impact of coronavirus on in-person examinations and learning.

1.2 Scope and Objectives

The main goal of this project will be to develop a program through Java and an appropriate IDE (IntelliJ in this case) which can simulate the types of behaviour exhibited by online spinner tools typically used by students. It will take a submitted piece of text, either typed in or loaded from a PDF file, transforming the text through substitution of words with their synonyms and restructuring of sentences. This technique will be refined through testing of various methods along with support from online APIs until we can find a solution which sufficiently "spins" the text while still preserving as much of its legibility and intent as possible. The spun text from this process will then be taken and converted to its root form by first performing tokenization to split the text into individual words or phrases, and then lemmatizing these "tokens"; where the root form of a word is found – where tenses are removed, and the words are simplified - such as "running" being converted to "run". In addition, synonyms which are less frequently used are converted to their most common equivalent; so, the word "sprinting" might also be changed to "run". This will ensure that our text is in as generic and basic a form as possible, so that when we perform the same operation on a corpus of sample papers, we are able to directly compare the similarities of the suspicious text with the sample papers. This similarity comparison will be performed through use of a distance metric which will be used to judge the efficiency of both our spinning tool, and how reliably we can take a spun text and return it to a form where it can still be identified against the source it was taken from.

A simplified overview of the primary objectives of this project are as follows:

- Using Java in combination with the IntelliJ IDE, create a program which can simulate the behaviours of online spinning tools as closely as possible by:
    - Allowing the user to input some text by hand, or by loading it in through a PDF file, using file reading support from Fontbox.
    - Taking the supplied text and converting it to a "spun" format by transforming the text into individual tokens through tokenization, which are then converted to a randomly selected synonym before the words are recombined, also potentially impacting the sentence structure and flow along with it.
    - Allowing the user to change the step size of each transformation, i.e., whether every single word should be spun, every second word, eighth word, and so on.
    - Outputting the final spun piece of text to the system, which should still be legible and understandable but is different enough to hopefully evade plagiarism detection tools.
    - Refining and testing our program to find the balance between a high degree of both difference from the original, and legibility (note that this process is mainly driven by human judgement, as it is difficult for a program to determine what truly "makes

    sense", and, if it flows with the style of writing, word choice and flow typically adopted by the student.

- Develop a complimentary tool which can take a piece of spun text produced by the program and compare its similarity with a set of sample source papers to check for any plagiarism by:
    - Taking some text that has been displayed in the program, typically by first performing spinning on an existing piece of text, but also can be added manually in the same method to above.
    - Converting this text to its root form by means of tokenization and lemmatization, as above (described in more detail later), before sending these words to a method which selects the most used synonym associated with each word and returning it.
    - Repeating this approach with each of the sample papers in our corpus so that we have both the spun text and the corpus in its most "basic" form.
    - Comparing the "de-spun" text against the "de-spun" corpus texts one by one using a distance metric which calculates the number of edits needed to go from the former to the latter and produces a score from 0.0 to 1.0, with a higher number indicating a high degree of similarity.
    - Outputting to the user whether a piece of text is suspected to be plagiarised, and what paper it may be plagiarised from, if this score is over a certain threshold.
    - Testing and refining this program until we are happy with its ability to narrow down text into its most basic form, and that it can strike a balance between efficiently detecting potential plagiarism while avoiding false positives.
- Further test and develop the program, to ensure that both segments of the program can operate both independently and as part of a whole without any errors and to a significant degree of accuracy.
- Evaluate the performance of the spinning and de-spinning program created by spinning different texts taken from the corpus, varying in size, scope, number of distinct sections, and so on, and recording the similarity scores produced for each, and under what conditions the program is more likely to flag plagiarism.
- Retrospective analysis of the project during the dissertation to establish which areas could be improved upon, and in what ways.

1.3 Achievements

This section looks at the objectives outlined in the abstract, and comments on whether they were met, and what could be done to further improve, given more time.

- Design and construct a program which can take text from a field, or loaded in through a PDF file, and "spin" the text, using the support of APIs to convert words into their root form, and then to a synonym.
    - The spinner portion of the program has been well developed and simulates the behaviour of a typical online spinner to a good degree, specifically when converting words to their synonyms. Some words are not edited, such as those beginning with a capital letter, or "words" which are only numbers, as these could sometimes be interpreted by the dictionary API as being the last names of famous people or similar and ended up affecting the legibility of the text significantly when edited. Despite this correction, there are still some obvious issues with the spun text; suffixes are difficult to reattach correctly and if the chosen synonym does not fit with the style of the suffix used with the original word then it can appear strange, and I was unable to

find a reasonable method of shuffling the structure of sentences and paragraphs while still maintaining a flow to the text that made reasonable sense, as some context would be lost by the program randomly shuffling sentences and discussing a topic before it was explained, for example.
    - There is a degree of difficulty in understanding the spun text on some occasions, due to odd synonym choice, slight changes in sentence structure, and suffixes not working with the new word. However, I believe this to be in part an inherent problem with online spinner tools; only the most advanced and complex tools exhibit a high degree of intelligence when outputting spun text that ensures it continues to flow in the same manner and make sense, with most tools (particularly the most widely available and free ones) producing results that can "mostly" make sense in the same vein as our results. As a result, I believe that we have succeeded in the goal of replicating the typical behaviour of a spinner tool.
    - Given more time, we would be able to investigate in more detail the sentence structure editing portion, tidy up the output in terms of unusual word choice and suffixes, and hence attempt to reduce the similarity score even further.
- Further improve the program by creating a "de-spinner" where the user can compare the similarity of the supplied text and a corpus of sample papers, by converting both to their root forms as far as possible, trimming any formatting differences out, and comparing the similarity of the two using the Levenshtein Distance metric to gauge likelihood of plagiarism.
    - The "de-spinner" works well to convert the given text to its root form; given the difficulties surrounding most of the text not being spun (around 90%), the fact that we are able to convert the spun text and corpus texts to their "root" forms and obtain a similarity score of around 94% is pleasing to see and indicates that the de-spinner tool has worked very well.
    - With more developments in the spinner portion, the de-spinner would need to be expanded to compensate, mainly in terms of shuffling the placement of sentences or passages around to try and undo the work performed by the spinner.
- Develop the Levenshtein Distance scoring metric, which behaves similarly to other distance metrics, but places greater weight on the number of changes required to take one string to another, hence noticing changes in tense or the use of some synonyms more easily.
    - The Levenshtein distance metric was a viable choice for this project, as it focuses on the specific edits needed to go from one word to another. Initially the method was supplied the entire "de-spun" text and compared against the full text of each corpus text, but we found it to be just as efficient by splitting the text into smaller chunks given that the metric compares every character in the first text with every character in the second. This opens the possibility for conducting similarity checks on individual chunks in parallel, potentially saving a lot of time particularly with larger files.
    - It would have been useful to investigate how other distance metrics such as the Cosine and Jaccard similarity metrics stacked up against the Levenshtein distance to conclude whether we made the right decision in choosing it; comparisons of similarity scores returned under different conditions would offer this information.
- Testing and refinement of the distance metric through repeated testing; taking samples of text from the corpus, spinning it (perhaps more than once) and attempting to compare the similarity of this spun text with the corpus as discussed prior – until the level of accuracy of the "de-spinner" has reached an acceptable point.
    - Not much could be done to improve the metric in my view – it worked very well at the beginning of its implementation and any potential increases in its ability to detect similarity would come from further developments in the de-spinner aspect as

well as possibly using other distance metrics such as the Jaccard coefficient in parallel.

- Analysis of the results gained from testing, including what types of spin detection the system is more, or less, capable of, and why.
    - Results that were gathered surrounding the performance of both the spinner and de-spinner (see Section 4) indicated that the spinner had made a good start but that there were many improvements still to be made regarding dealing with capitalised words, changing of sentence structure and flow, and so on. We did however see that the de-spinner performed very well, as did the similarity checker.
    - The latter aspect of this point was unsuccessful for the most part, as we were only able to implement the synonym-swapping aspect of the spinner, and hence it is the only type able to be analysed properly by the system. There was scope to instead use some online spinner tools in combination with our de-spinner to assess its ability but unfortunately there was not enough time to complete this research – I believe that the Levenshtein distance would still work well given how it compares each character in both texts to one another, however.
- Consideration and discussion of the results of the project in terms of what it could mean for the wider academic community, particularly for examiners, if developed further and refined with further natural language processing tools.
    - Overall I believe that this project has been a success, and provides a good starting point from which to further expand this concept; we have shown that the use of a de-spinner in combination with a similarity checker can be excellent at picking up on typical plagiarism behaviours, and if this was expanded further through further refinement of our own spinner tool, using this to test new approaches to plagiarism detection, and moving towards a web-based corpus system offering much more scope, I am confident that this tool could be extremely useful in the academic setting and save examiners a huge amount of time from not having to manually look for plagiarised texts.

2 State-of-The-Art

2.1 Natural Language Processing (NLP)

Natural Language Processing or NLP is a branch of computing with the goal of allowing computers to read and understand spoken or written words in the same way humans do.[4] It incorporates computational linguistics in combination with machine learning to try and understand the meaning of pieces of speech. It can also consider the flow and sentiment of text in detail, rather than just a surface-level understanding of what each word means.

Natural language processing has many applications throughout the industry, including in:

- Speech recognition
- Spam detection
- Machine translation
- Virtual chatbots
- Product review sentiment analysis
- Auto-correct tools

Because of the complexity and number of human languages, NLP is a difficult field since there are always exceptions to language rules which make it hard to consistently determine the meaning of a word or sentence. NLP needs to be able to deal with strange behaviour such as metaphors and sarcasm as well as grammar that might change with local dialect, among many other issues. There are several tasks involved in Natural Language Processing:

- **Speech recognition** is the process of taking voice input and converting it to text. Typically, the voice data is received through something like microphone input but could also be loaded from a sound file or similar. The challenge with speech recognition comes with the diverse ways every person talks; there is no truly consistent way in which words are pronounced, and accents, speech patterns, lisps, slurring, and so on need to be considered when processing the data.
- **Part of speech tagging** is the method of figuring out which category of language (part of speech) a word or phrase fits into. These include things such as nouns, pronouns, adjectives, verbs, prepositions, and so on. The part of speech tag given to a word or phrase can change depending on the context of the sentence surrounding it, which presents a difficult challenge for training NLP programs; for example, the word "running" could be a noun or adjective depending on if it referring to the sport or the activity of operating something, respectively.

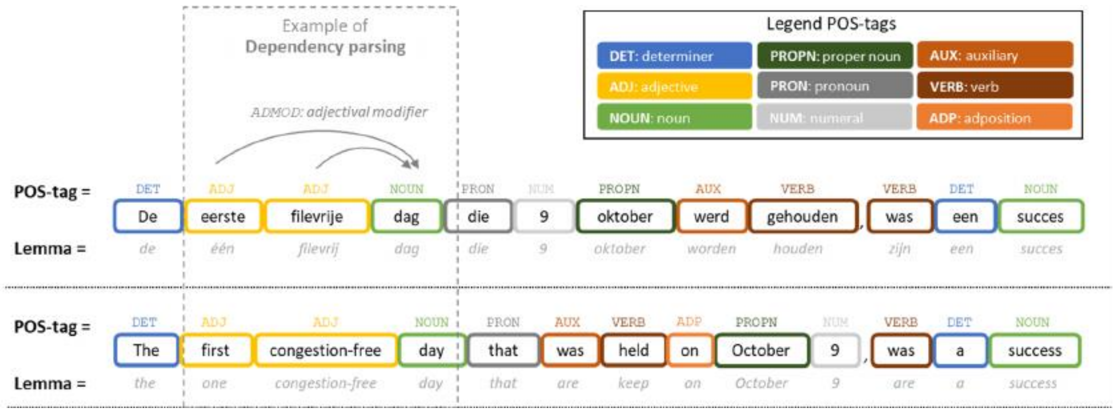

Example of
Dependency parsing
ADMOD: adjectival modifier
Legend POS-tags
DET: determiner
PROPN: proper noun
AUX: auxiliary
ADJ: adjective
PRON: pronoun
VERB: verb
NOUN: noun
NUM: numeral
ADP: adposition
POS-tag =
DET ADJ ADJ NOUN PRON NUM PROPN AUX VERB VERB DET NOUN
De eerste filevrije dag die 9 oktober werd gehouden , was een succes
Lemma =
de één filevrij dag die 9 oktober worden houden zijn een succes
POS-tag =
DET ADJ ADJ NOUN PRON AUX VERB ADP PROPN NUM VERB DET NOUN
The first congestion-free day that was held on October 9 , was a success
Lemma =
the one congestion-free day that are keep on October 9 are a success

Figure 1. Diagram showcasing how POS Tagging looks at a sentence in Dutch versus English.[5]

- **Word sense disambiguation** refers to choosing the correct meaning of a word based on the context of the way in which it is used. This involves complex analysis of semantics (the meanings behind words) which needs to consider the many ways in which words can be used, such as “pen” referring to both a writing utensil and an enclosure for animals, as well as being the shortened form of “penitentiary”.
- **Named entity recognition** is a crucial step of NLP and involves identifying words which are specially important, such as the names of people or places. These words are often capitalised but that cannot be guaranteed, so a natural language processing AI needs to be able to distinguish named entities from “regular” words, particularly in cases of word spinning, where an acronym might be picked up by the system as the shorter form of a longer word, when instead it should be left alone. NER often also considers dates and times as well.
- **Sentiment analysis** is the identifying of aspects of speech such as sarcasm, confusion, doubt, and other emotions from a piece of text. It classifies data into three main categories of positive, negative, and neutral and is widely used in the context of customer feedback analysis; particularly on a large scale to categorise reviews so that the business can make informed decisions about public feeling of the product or service. Because sentiment is such a subjective thing to analyse, this can be a complicated process.

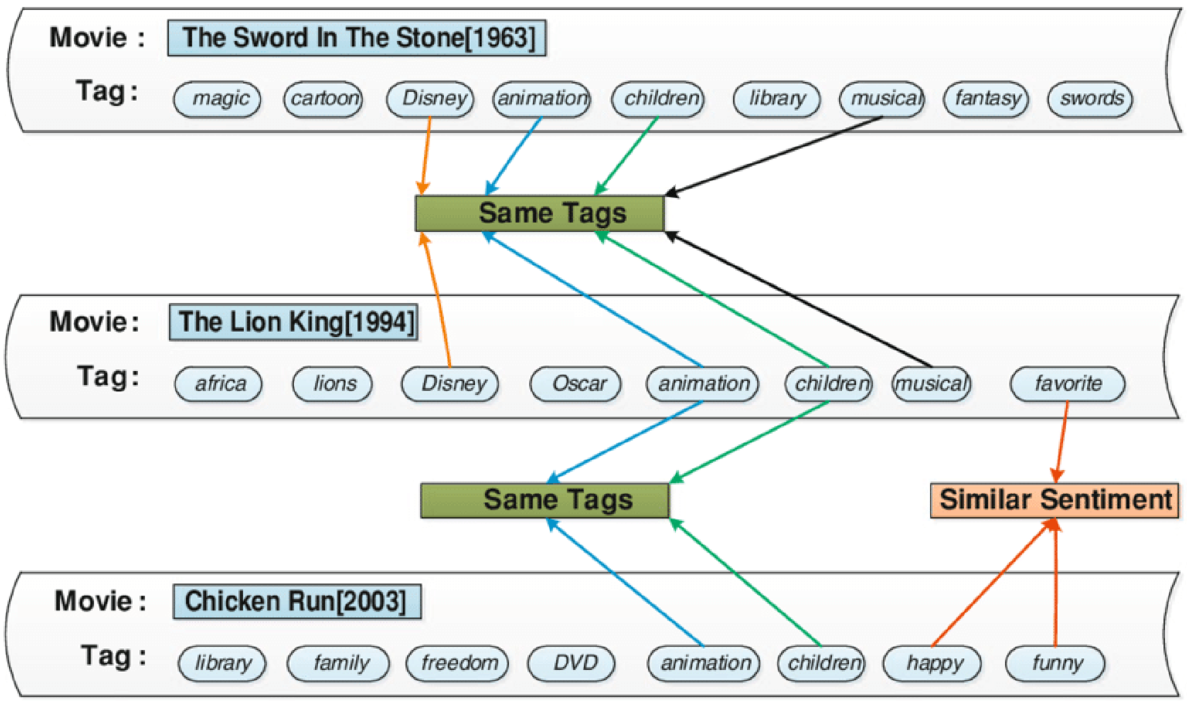

Movie : The Sword In The Stone[1963]
Tag : magic cartoon Disney animation children library musical fantasy swords
Same Tags
Movie : The Lion King[1994]
Tag : africa lions Disney Oscar animation children musical favorite
Same Tags
Similar Sentiment
Movie : Chicken Run[2003]
Tag : library family freedom DVD animation children happy funny

Figure 2. Diagram of sentiment analysis using tags to recommend movies.

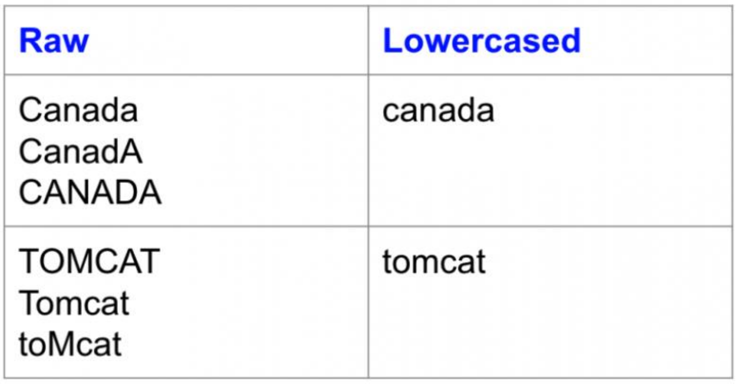
- **Co-reference resolution** is the identification of words which are related to each other. This is usually done in pairing of pronouns with the corresponding person’s name but also includes inanimate objects or businesses. For example, the phrase “Sam went to the bank to see how much money the branch had in her account” ties the name Sam with the word “her” and “the bank” with “the branch”. It can struggle with sentences where two entities share the same pronoun. In the phrase “The bank could not process the customer’s transaction because they did not have the right details”, it is unclear if “they” refers to the bank or the customer, and so requires further context to process correctly, and might mean the NLP system has to assume about the context.

- In addition to the above, there is also **natural language generation**, which is the process of taking information generated by the NLP system and converting it into a readable human format. This process considers all the other aspects of natural language processing including part of speech tagging and sentiment analysis to produce an output that appears as close to recognisable human speech as possible. One example of natural language generation is with online chatbots, who need to respond to human input in as realistic a manner as possible.

All these aspects are difficult to nail down on their own but combining them into a system which can incorporate all aspects of natural language processing and deal with exceptions to the rules at the same time is an ongoing project in the computing science sector, and a particularly important one when we consider the rise of big data and the processing time involved in working with it.

An early stage of natural language processing is the pre-processing, or data cleaning, of data to convert it to a tidy and legible state which is much easier to process.[6] There are many techniques which are generally used in combination with one another, notably a few which were extensively used in the project:

- Converting words to **lowercase** accounts for any errors in capitalisation in the text being treated, for example if the third letter is accidentally capitalised. Converting everything to lowercase means that there is only one version of each word to process which can help to cut down the processing time. In the project this is done at an early stage when dealing with both the corpus of source texts and the spun text to ensure that they can be compared accurately.



Raw	Lowercased
Canada CanadA CANADA	canada
TOMCAT Tomcat toMcat	tomcat

Figure 3. Transformation of words from capitalised form to lowercase.[6]

- **Tokenisation**, as discussed earlier in the paper, converts text into a set of words or short phrases called tokens, which are easier for natural language processing systems to digest and analyse. Within the context of the project tokenization is done before the next stage of lemmatization so that we can pass each individual word or phrase to the dictionary as an array and return the lemmatized form of each.

- With **natural language processing**, stop word removal gets rid of words which have no significant meaning in the text, and which simply add noise to the text that reduce the accuracy. Words of two or less characters were omitted from the spinning and de-spinning process to not interfere with the accuracy of plagiarism detection and improve legibility.

- **Lemmatization** takes words and converts them to their root form, such as "sleeping" and "slept" to "sleep". In the program this allows for a lookup of the word in the thesaurus to find a synonym before the modification of the word to fit as closely with the synonyms as possible. A similar process called stemming takes words to their root forms but does not guarantee that the words are valid, so for example "movie" would become "movi" instead of remaining as it is, since it is already in its most basic valid form. This approach was decided against as while stemming is a simpler process computationally, it did not give the results needed when spinning.

One of the most common tools associated with natural language processing is the Natural Language Toolkit, or NLTK.[7] It is built on Python and contains many libraries for all distinct aspects of natural language processing as described above, including semantic reasoning and the stemming and tokenization used in this project. A large majority of natural language processing tools use NLTK as a base to build from or draw inspiration from, and indeed all the external java libraries used in this project have done so as well, in particular Wordnet, which has an equivalent fork created for Java named JWI which was used here. These libraries will be discussed in more detail later.

It is also important to consider that some NLP systems are trained using supervised learning whereas others are unsupervised; the method is dependent on what type of work is being done. Supervised learning requires that the data points be labelled so that the model can perform tasks such as classification; part of speech tagging requires supervised learning to understand the patterns in which types of words are verbs, nouns, adjectives, and so on. Conversely, unsupervised learning does not require labelled data and as a result is generally easier to set up, but more computationally expensive; word prediction tools such as those seen on mobile devices are trained with unsupervised and unlabelled data as there are no classifications or categories to include the data in.

2.2 External Java Libraries

A handful of external Java libraries were pivotal in their role of supporting the development of the both the spinner and de-spinner programs. This section will very briefly discuss in how each operates and the backgrounds behind them.

2.2.1 PDFBox

PDFBox is an open-source tool supported by the Apache Software Foundation which allows for Java to interact with PDF documents.[8] Typically, native Java file handling methods cannot process PDF files as it does not understand the way in which PDF files are constructed. PDFBox allows Java to both read and create PDF files, as well as fill out forms, digitally sign, and merge multiple PDF files together.

This project used Apache PDFBox 3.0 to read in the text from PDF files for spinning and similarity checking, as most academic papers available online are presented in PDF format, although the program can still handle plain text being typed into the system.

2.2.2 OpenNLP

Apache OpenNLP is another Apache Software Foundation assisted tool which offers an open-source machine learning tool for natural language processing.[9] It contains facilities for common NLP tasks such as stemming, tokenization, part-of-speech tagging, sentiment analysis and named entity recognition. It effectively acts as a platform that other NLP programs can build upon to reach their goals.

OpenNLP can be used through Java, or the command line as needed. A full documentation of the tool can be found at their website listing all its features along with a brief example of how to use it. The program in this project makes use of the tokenization, lemmatization, and part-of-speech tagging parts of the tool. There was scope and opportunity for use of aspects such as the sentence detector and name finder however there was not enough time to explore these in proper depth. This will be further discussed in the evaluations section.

2.2.3 JWI / Wordnet

WordNet is a database created and supported by Princeton University that contains almost the entire English lexicon.[10] It consists of synsets of nouns, verbs and adjectives which each signify a different concept. These are structured as trees based on lexical relationships which help to link distinct aspects of the English language together. It works on a higher level than a normal thesaurus as the tree-based network can group words nearby that might be semantically quite different but can be used in a similar context. There are over 117,000 synsets in WordNet that form these trees and each synset has a definition for each word or phrase, with different meanings of the same word being in different synsets.

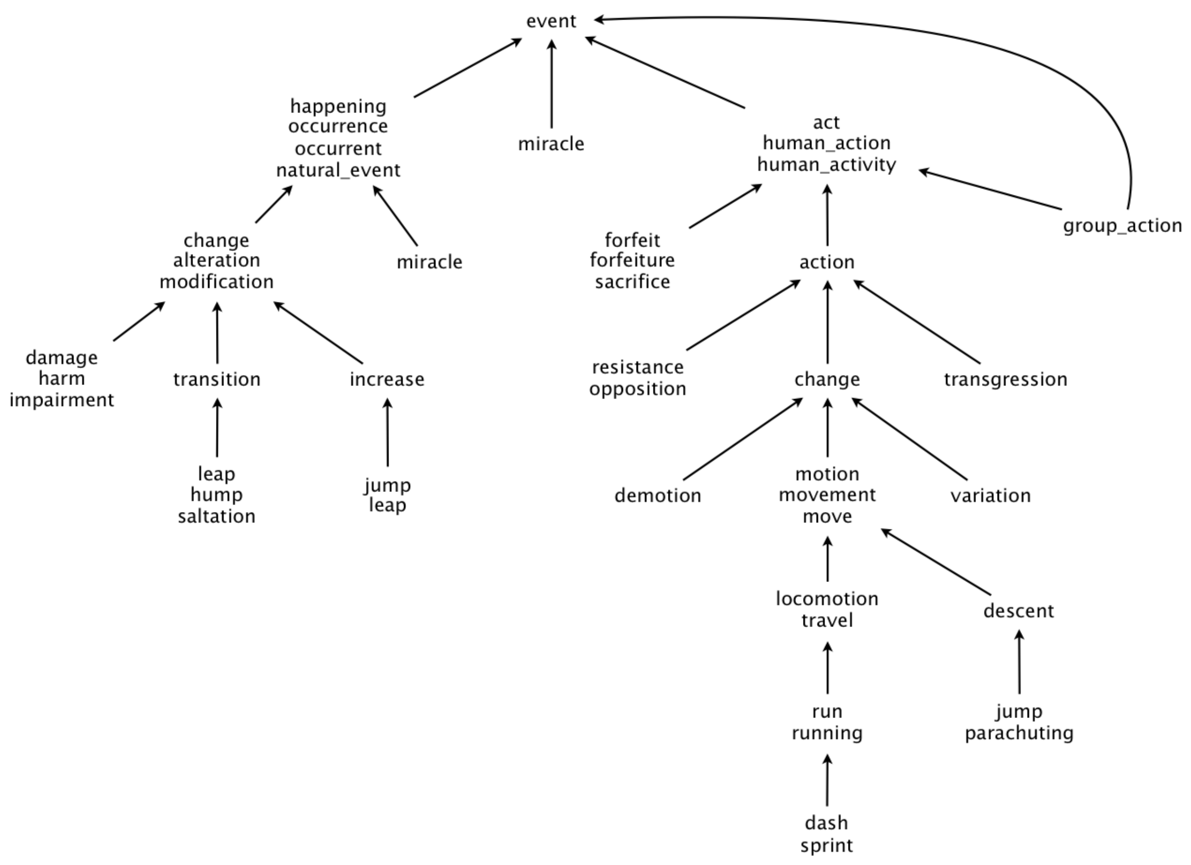


Figure 4. An example of the tree-based network of synsets.[11]

Wordnet uses what is known as super-subordinate relation to link more general synsets such as {furniture, piece_of_furniture} with specific ones such as {bed} or {table}. There are specific synset relations for verbs, nouns, and adjectives but information on these can be found on the website and in documentation.

JWI (The Java WordNet Interface) builds upon the database provided by WordNet to allow for Java to interact with it directly.[12] It uses API calls to retrieve synsets and definitions from the dictionary for analysis and allows for browsing by semantic or lexical indexes. In addition to allowing for remote access to WordNet through Java, the dictionary can also be loaded into the local system for much faster access. The local version is accessed through a Dictionary object whereas the remote version uses an IDictionary object. JWI was used in the project to allow for the spinning and de-spinning of text once it had been converted to a tokenized and lemmatized state.

2.3 Previous Research in This Area

A crucial step in any research project is the identification and discussion of the work that others have performed in this area surrounding de-spinners and the related topics. This allows for us to identify how our work fits in alongside other similar research, and what could perhaps be explored in more depth.

Firstly, research conducted by *Zhang et al*[13] in 2014 investigated the role that spinner tools have in the field of search engine optimisation. They raised that of a set of 427,881 pages from wiki sites which were prone to abuse, 52% of them were auto generated or spun articles. They found that there are many spam-style wiki pages which are generated from translation of other spun articles to a different language. This paper performs an in-depth analysis of the popular spinner tool simply named "The Best Spinner" and found that it utilises a continually updating synonym dictionary to spin its contents into another form and offers the ability to select the frequency at which words are spun, and whether or not to omit the original word from the synonym list when choosing a new word.

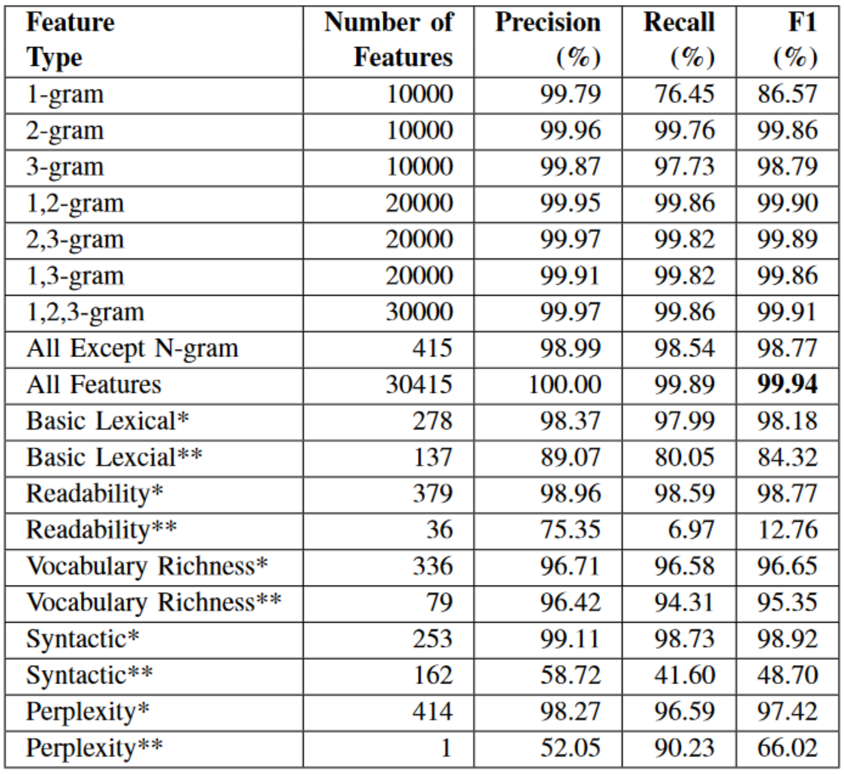
They create a tool to help perform de-spinning and similarity checking of spun SEO texts. Here, they use the Jaccard coefficient to gauge the level of similarity between two sets of text. It is a straightforward way of performing similarity checks and uses the below formula:

$$\frac{Words(A) \cap Words(B)}{Words(A) \cup Words(B)}$$

Figure 5. Jaccard Coefficient Formula.[13]

In the case of comparing two sets of text, it counts the number of words which are shared between both sets and divides it by the total number of unique words across both sets. Compared against the Levenshtein distance used in this project, I feel the Jaccard coefficient to be a less effective method of evaluating similarity as it does not take into account the positioning of words when counting them, and hence could result in some high similarity scores between two texts that happen to share words with one another but hardly any relation in terms of position or meaning. I believe that because the Levenshtein distance metric looks at the changes required to go from one form to another for each word, phrase, or paragraph, that it makes a more effective indicator of how likely one piece of text is to being plagiarised from another.

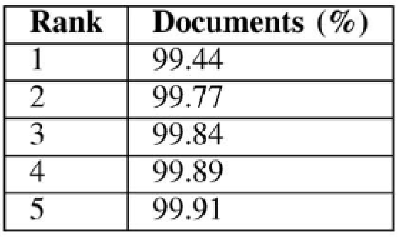
A similar paper by *Shahid et al*[14] also mentions The Best Spinner as a common tool for SEO-oriented plagiarism as well as acknowledges the work of *Zhang et al*[13]. They point out the reverse-engineering strategy used by DSpin no longer works effectively, as the dictionary used by The Best Spinner is now cloud-based, and hence cannot be accessed as easily. They raise that a number of spinning software have taken similar measures to protect their software from reverse-engineering; this raises a question as to why these tools are so concerned with protecting against reverse-engineering as quickly if they are being used for legitimate purposes. This paper presents a method of detecting spun documents along with the works they are plagiarised from without having to access the synonym dictionary used by the spinner tool. From a random sample of 41,877 spun articles from the roughly 150,000 in the full set, they perform classification against the rest of the set using different feature detection methods. Here the paper finds that n-grams are the most successful when being used for detection of spun documents, with readability being the poorest.



Feature Type	Number of Features	Precision (%)	Recall (%)	F1 (%)
1-gram	10000	99.79	76.45	86.57
2-gram	10000	99.96	99.76	99.86
3-gram	10000	99.87	97.73	98.79
1,2-gram	20000	99.95	99.86	99.90
2,3-gram	20000	99.97	99.82	99.89
1,3-gram	20000	99.91	99.82	99.86
1,2,3-gram	30000	99.97	99.86	99.91
All Except N-gram	415	98.99	98.54	98.77
All Features	30415	100.00	99.89	**99.94**
Basic Lexical*	278	98.37	97.99	98.18
Basic Lexcial**	137	89.07	80.05	84.32
Readability*	379	98.96	98.59	98.77
Readability**	36	75.35	6.97	12.76
Vocabulary Richness*	336	96.71	96.58	96.65
Vocabulary Richness**	79	96.42	94.31	95.35
Syntactic*	253	99.11	98.73	98.92
Syntactic**	162	58.72	41.60	48.70
Perplexity*	414	98.27	96.59	97.42
Perplexity**	1	52.05	90.23	66.02

Figure 6. Spun document detection success by feature type investigated.[14]

Cosine similarity (a vector space model which measures the “angle” of similarity between two vectors, in this case the two texts)[15] is then used to compare plagiarised text against the top five matching documents which found that there is a 99.9% chance of the plagiarised text being detected within the top five classified documents. This research suggests that vector space models are a highly effective method of identifying the similarities between two texts, so it would be worth seeing how effective our model is in comparison.



Rank	Documents (%)
1	99.44
2	99.77
3	99.84
4	99.89
5	99.91

Figure 7. Percentage of matching documents found by rank position using cosine similarity VSA.[14]

A paper written by *Shkodkina et al*[16] and published in 2017 investigated the most common causes of plagiarism within Ukrainian academic institutions, finding that in 49% of surveyed academic submissions, they contained paraphrased text from a source rewritten in their own words without giving proper reference to said source, 37% of papers used downloaded essays from online that were claimed to be their own, and 31% engaged in copying and pasting style plagiarism of content straight from other sources. This highlights an issue with plagiarism detection surrounding the finding of the papers the plagiarism is tied to – in most cases plagiarised content is not credited, so of course we find that academic staff must spend a large amount of time scouring the internet for matching papers to compare for plagiarism, taking up a huge amount of time.

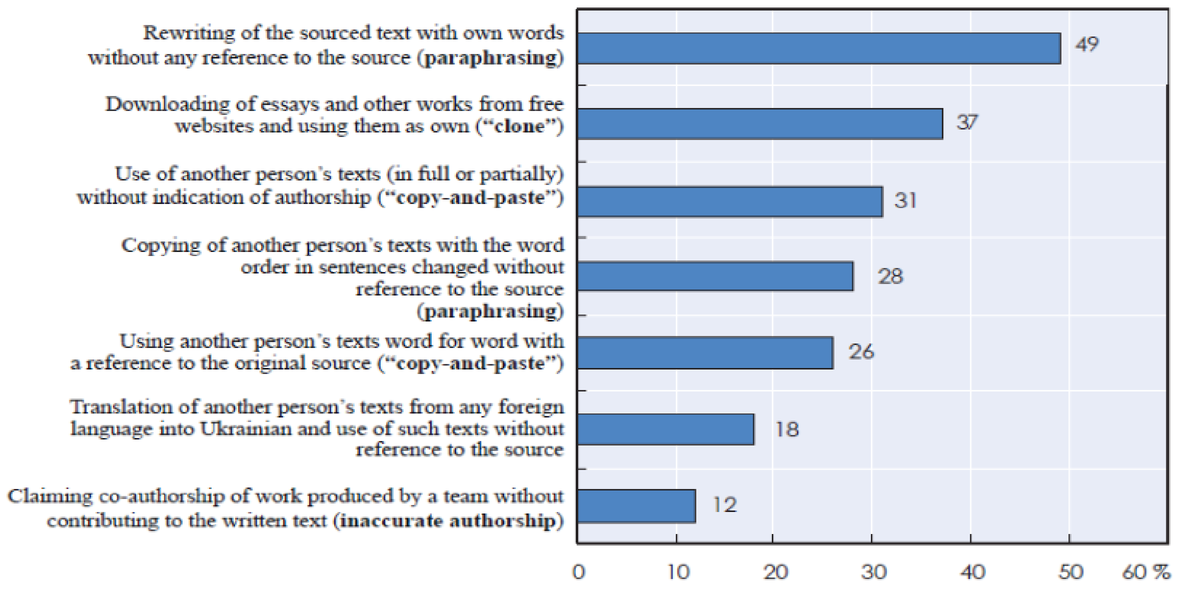


Figure 8. The frequency of varying forms of plagiarism within Ukrainian higher education.[16]

In addition, they also investigated the features that Ukrainian higher institutions felt were most important in anti-plagiarism software. The most desired features included a percentage showing how much of the text was original (93%), the ability to upload files or documents (86%) and providing reference to the primary sources found (70%). While the program from this project does not show how much of the text was original per se, it does use the Levenshtein Distance metric to calculate the level of similarity between the corpus texts and the suspicious text. In addition, it supports the ability to both type and upload text as well as highlight the similarity scores for each source which can at least point the user in the right direction to find the source by searching the name. For this reason, we can say that our program fulfils a good number of the most desired requirements for a plagiarism detector.

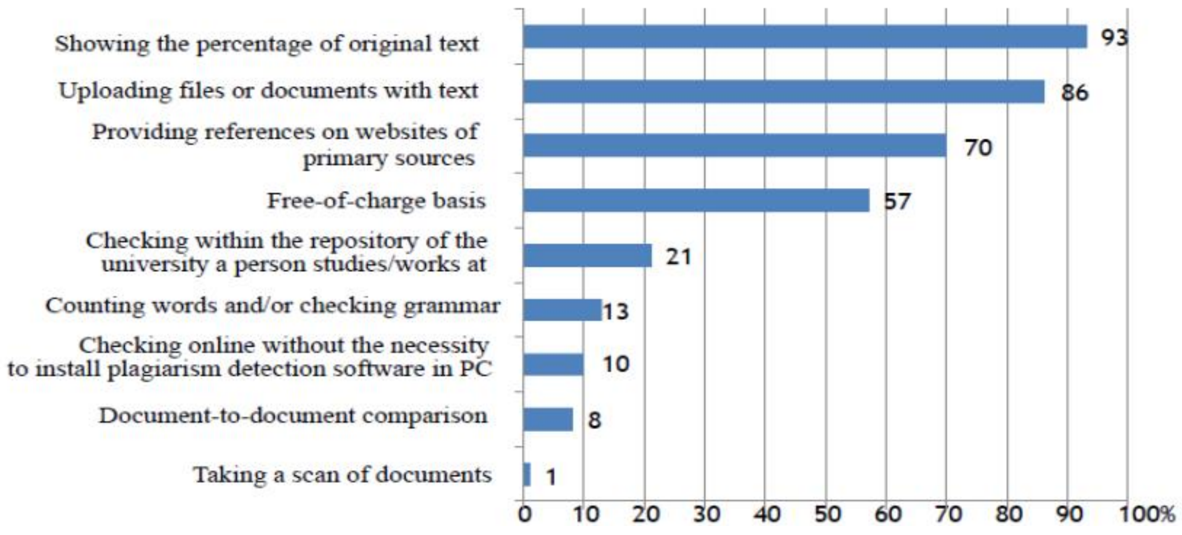
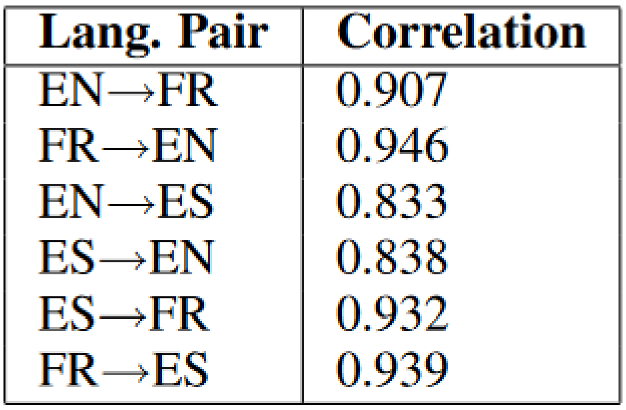


Figure 9. Survey results of Ukrainian higher education re. most notable features of plagiarism detection software.[16]

The report also highlights a number of features that Turnitin possesses. It highlights that Turnitin has the largest database of papers of any plagiarism detection software, including aggregating all the papers submitted by users for checking, giving it a big advantage over other systems. In addition, the paper found that it appears to have the ability to spot translated plagiarism, where whole sections of text are translated from a foreign language into another. This highlights a potential future idea to develop with our program, perhaps allowing the user to make a call to an online translation software such as Google Translate to try and pick up on translated plagiarism.

Research conducted by *Ferrero, Agnes et al*[17] builds upon the findings discussed above, as they investigated in more detail cross-language plagiarism and potential detection methods. By using a dataset consisting of English, French and Spanish texts through a mix of Wikipedia articles, scientific papers, and product reviews of which are both human and machine translated, they tested various cross-language plagiarism detection methods. They found that methods which are effective across a particular pair of languages are likely to be the most effective between other language pairs as well, so long as enough of a database exists for both languages. This shows that it is likely that most plagiarism detection tools (including potentially our own) are liable to work across multiple languages provided they can understand in enough detail the manner in which they work. The Pearson correlation coefficient looks at the level of correlation between two sets of text or other data. The results of the Pearson correlation analysis highlight the level of similarity between the different languages in terms of cross-language spinning method.



Lang. Pair	Correlation
EN→FR	0.907
FR→EN	0.946
EN→ES	0.833
ES→EN	0.838
ES→FR	0.932
FR→ES	0.939

Figure 10. Cross-language Pearson correlations between chunk/sentence granularity.[18]

In the context of code plagiarism detection, we can briefly look at the paper written by *Balaji et al*[19], which highlights the six stages of program modification during plagiarism as stated by *Faidhi et al*[20]:

0. The full original program without any modifications made
1. The only changes are superficial, such as comments
2. Identifier names are changed
3. Variable positions are changed
4. Constants and functions are changed
5. Entire loops and if/then statements are changed
6. Complete control structure change, difficult to detect plagiarism at this point

These stages highlight that while code plagiarism may seem very different, it does still use the same type of approach when being plagiarised, which supports the earlier discussion surrounding the usability of plagiarism checkers cross-language; there is no reason why a similar method to those employed when checking for plagiarism in written works cannot also work by comparing two sets of code.

Finally, we can look at the paper produced by *Foltynek et al*[21], which acts as a comprehensive literature review of all topics relating to academic plagiarism detection. It highlights that there are two primary types of plagiarism detection system; extrinsic approaches which compare the suspicious document against the corpus of non-plagiarised works to find all documents with a similarity above a threshold, and intrinsic detection systems, which specifically looks at the input text to try and find unusual changes in the text regarding writing style, flow, word choice, and so on. It raises that is impossible to compare every document available across the internet against a suspicious piece of text in a reasonable timeframe to try and find a match. Extrinsic plagiarism detection is split into two main stages of candidate retrieval and detailed analysis. Candidate retrieval refers to the finding of documents with content that lines up with that of the suspicious document, and for this reason it might be beneficial to keep a corpus of papers in one sector rather than to search the entire internet so as to save time. The detailed analysis stage should take the set of matching documents, split them into fragments, and compare each fragment against each fragment of the suspicious document to identify the similarities between each pairing. They highlight that there should be three stages to detailed analysis; **seeding**, where parts of the suspicious text are found within the set of matching documents, **extension**, where the matching parts are extended outward in an attempt to find the full paragraph or section which has been plagiarised from, and finally **filtering**, where some tidying is performed on fragmented parts of text that cross over or are too short.

This paper also highlights the various plagiarism detection methods that can be used along with the cross-section of which types of plagiarism can be detected using them.

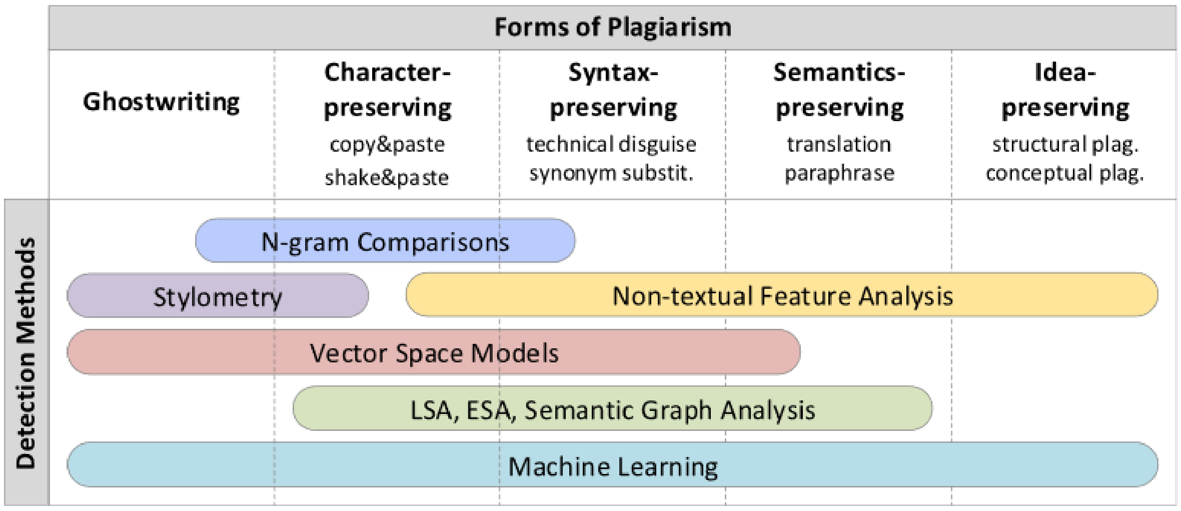


Figure 11. Chart showing the suitability of different detection methods re. plagiarism.[21]

In the context of this project, we primarily use N-gram comparisons to judge whether one set of text is like another, although one could argue this ties in with machine learning methodology due to the use of natural language processing packages like OpenNLP to split up and organise the data into tokens and lemmatised words. Since we only look at text and not images or graphs, we cannot perform non-textual feature analysis and so lose some ability to pick up on plagiarism in that sense, although the concept of the de-spinner aims to mitigate the issue of missing semantic and idea-preserving plagiarism by converting everything to a root form and looking for similarities in meaning in that sense.

Overall, it is clear to see that the field of plagiarism detection has been widely researched, even from the small selection of academic papers analysed. There has been investigation into the typical factors a plagiarism detector should have, which methods work for specific types of plagiarism, and how plagiarism detection methods are able to work between different languages using the same techniques provided that there is support. There is however little research into the use of de-spinner tools to aid in plagiarism detection, and thus I believe that this project can fit into a niche that would also allow for expansion upon the topics covered in this literature review by incorporating them into a practical program – for example the fragmentation of the texts into separate chunks and the cross-comparison of each using our similarity metric.

Later in this paper (see section 4) there will be discussion on the missed opportunities and potential improvements that could be made to improve this project, including by perhaps incorporating translation software in some manner for cross-language plagiarism detection along with using web search to locate additional research papers.

3 Problem description and analysis

This section will explore in more depth the process involved in this project, from the initial planning of stages, to how they were implemented, and finishing with an evaluation of how the process went. We also highlight how this project follows general computing principles as set out by the British Computer Society (BCS), in section 3.4.

The goal of this project is to develop a tool which can aid with the task of detecting plagiarism in academic works by taking a piece of submitted text and comparing it against those of a set of stored academic papers in a corpus. Currently academic markers devote a large amount of time to sifting through individual academic submissions and if they find a section which seems out of place in terms of complexity or type of word choice, or general flow, they must then search for this suspicious text across the internet, which cannot always be found simply by copy-pasting the passage into a search engine and is hence a very time consuming process. If this time is not taken to carefully check for plagiarism and follow leads, then there is the risk of students succeeding in academic misconduct and presents an unfair disadvantage for those students who are honest and responsible. If less time were spent on searching for and investigating plagiarism then academic markers would be able to dedicate more time to other tasks or responsibilities, so for this reason I feel that it is crucial to create a tool to aid with this.

The tool will use a GUI and will allow the user to take a piece of text, either typed or loaded in through a PDF file and display it to the system. They will then be able to spin this text into another form using a button which takes the loaded text and converts it to another form which should as a result be less likely to be flagged by plagiarism detection tools than the original text. The behaviours programmed into the spinner tool aspect should be able to imitate to a good degree the types of results seen when using popular online spinning tools to convert text; the more closely it can resemble these behaviours, the more effective we will be able to develop and optimise the de-spinner tool. The de-spinner tool itself will take a supplied text (perhaps text that has just been spun by the system) and convert it to its most basic, or root, form, by attempting to cut down the lengths of sentences and paragraphs as far as possible and converting each word to their most common synonyms. This 'root form' text is then compared against the texts of a set of corpus papers which are also converted to their root form. They are compared by means of a similarity metric which grades how many changes are present between the two and determines a percentage likelihood that the submitted text has been plagiarised from each individual text.

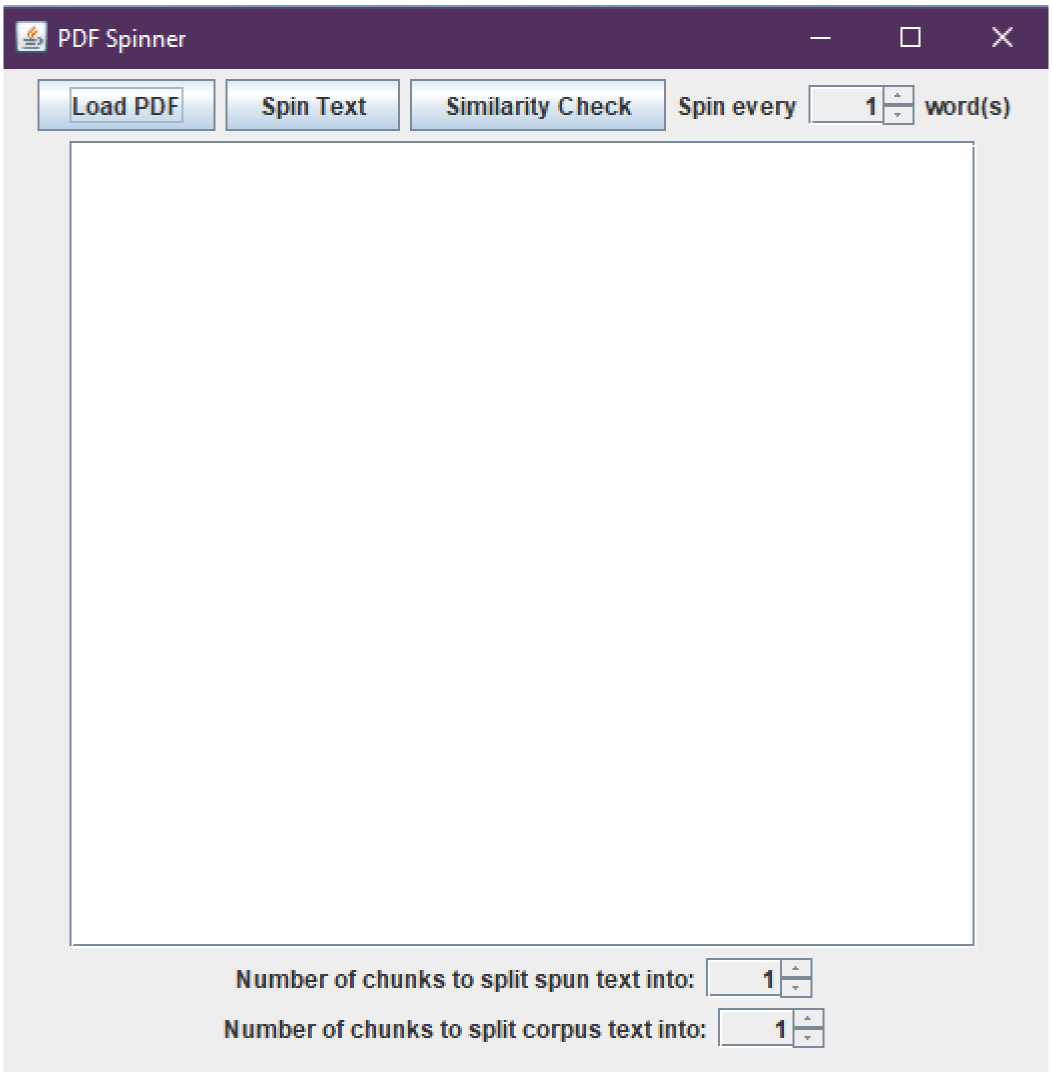
3.1 Spinner Program

3.1.1 Plan

The spinner tool makes up around half of the overall program and will perform a wide range of functions to convert a given text into another form and imitate the types of results seen with online spinner tools as closely as possible. The user will be able to click a button which allows them to load in a PDF file; the text from this file will be printed to a text area on the GUI, which can also be typed to if the user would like to make any edits or submit their own portions of text. Upon clicking a 'spin' button, the contents of this text area will be taken by the program and sent to a method which iterates through the words and converts them to one of a list of synonyms; which one is selected will likely be randomly generated. The goal will also be to take groups of words or sentences and reorder them around a point, such as a conjunction word or piece of punctuation, in a way which leads to a paragraph saying more of less the same thing, albeit in a different order. The program will output the spun text to the text area, along with execution time for the spinning process, and the number of words changed compared to the total word count of the text.

When converting the text to a spun form, the program will need to first split the text into individual words through tokenisation, before stripping them down to their most basic form through lemmatisation. During this process, the suffixes of the words will need to be stored so that they can be reattached once the words are converted to synonyms, to not affect the readability of the final spun text. The spinner tool also allows the user to select the percentage of words they would like to be spun by changing a 'step size' counter – a step size of one means every word is spun, two means every second word, and so on. This will give the user some freedom in how dramatically they want to change their text and might offer a trade-off between readability and concealment from plagiarism detectors. The spinner will also need to account for formatting and where possible should try and maintain the same use of line separators, tabs and breaks as the original text, so this will need to be tracked somehow.

3.1.2 Implementation



Spin every 1 word(s)

Number of chunks to split spun text into: 1

Number of chunks to split corpus text into: 1

Figure 12. A screenshot of the spinner and de-spinner tool when first loaded.

The program GUI is fairly simplistic and contains the facilities for both the spinning and de-spinning/similarity checking aspects. There are three JButtons; **Load PDF** opens a file explorer menu which defaults to the corpus folder as specified in the code, with a restriction to only showing and allowing selection of PDF files. **Spin Text** triggers the spinning aspect of the program, and **Similarity Check** calls the methods which manage the de-spinning and similarity checking process. There are three JSpinner elements as well, the first of which controls the step size of the spinning, i.e., how many words to skip over – a value of 1 simply results in every word being spun while four will spin every fourth word, for example. The other two JSpinners control the number of segments to split both texts into, meaning the user can control the balance between the scope of each comparison and the potential accuracy; this will be further explored when discussing the de-spinner aspect below. None of the JSpinners can be edited other than through use of the arrow keys to prevent any incorrect values from being supplied and causing an error in the program.

When the user selects a PDF using the **Load PDF** menu (controlled through the launchFileChooser() method using a JFileChooser), the file address is selected and passed to the loadPDFText() method. This method makes use of the Fontbox/PDFBox external library[8], created to aid in Java file handling operations, by creating a PDDocument object using the selected file. The library's PDFTextStripper() class is then called to get the text from the loaded in document and return it to the calling Load PDF button event handler, where the text is then printed to the JTextArea. We include a JScrollPane attached to this field so that we can scroll through the full text, since it would never all fit on the screen at once.

Upon clicking the **Spin Text** button and calling its event handler, the first stage is to store which words in the text contain a new line character, so that we can restore the text to as close to its original formatting as possible when the spinning is finished. The text from the field is split up into individual array elements on every space, and for each word containing a new line character, it is added to the wordsWithNewLine ArrayList. The next stage is to perform tokenisation on the full text; this is the process of splitting the text into individual words or short phrases known as tokens. This is fulfilled by the OpenNLP library, which is used to support the natural language processing aspects of the program.[9] The text is passed to the tokenizeText() method which first fetches the English-language version of the tokeniser and then tokenises the text using the TokenizerModel class from OpenNLP before returning the tokenised text to the event handler. Next, we call our getSuffixes() method in the CompareStrings class to find and store the suffixes of the tokenised words so that they can be added after the spinning process. The getSuffixes() method simply iterates through each of the tokenised words and if the end of the string matches one of a set of suffix rules, then it is stored in an array which is then returned to the calling method. Again, we make use of OpenNLP, this time to lemmatise the tokenised text, calling the lemmatizer() method in the LemmingFramework class. This method starts by taking each word and attempting to find its part of speech tag using the posTagger from OpenNLP; these tags are stored in an array and passed to the OpenNLP lemmatize() method along with the tokenised text so that it can find the correct lemma for the context in which the word is used, occupying an array with the lemmatized versions of each word, except in the case where a lemma cannot be found, in which case it is left as it was. Lemmas are used as opposed to stemmed words as the latter are not always guaranteed to be dictionary words and so are difficult to reliably find synonyms for. The lemmatised text is returned to the event handler method and passed to the editWords() method, along with the value of the JSpinner for controlling step size as discussed earlier, and the value 'false' for isDespinning, as this method operates differently when being used by the de-spinner to convert the text to its most "basic" form.

The editWords() method iterates through each word in the array; if the modulus of its position in the array divided by the step size is zero, then the method knows that is on a word which should be spun, otherwise, it will simply leave the word alone and move on to the next one. We then check if the word begins with an upper-case letter, as if it does it should not be spun. The reason for this is that we found during testing that names of people and places would often be interpreted by the thesaurus as being the last names of famous people or abbreviations and would edit them to values which were wildly unrelated. It also aborts the spinning of the "word" if it is simply a number or if it is a stop word, that is, a short word such as a conjunction which contributes little to the overall meaning of the sentence. Words which pass these checks are then passed to the getSynonyms() method to fetch a synonym list for that word. Here we again find the part of speech tag for the word and use it to find a matching index word in the dictionary, based on this tag's value. If we cannot find a matching index, then there are no synonyms for this word, and we return null. Otherwise, the JWI library used to support the Wordnet API is used; we find a synset containing the lemmatised synonyms by passing the word ID to a dictionary object and extract the lemma from each entry in the synset, returning them to the editWords() method. Now that we have the set of synonyms for the word, we randomly select one from the list and replace the original word's position in the array with this new value. If the synonym list is empty, then we leave the word as it is. Once all words have been converted to a synonym, we return the changed array to the event handler.

The next stage of the process concerns reattaching the stored suffixes to the new words as best as possible. The reattachSuffix() method performs this function, iterating through each word in the spun text array and attaching the corresponding suffix at the same array position as it using a set of rules laid out by yourdictionary.com[22]. These reconstituted words are then returned to the event handler which performs some final tidying by clearing the text fields, joining all the array elements into one string using the String.join() method, and replacing each comma with a space before printing the final, spun text to the text area.

3.1.3 Evaluation

To assess the quality of the spinner tool, we made use of a custom-made assignment on Canvas which allowed me to submit some documents for similarity checking and use the returned score to assess the effectiveness of the program's article spinning. I first spun each document before pasting the results into individual .docx files and submitting these to Canvas, recording the results in the Results.xlsx Excel spreadsheet (The full file names can be found in the spreadsheet).

File Name	No. Words	Similarity Score
A	4313	94%
B	17277	88%
C	5668	15%
D	11198	93%
E	1738	91%
F	3994	88%
G	5817	87%
H	4957	92%
I	6298	93%
J	11848	89%
K	11441	92%
L	7066	90%
M	761	81%
N	11489	93%
O	2471	79%
P	5823	91%
Q	4550	79%
R	8244	92%
S	8142	89%
T	7541	93%

Table 1. Turnitin similarity score results for spun articles.

These results show an obvious outlier in paper C, with only a 15% similarity score. Given that the Turnitin report for this submission showed mostly only items such as place or institution names being flagged by the similarity checker, we can likely attribute this low score to Turnitin not having access to this paper within its corpus, otherwise we would see a much higher score. The average similarity if we discount this outlier is 89%, showing us that the spinner does at least succeed in masking some of the plagiarism taking place, but only a little over 10%. I believe this to be because of the restrictions placed on the number of words which are able to be spun because of avoiding capitalised words or stop words shorter than three characters – with academic papers there are a large number of technical terms and other capitalised words, so if we could find a way to work around the issue of names or abbreviations being introduced as synonyms, then I believe the spinner would be much more effective at covering the full text. Further analysis of the spinner tool in conjunction with the de-spinner follows in section 4. It is also worth noting that there is no apparent correlation between the word count of the document and the similarity score produced by Turnitin, as seen below:

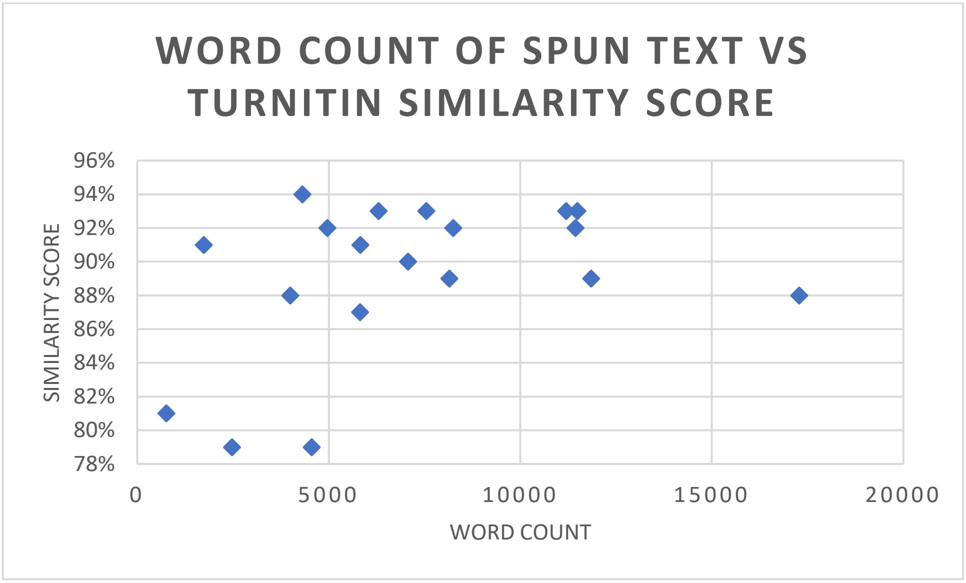


Figure 13. Word count vs Turnitin similarity score of spun text.

This indicates that Turnitin operates in a way in which the size of the text makes no difference to its ability to detect plagiarism, and so it might be important to further investigate the swapping of sentences, changing of tenses and so on, and how they impact the performance of Turnitin.

There are some aspects in which there is a large amount of room for improvement or introduction of new features. Firstly, when conducting the similarity score testing with Turnitin above, I would have liked to perform multiple runs of each to get a more reliable result, but Turnitin can be rather slow to return a score at times. Secondly, as mentioned elsewhere, a far more successful spinner tool would see proper implementation of the synonym-switching aspect, including dealing with abbreviations and names, and include other aspects of common spinners such as sentence swapping or tense/sentiment editing.

Overall, I feel that the spinner tool has made a successful start, but there is still a lot of room for improvement. It can accurately replicate the manner in which word are changed to their synonyms and through use of suffix rules we are able to ensure as much of the text continues to make sense as I believe is achievable. However, we were unable to reliably change capitalised words without significantly affecting the readability of the text, nor were we able to develop a system for swapping sentences and paragraphs around, and I believe these are some important next steps to take to move the spinner tool further as we seek to create a tool which can be used by the de-spinner to evaluate its own performance.

3.2 De-Spinner and Similarity Checker

3.2.1 Plan

This is arguably the primary focus of the project and hence is likely to be the part which takes the most time to perfect and refine. The idea behind the de-spinner is to take a piece of spun text and convert it to a form which sees the typical spinning behaviours undone and allows for a comparison between this de-spun text and whatever source it may have been lifted from, through use of a similarity metric.

The de-spinner portion will use the same program and GUI as the spinner portion to allow them to follow on from one another if needed. There will be a corpus of sample papers as a folder that the user can store PDF files in, which are used for similarity checking with the de-spun text. It is unlikely that there will be time but it would be beneficial if possible to construct a facility for fetching papers from an online-stored corpus or by directly searching for parts of the de-spun text using a search engine, as this would make the program significantly more portable and lightweight by eliminating the need to store all of these papers on a local system. The user will be able to click a button on the interface which initiates the similarity checking process on the text currently in the field. This will make a call to a method which takes the contents of the field and tokenises it into individual words, before lemmatising them, as with the spinner process. We will need to ensure that the words are converted to their most ordinary form, so the plan is to fetch the list of synonyms as normal and find some way (perhaps with the help of an API) to find the most commonly occurring word in the English language from that set of synonyms and choose it to be our replacement word. When all words are converted to their 'most common' form, the text can be reconstructed – there is no need to track suffixes or similar as we are trying to reach a basic form.

Once both the suspicious text and corpus text are in their root forms, the program will then pass these texts to a similarity scoring metric; initially we will code for only the Levenshtein Distance metric, but if there is time it would be beneficial to include other similarity scoring metrics to contrast their abilities at noticing plagiarism. The Levenshtein Distance takes the two texts and analyses how many changes need to be made to take one text to another through deletion, addition, or substitution of characters. A higher score should indicate a higher degree of similarity and hence indicate how successful the chosen spinner is, whether it is the one from the program or some other online tool. The 'root form' suspicious text will be compared in this manner against those of every corpus text, returning to the user a similarity score after each comparison, and whether it is likely to have been plagiarised from that source. We will need to investigate an appropriate threshold value for the similarity score to both catch instances of faint plagiarism without giving too many false positives, where no plagiarism has actually taken place.

The de-spinner will also allow the user to select how many chunks to split both the corpus texts and suspicious text into. The idea behind this was that it may be more accurate to conduct comparisons between smaller sections of the corpus and the corpus texts as a whole, as I feel that comparing the entire document to another might not result in a high similarity score if only a small section has been plagiarised.

3.2.2 Implementation

As with the spinner, the de-spinner and similarity checker portion is triggered from the same GUI. There is no facility for de-spinning a piece of text without performing a similarity check as I did not feel that it was worth including or taking up space in the interface, so they come as one piece.

Upon clicking the **Similarity Check** button, the event handler creates an array containing the list of files from the corpus folder (specified in the program) and iterates through each one, performing the de-spinning and similarity checking process on each. We start by performing tokenisation on both the corpus and text field texts, through the tokenizeText() method as per the spinner method discussed above. We then call the editWords() method on these tokenised texts but pass the value of 'true' for the isDespinning argument. This results in the editWords() method executing as normal, but instead of randomly selecting a synonym from the list to change a word to, we select the first synonym in the array. Analysis of the way in which JWI returns the synset and by extension the synonym lists shows that every synonym of a word shares the same array; that is, all the synonyms are in the same order for each. Because of this we can always select the first entry and use this as our "root form" word.

Once we have these root form spun texts, we need to create a copy of them which we can perform our segmenting operations on by using the Arrays.toString() method on each and storing the character length of each in a variable. We then fetch the values of both JSpinners controlling the number of chunks the user wishes to split the text into; if the text is split into more, and smaller, chunks, then the similarity checking is likely to be more accurate since each chunk of the spun text is compared with each chunk of the corpus text, but also risks false positive results if the chunks are too small as the metric will pick up on similar passages that are perhaps only a sentence or two long, and which could have been quoted, for example.

Once the event handler calculates the size of each segment (the character length divided by the spinner value), we now iterate through a loop for each string based on the values of their respective spinners, which splits the strings into individual array elements (substrings) from the starting position to this position plus the segment size. The starting position is changed to this ending position on each subsequent loop so that we end up with the spun text split into (roughly) equally sized array elements. The next stage compares each segment of one text against every segment of the other; since the segments are not making the text any longer there should not be any noticeable drop in performance from this process. We start by creating an array to store all the similarity scores whose size is the product of the two JSpinner values. Using a nested for loop to allow for comparisons between all combinations of the two strings, the event handler calls the CompareStrings.score() method on the given combination of segments, using an iterator value to store each similarity score in a new array element.

The similarity checking aspect makes use of the Levenshtein Distance metric to compare the two supplied strings. It starts by finding the longer of the two strings, which is used to calculate how much of one string is different to the other using the formula $\frac{maxLength-editDistance}{maxLength}$. Both strings are converted to lower case so that identical letters are not treated as separate entities and create an array to store the costs involved in changing one string to another. The method then loops through the length of the second string and fills the costs array with the index values of this loop to set a baseline cost. We next use a nested loop whose outer loop works through each element in the first string and whose inner loop works through each element in the second. If the value of j is above 0 then we set currentValue as equal to the value of the previous array element. We then take the character at position i and j of strings one and two respectively, and if they are the same, we again edit currentValue. This is done by finding the smaller of the two values between currentValue and previousValue, and then finding the smallest between it and the value of costs[j]; we assign this number plus one as the new value of currentValue. After the if statement, costs[j-1] is set to the value of previousValue, which is set to the value of i at the start of each i loop. previousValue takes on the currentValue number. Once we are beyond the first loop of i, we store this previousValue in the final position of the costs array. A clearer example flow of operations is included below as pseudocode:

```
maxLength = max(first.length(), second.length()) = max(2,2) = 2
string 1 = "be", string 2 = "he"
int[] costs = new int[3]

(i = 0)
int previousValue = 0

(i = 0, j = 0)
costs[0] = 0

(i = 0, j = 1)
costs[1] = 1

(i = 0, j = 2)
costs[2] = 2

(i = 1, j = 0)
(i = 1, j = 1)
currentValue = costs[j - 1] = costs[0] = 0
costs[j - 1] = previousValue = 1
previousValue = currentValue = 0

(i = 1, j = 2)
currentValue = costs[j - 1] = costs[1] = 1
costs[j - 1] = previousValue = 0
previousValue = currentValue = 1

(i = 2, j = 0)
(i = 2, j = 1)
currentValue = costs[j - 1] = costs[0] = 1
costs[j - 1] = previousValue = 2
previousValue = currentValue = 1

(i = 2, j = 2)
currentValue = costs[j - 1] = costs[1] = 0
currentValue = min(min(currentValue, previousValue), costs[j])
currentValue = min(min(0, 1), 2) + 1 = min(0,2) + 1 = 0 + 1 = 1
costs[j - 1] = previousValue = 1
previousValue = currentValue = 1

costs[2] = previousValue = 1
numberToReturn = ((maxLength - computeEditDistance(first, second)) / maxLength
               = ((2 - 1) / 2)
               = 0.5
return numberToReturn
```

Figure 14. Pseudocode illustrating the general flow of the similarity checker process.

Once we have iterated through every combination of segments and obtained their similarity scores, we calculate the average similarity score across the texts by adding up each array element’s value and dividing it by the size of the array. Here we decide if the text is likely to have been plagiarised from this source text; the value I settled on was a similarity of greater than or equal to 35% - in this case we send an alert to the user to let them know, otherwise we can simply just print the similarity score and move on to the next paper. It is important to note that the similarity checker requires a lot more work and is not infallible, so part of the testing process was to find a balanced threshold that reduced false positives while still noticing obvious plagiarism. With more time to develop the program we could refine these elements further – a full evaluation follows.

3.2.3 Evaluation

The bulk of the evaluation of the de-spinner was conducted during the analysis phase (see Section 4), but it was also important to verify that the de-spinner was able to convert the full document to a root form, and that the de-spun text had a 100% similarity with the original text when also converted. These results are below:

File Name	Similarity to Original	Number of Words Spun	No. Words	% Words Converted
A	100%	444	4313	10.296
B	100%	1997	17277	11.560
C	100%	596	5668	10.513
D	100%	1176	11198	10.503
E	100%	229	1738	13.176
F	100%	544	3994	13.627
G	100%	754	5817	12.962
H	100%	561	4957	11.321
I	100%	1122	6298	17.815
J	100%	1448	11848	12.224
K	100%	1380	11441	12.063
L	100%	722	7066	10.216
M	100%	80	761	10.559
N	100%	1265	11489	11.010
O	100%	312	2471	12.632
P	100%	602	5823	10.336
Q	100%	532	4550	11.687
R	100%	1014	8244	12.296
S	100%	963	8142	11.833
T	100%	971	7541	12.880

Table 2. Assessment of De-spinner quality when comparing two of the same text converted to root form.

As we can see, in every case we are able to convert the source text into an identical root form in both cases, showing us that the method of reaching a "root" form by selecting the first synonym in the list for each word is always consistent. Secondly, we see that only a small percentage of the words are being converted to a root form; as with the spinner this is around 12% on average, and this is because the de-spinner operates using the same spinner method and so skips capitalised words, numbers, and stop words. With a dedicated method or more improvements to the current one we could ensure that every word is being converted and hopefully still reach 100% similarity between two instances of the same converted text; sadly, there was not enough time to properly implement this.

Some areas in which I felt improvements could be made primarily surround the de-spinner, as I believe that the similarity checker was implemented very well, although it would be useful to compare and contrast other similarity metrics such as the Jaccard coefficient and Cosine similarity score to understand how the Levenshtein distance compares to other metrics in this context. Regarding the de-spinner, one of my main concerns is with execution time; as the word count of each document increases, the execution time for the de-spinning process increases linearly (see Figure 18) and this indicates that times could become exceptionally large when dealing with much larger papers, such as

PhD-related ones. For this reason, it would be useful to investigate how the chunking feature of the de-spinner can be assessed in parallel using multiple CPU threads, potentially halving, or quartering the execution time depending on how powerful the CPU is. I would also have liked to make the move towards hosting the corpus online in a much larger database, so that the program itself becomes more portable and simply accesses the papers via a key; this in combination with the previous point regarding parallel execution would, I believe, help to hugely cut down on execution times involved with this aspect and hence save academic markers more time.

Overall, I believe that the de-spinner and similarity checking portion of the program have been highly successful. When we consider the difficulties surrounding the spinner in terms of the small proportion of text successfully spun, the results obtained from the de-spinner are very good. With around 90% of the text remaining un-spun in the spinning process, the fact that we are able to convert both the original corpus text and spun text to its root form using our system of picking the first synonym in the list for each word and obtain around a 94.4% similarity score on average between the two shows a good implementation of this process as well as indicates that the Levenshtein Distance metric is a suitable method of gauging similarity. There are certainly some improvements to be made; as the spinner becomes more fleshed out we need to account for this with the de-spinning process which could increase the execution time overhead, but I am confident that the similarity scoring method would handle these new behaviours well as it already compares each character of one string against each character of the other, so sentences which are in wildly different locations will still be picked up on by the metric.

3.3 Professional Computing Approach

To ensure that the project remained on schedule and that tasks were completed within a relevant timeframe, a personal diary was kept tracking the day-to-day progress. This helped me to stay on schedule and plan my work around my job and other responsibilities. Calendar and email applications such as Microsoft Outlook helped organise and plan regular meetings with my supervisor to discuss progress and potential next steps.

The BCS Code of Conduct is a guideline for all of those working in the computing sector, and there are many ways in which the project has aligned with these responsibilities and principles:

1. **Public Interest**
    a. Originally this project was going to use a corpus of papers from students at the university for testing and refining of the program, however this was decided against as publicly available papers surrounding the same topic are used instead, meaning no private personal data is processed as part of the project.

    c. The project does not factor in any attributes of a person's individualism when evaluating the program.

    d. This program was designed in a simple enough manner that it would be able to be used by people without high computer skills if it were to be scaled up and put to public use.

2. **Professional Competence and Integrity**
    a. Along with parts b and c of this section, the project is an opportunity to both learn new skills, such as how to perform natural language processing tasks, and the lengthy planning involved in such a task, while also developing existing problem solving and time management skills and building upon existing knowledge of Java and object-oriented programming.

e. Regular meetings with my supervisor allowed for back-and-forth discussions and feedback regarding what was good, what could be improved upon, and where to go next.

g. No bribery or financial incentive that would threaten the integrity of the project was involved in any way.

3. Duty to Relevant Authority

a. I ensured that to the best of my ability, this work is professional, appropriate, and within the interests of the relevant authority (the University of Stirling).

c. I accept personal responsibility for all the work undertaken during this project (except in cases where external APIs were needed, for example) and that any issues or mistakes are mine alone.

e. I do not pretend that this project was more successful than it was in reality and acknowledge the many ways in which it could be improved in future given further growth in myself and my abilities.

4. Duty to the Profession

a. No illicit, illegal, or questionable actions were performed during the completion of this project which risked causing harm to the reputation of the BCS and the wider computing society.

d. I ensure that I always treat my colleagues with the utmost respect.

e. Staying connected with others on the course, particularly through review of their elevator pitch videos has allowed me to push my colleagues to produce the highest-quality work they could.

4 Analysis and Conclusion

4.1 Analysis of Data

4.1.1 Spinner

The testing of and analysis of a project is arguably the most important aspect of the entire process. Both the spinner and de-spinner/similarity checker aspects were tested to allow us to analyse the general patterns seen with the data, in particular average execution time and similarity score.

File Name	Abbr.
A Flexible Agent-Based Model to Study COVID-19 Outbreak.pdf	A
agent-based covid-19 diffiusion.pdf	B
Agent-Based Framework for the North Carolina.pdf	C
agent-based modelling overview.pdf	D
agent-based modelling spatially inhomogenous.pdf	E
agent-based simulation deep learning.pdf	F
Analysis of COVID-19 in Tokyo.pdf	G
CITY-SCALE SIMULATION OF COVID-19 PANDEMIC.pdf	H
COVID-19 AGENT BASED MODEL WITH.pdf	I
Detecting Automatically Spun Content on the Web.pdf	J
effectiveness of main solutions to covid-19.pdf	K
Exploring the effectiveness.pdf	L
health and economic benefits.pdf	M
How to restart An agent-based simulation model.pdf	N
interrelation between the.pdf	O
Modelling COVID-19 transmission in.pdf	P
simiulating delay in seeking treatment.pdf	Q
universal masking is urgent.pdf	R
UTLDR an agent-based framework for modelinginfectious diseases.pdf	S
verification of an agent-based disease model.pdf	T

Table 3. Filenames of papers and their corresponding short name (for more concise looking tables).

In the case of the spinner, it was decided that the best way to evaluate the performance was to spin the full document on every word and then call the similarity checker on this and the original version to assess how similar the two were and hence how good the spinner is. It was also important to measure the level of readability of the spun text, so for this the site **datayze**[23] was used; it allows the user to submit a passage of text and get information about it, most importantly a score of how "readable" the text is. There were six readability scores to choose from, and I selected the Flesh Reading Ease score and Gunning Fog Scale Level. The Flesh score uses the total number of words, sentences, and syllables to compute a score from 0 – 100, with a lower score indicating a text that is harder to read. The formula for this score Is $206.835 - 1.015(\frac{total\ words}{total\ sentences}) - 84.6(\frac{total\ syllables}{total\ words})$ (formulas found on the site readable[24]). The Gunning Fog Scale Level places weight on complex words, which are those with three or more syllables, and gives a score between 0 and 20, with a higher score indicating a higher level of difficulty and so education required to understand well. It uses the formula $0.4(\left(\frac{total\ words}{total\ sentences}\right) + 100(\frac{complex\ words}{total\ words}))$. I felt that looking at more than one scoring metric would give me the greatest sense of success when looking at the readability aspect; the scores are shown for the both the original text and the spun text. Various other factors were tracked such as the number of words in the text, average similarity, percentage of words spun, and the average time for execution. Please see Appendix 5.1 for the full table containing data relating to the spinner analysis.

Firstly, we can look at the average execution times when spinning the documents. Table 2 shows us what appears to be a linear relationship between the growth in document word count and the average execution time, but it could be interpreted as exponential to some extent. For this reason, it would have been helpful if time had allowed to use a larger corpus for more data points in order to draw a more complete comparison, however we can most likely conclude that the relationship is linear given the results of the execution time tests below when dealing with the de-spinner. The longest execution time was seen with paper B(**6068ms**), and the shortest with M (**91ms**). As discussed below as well, this time could be further cut down perhaps by means of parallel execution of the spinner on individual sections and this could be explored in future, but I am happy with the execution time seen given that the average execution time across all papers was 1749.2ms, or around 1.75 seconds.

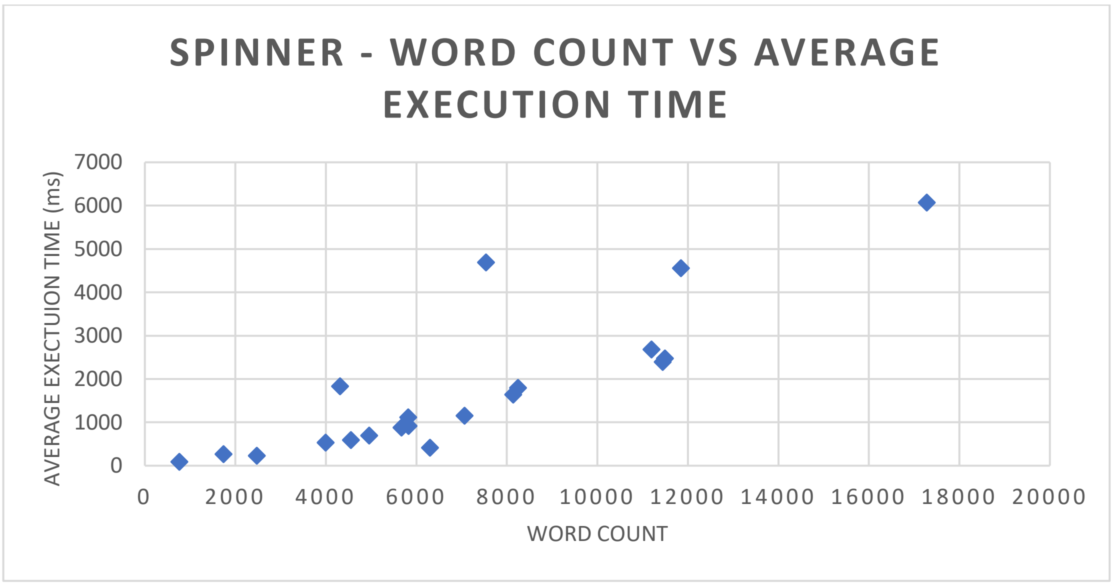
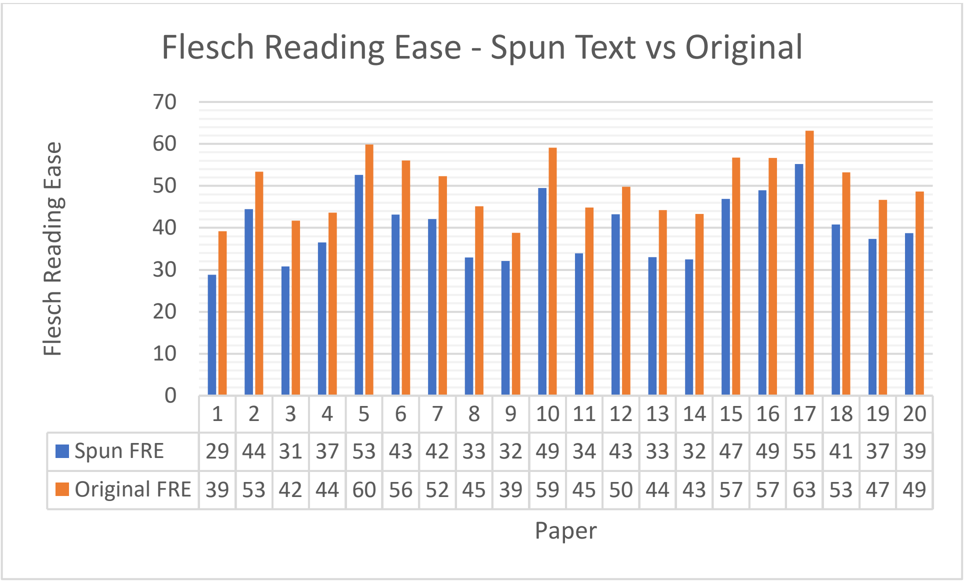


Figure 15. Scatter graph showing a comparison between the word count and average execution time of the spinning process.

It is important to highlight the high level of similarity across all the spun texts with their original source. The average similarity level was 88.019%, meaning that on average around 11.981% of the document was spun. This is an extremely low number and is best explained by the fact that our spinner tool is not as fully developed as some online tools are. Capitalised words, stop words of two or less characters and numeric 'words' are all ignored which gives us a rather small pool of words that are able to be edited. Given further development in terms of implementing other characteristics of online spinner tools such as the restructuring of sentences, careful changing of tenses and making use of punctuation to break up a sentence, I believe we would be able to achieve a much lower similarity score. This low degree of spinning is reflected in the results for the de-spinner and plagiarism checker below.

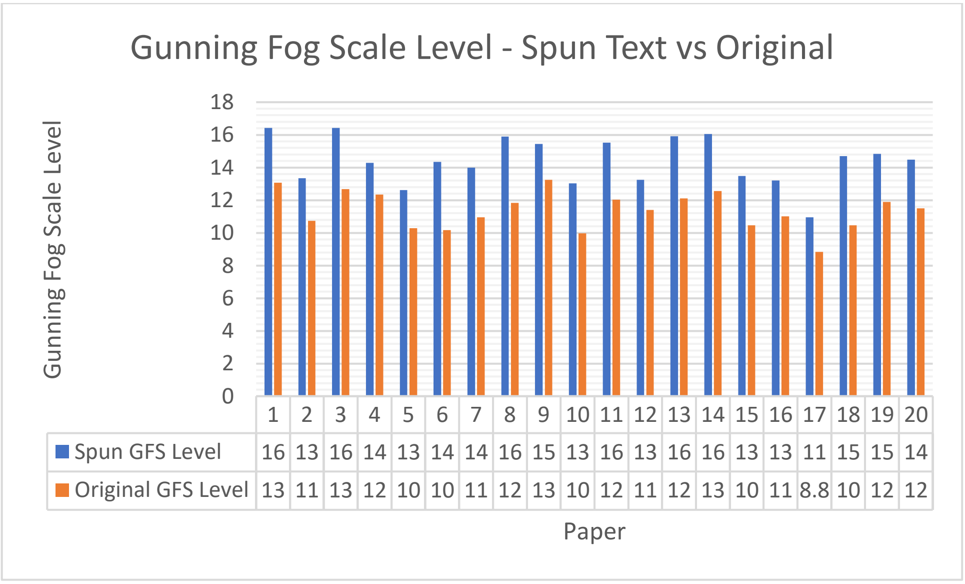
Finally, we can compare the Flesch Reading Ease and Gunning Fog Scale Level for both the original text and its spun equivalent. In the case of the former, Table 3 shows the comparison and values. Here we can see that in all cases, the FRE score is higher for the original than the spun version, showing a higher level of readability, that is, it is easier to understand. The average FRE score for the spun texts is 40.19, versus 49.82 for the original texts, only a difference of 9.63, indicating that the spinner does a relatively good job of maintaining a similar level of readability, although there is no doubt that this could be improved by ensuring that suffixes are readded in a more sensible manner; at the moment they can sometimes look unusual when added to a synonym that does not use the same rules as the original word. The largest difference in readability was 12.87 with paper F, with the smallest seen in paper L at just 6.58.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
🟦 Spun FRE	29	44	31	37	53	43	42	33	32	49	34	43	33	32	47	49	55	41	37	39
🟧 Original FRE	39	53	42	44	60	56	52	45	39	59	45	50	44	43	57	57	63	53	47	49

Figure 16. Side-by-side bar graph comparing the FRE score for spun texts vs the originals.

Looking at the Gunning Fog Scale Level (GFSL) values in Table 4, we can see the same pattern of the original text always being more readable than the spun text. In this case, all the original text scores are lower, which shows a higher readability. The average GFSL score for the original text was 11.38 compared to 14.44 for the spun text, giving us a difference of 3.06. The largest difference in score was on paper R, at 4.23, with the lowest once again on paper L at 1.84. The fact that both scoring systems show different patterns in the shape of their bar graphs, and the fact the greatest score difference was seen on two different papers, F and R respectively, highlights the importance of picking the right scoring system to evaluate a spinner tool. In both cases, the drop in readability was relatively low, certainly smaller than had been expected, giving us a good basis to say that as our spinner is at the moment, it is very good at producing relatively readable and understandable results, and ones which reflect the results outputted by online tools. As mentioned earlier, there is scope to reduce the execution time further as well as considerably develop the features of the spinner in order to cut down on the level of similarity between the spun text and the original even more, given more time.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
🟦 Spun GFS Level	16	13	16	14	13	14	14	16	15	13	16	13	16	16	13	13	11	15	15	14
🟧 Original GFS Level	13	11	13	12	10	10	11	12	13	10	12	11	12	13	10	11	8.8	10	12	12

Figure 17. Side-by-side bar graph comparing the GFSL score for spun texts vs the originals.

4.1.2 De-Spinner

In the case of the de-spinner, two main aspects were investigated; the execution time of the entire de-spinning and similarity checking process, and the similarity score itself. For this reason, each text was compared against itself and every other corpus text in the collection; that is, we take the spun text of corpus paper A and perform de-spinning and similarity checking between it and corpus papers A, B, C, and so on. In this way, we obtain a large table with an identical X and Y axis and can draw comparisons between each combination.

Firstly, looking at the similarity checks(Please see **Appendix 5.1**), we can clearly see that the greatest level of similarity lies between the spun text and its original parent text in all cases, shown clearly by the heatmap, with higher values approaching red and lower ones approaching green. The similarity scores sit above 92% in all cases, with an average of 94.4%, while the cells which do not line up between the X and Y axis show an average similarity score of just 26.3%. The highest non-diagonal score was 42.5%, between papers N and K, and similarly, K to N gave a score of 42.1%. This suggests that these two papers were unnaturally similar in terms of content, and looking at them we see that the contents of paper K share a great similarity with the contents of paper N, simply reworded in a different manner with more focus on different areas – they are published by the same writers so we cannot claim plagiarism in this case but is still a very strange case, and one of luck that I did not notice the two were similar when finding sample papers. Therefore, given that the average similarity score for unrelated documents is 26.3% while the reworded papers K and N give around 42.3% similarity, I propose a threshold value of around 35-40% for deciding if plagiarism has taken place, using the current setup of the de-spinner and plagiarism checker.

While 40% is a remarkably high threshold for deciding if plagiarism has taken place, we must factor in that the texts were first spun using our spinner tool, and in it no capitalised words were spun, nor were words shorter than three characters in length. Because of this, as highlighted earlier only around 11.98% of the documents are spun on average, so we would expect a very high similarity score for all our tests – hence for this reason I recommended the 35-40% threshold. To compare the performance of our program against online spinning tools instead, we take the text from each corpus paper and spin it using the online spinner tool free-article-spinner[25] before using our de-spinner on this text and getting a similarity score against its parent paper. This nets us an average similarity score of 24.6% when using text that has been spun online.

Name	Local Similarity Score	free-article-spinner.com Similarity Score
A	94.223	24.25
B	94.241	8.14
C	95.196	21.81
D	94.392	31.9
E	94.180	60.86
F	94.313	27.88
G	92.473	20.09
H	93.905	21.38
I	95.257	18.5
J	93.904	11.35
K	95.234	11.97
L	94.890	16.98
M	94.177	72.97
N	93.986	11.75
O	95.581	42.09
P	95.479	18.95

Q	94.464	24.26
R	93.788	14.5
S	93.921	14.51
T	94.764	17

Table 4. Comparison between locally spun similarity scores and online-spun ones.

We can see that there is a lot of variation in terms of the similarity scores, but little correlation between the local and online spinner similarities. The lowest score is 8.14% for paper B and 72.97% for paper M, a difference of 64.83%. It is unclear exactly why there is such a high variation in these scores, but I believe that it is a result of the level of development of this project's spinning tool when compared to the online tool. The project tool (the 'local' one) only converts words to synonyms with a slight change in sentence structure consequently, and even then, only works with non-capitalised words and non-stop words. On the other hand, the online tool does manage capitalised words yet does not reshuffle sentences or appear to perform any other typical spinning function. If we consider that most academic papers make extensive use of official terminology or scientific names which must be capitalised, then I believe that this explains the high variation in similarity scores for the online tool – some papers may contain a higher proportion of capitalised words than others, most likely ones which are longer, for example. We can also consider the below scatter plot of word count against similarity score for the online spun text, which further supports this theory by showing that similarity score increases exponentially as the word count decreases (See Appendix 5.2 for the full table).

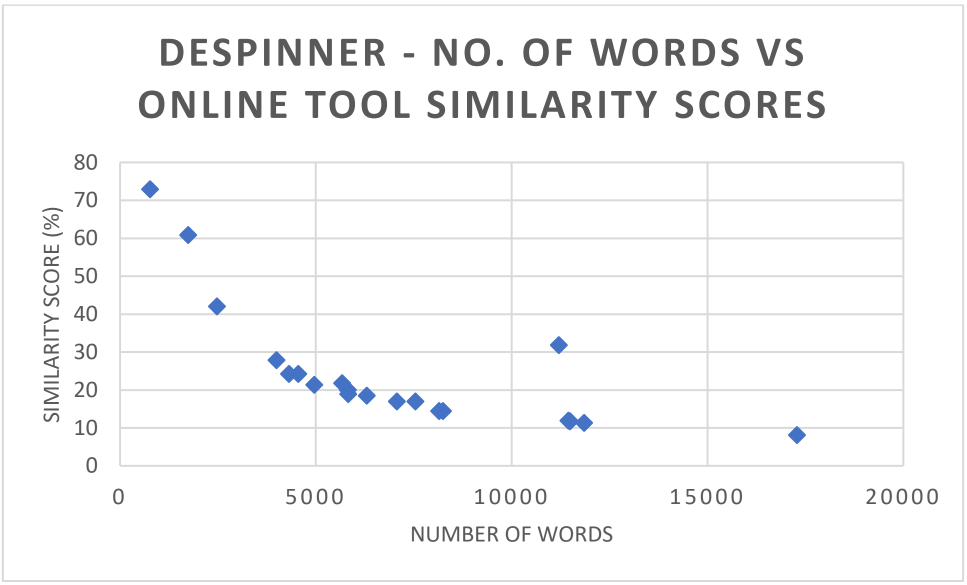


Figure 18. Scatter graph showing the number of words against the similarity score for the online-spun texts.

Hence, if our tool were to be further developed in a manner as to be able to deal with capitalised words without changing the names of people or abbreviations/acronyms, I believe we could see scores much more like what was seen with the online tool.

Looking at the execution time for the de-spinner and similarity checker part, we can take the average execution time for each paper when being compared both with, in a spun form, and against, in its normal form, another paper.

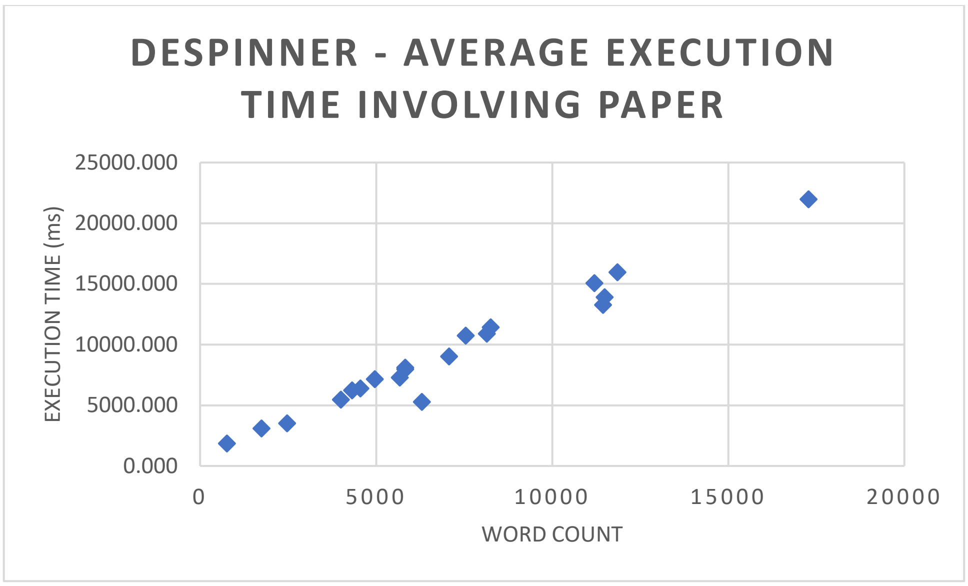


Figure 19. Scatter graph comparing the de-spinner/similarity checker execution time against word count per paper.

This graph clearly showcases that the average execution time for the de-spinning and similarity checking process increases linearly as the size of the files increase (Please see Appendix 5.3 for the full table of results). The longest execution time across the individual results, unsurprisingly, was between paper B and itself, at 51649 milliseconds, or around 51.6 seconds. Conversely, the shortest execution time was between paper M and itself at just 212 milliseconds or just over a fifth of a second. This is pleasing to see as it shows that there is no inherent problem with the way the de-spinning and analysis process is conducted; the only limiting factor to the execution time is the size of the file itself. There are likely a few ways to further reduce this time, possibly by conducting comparisons of individual chunks of the text in parallel using multiple cores, and merging the end results, and this would be worth investigating if this project were continued with in the future as saving time looking for plagiarised text was an important idea behind this project being conducted.

In conclusion, I believe that the project has been successful in the goal of creating a program which can imitate the results produced by online spinner tools and take spun text and convert it to a root form in which it can be compared to a set of original papers. There was a drop in readability associated with the spinning tool, but this is to be expected as I had found it difficult to guarantee that the swapped words would always make sense in the context of the sentence, and this unusual word choice can often be seen with online spinner tools as well, being one of the ways that academic markers can detect possible plagiarism or article spinning. The execution time for both the spinner and de-spinner appear to be linear which is pleasing as it shows there is no factor influencing it other than the word count of the text. The de-spinner, when considering the small percentage of the text that was spun to begin with, does a very good job at converting both them and the corpus papers to their root forms, and when comparing the spun version of a text to its original, only shows a drop of around 12% similarity at 88.02% on average. There are a number of improvements which could be made to the program in terms of the behaviours of the spinner as discussed above, such as shuffling of paragraphs or sentences around conjunctions or punctuation, but with this we would need to take it into account when developing the de-spinner and so we could expect longer execution times as more permutations are needed. Overall, I believe that this project has made a good start and has showcased the potential that a de-spinner tool can have in place of time-consuming manual plagiarism searching by examiners, given proper development and fleshing out of features, as well as cutting down of execution time, and I would be very excited to further develop this system as a side project.

4.2 Evaluation of Project

4.2.1 Areas of Strength

It is important to consider the ways in which the project was successful as part of a final analysis, and there are a few areas of note:

- I was pleased with the level of similarity between the spinner outputs retrieved from my program when compared to those of online spinners; while they do not read particularly well, the word choice is remarkably like the online version. While it would be nice to develop the spinning portion further, it is good that it worked to the standard of a typical free online spinner tool, and I believe that this helped with recording results that were more representative of these tools.
- In addition, I am happy that I managed to get the project completed on time and to a standard with which I was happy. It was a great learning experience developing skills surrounding natural language processing and continuing to strengthen my existing problem solving and essay-writing skills, and so I am pleased that despite the tight time constraints faced at times, I was still able to complete my research.
- I am satisfied with the quality of the results obtained, and how in-depth I was able to analyse them. Being able to assess the readability of the spun text as well as the similarity scores for each paper against the others when spun gave me a much more detailed overview of the quality of the system created and, along with the detailed graphs created, allowed for conclusions to be drawn on the most important aspects of the program such as similarity checker quality and overall average execution time. The initial execution time of a full paper spin was over one minute, but this was cut down to around 1.75 seconds on average after implementing smarter use of the dictionaries and no longer making calls to an online API every time a word was requested.

4.2.2 Areas to Improve Upon

Conversely, there are areas in which I feel further developments and improvements could have been made to strengthen the quality of the work:

- There is scope for improvements to be made to the comparison component of the de-spinner, where each chunk of the suspicious text is checked against each chunk of the corpus texts in turn (the number of chunks to use is determined by the user). This could have been improved by employing the techniques discussed by *Foltynek et al*[21] of seeding, extension, and filtering. The program follows the seeding procedure by matching text between both documents, but it does not extrapolate outward from that point (extension) to try and find the full paragraph or section that has been pulled from (and discover any other plagiarism), nor does it attempt to tidy up each fragment to reduce overlapping and potentially artificially inflated similarity scores. If these measures were implemented, I am sure that we would be able to more accurately refine the similarity score and produce more reliable results.
- Additional similarity scoring systems could have been analysed, rather than just the Levenshtein Distance metric, to research the most appropriate for plagiarism detection. This could have led to an extra layer of testing, where each spun and de-spun piece of text could be evaluated for similarity with the original source using every metric, to get an average and a sense of which metric delivered the most valuable results. Without this, we had no way of knowing if the Levenshtein Distance metric was truly appropriate or not, although the results gathered seem to give us a good enough level of accuracy regarding scoring.

- I feel that improvements could have been made to the spinner tool in order to incorporate some element of reshuffling of sentences and paragraphs to try and evade plagiarism detection – I was unable to find a reliable way of swapping sentences or parts of them around in a way which continued to have the paragraph be understandable, but I am confident that with more time we would have been able to spend more time on this, perhaps using the aid of OpenNLP to identify conjunctions as points where two halves of a sentence could be slightly reworded and swapped.
- As discussed earlier, it could have been useful to investigate if, and how, the program could be run in parallel. As one of the primary focuses of this project was on saving academic examiners time in marking submitted work, running the program in parallel, perhaps by comparing the suspicious text against multiple corpus texts at once using a multi-core process, could have saved a large amount of execution time, and hence gone a long way towards a more worthwhile system to use.

4.2.3 Future Work

Given the opportunity to expand upon and continue this research, there are some areas that I would have liked to investigate:

- I would have liked to experiment with the addition of a web-search system for locating matching research papers, rather than having to rely on the small corpus held in the local system. This could potentially be done by taking samples of text which do not line up with the normal style of the piece of text and performing a web search for these specific phrases. The ability for the program to do this would greatly reduce the overhead involved for academic examiners by not requiring that they download a repository of papers to their local system which can never possibly be exhaustive.
- Tying in with the above point, it would be worth investigating other methods of detecting plagiarism beyond simply performing synonym-matching. As highlighted in the literature review (section 2.3), other features such as sensibility and perplexity can be used, perhaps in combination with synonym-matching, to identify suspect sections of the text which are unnatural; since our program does not focus so much on the flow of a sentence or paragraph, there could be a lot of plagiarism that the tool misses simply because of sufficient word change, when in reality it might be structured in roughly the same way (bullet points, commas, full stops, run-on sentences and so on).
- Similarly, given more time I would investigate the use of non-textual feature analysis in plagiarism detection, which looks at features such as images, graphs, captions, and headings to identify suspicious parts of text. As discussed in the literature review, non-textual feature analysis allows for detection of syntax, semantic and idea-preserving plagiarism which cannot be noticed by the approach taken in this program. A combination of both the synonym analysis along with sensibility and perplexity checking, and non-textual feature analysis could widely increase the chances of detecting more subtle plagiarism; perhaps a chart is claimed to be original work, but upon investigation this leads us to an article with the same chart, and searching laterally through the rest of the article could spot some text which shares similarity with the content of the submitted paper.
- It would have been useful to incorporate a series of toggleable options or “flags” into the GUI to control numerous factors of the program, such as whether to spin certain parts of speech such as nouns, remove stop words or not, or to investigate other scoring systems such as the Jaccard coefficient. This would add a high degree of flexibility to the system and could potentially tie into the possibility of introducing cross-language plagiarism checking by

allowing for translation between two selected languages, or searching for matching documents in that target language, broadening the scope of ways to search for plagiarism.

References

[1] ‘Edit Distance | DP-5 - GeeksforGeeks’. https://www.geeksforgeeks.org/edit-distance-dp-5/ (accessed Sep. 08, 2022).

[2] ‘Plagiarism: Facts & Stats - Plagiarism.org’. https://www.plagiarism.org/article/plagiarism-facts-and-stats (accessed Aug. 22, 2022).

[3] ‘Internet - Our World in Data’. https://ourworldindata.org/internet (accessed Aug. 23, 2022).

[4] ‘What is Natural Language Processing? | IBM’. https://www.ibm.com/cloud/learn/natural-language-processing (accessed Aug. 27, 2022).

[5] T. Manders and E. Klaassen, ‘Unpacking the Smart Mobility Concept in the Dutch Context Based on a Text Mining Approach’, *Sustainability (Switzerland)*, vol. 11, no. 23, Dec. 2019, doi: 10.3390/su11236583.

[6] ‘A Gentle Introduction to Natural Language Processing | by Ronak Vijay | Towards Data Science’. https://towardsdatascience.com/a-gentle-introduction-to-natural-language-processing-e716ed3c0863 (accessed Aug. 29, 2022).

[7] ‘NLTK :: Natural Language Toolkit’. https://www.nltk.org/ (accessed Aug. 28, 2022).

[8] ‘Apache PDFBox | A Java PDF Library’. https://pdfbox.apache.org/ (accessed Aug. 29, 2022).

[9] ‘Apache OpenNLP’. https://opennlp.apache.org/ (accessed Aug. 29, 2022).

[10] ‘WordNet’. https://wordnet.princeton.edu/ (accessed Aug. 29, 2022).

[11] ‘WordNet Assignment’. https://www.cs.princeton.edu/courses/archive/spring20/cos226/assignments/wordnet/specification.php (accessed Aug. 29, 2022).

[12] ‘Java WordNet Interface’. https://projects.csail.mit.edu/jwi/api/ (accessed Aug. 29, 2022).

[13] Q. Zhang, D. Y. Wang, and G. M. Voelker, ‘DSpin: Detecting Automatically Spun Content on the Web’, 2014. Accessed: Jun. 08, 2022. [Online]. Available: http://evasivemosaic6837.wordpress.com/2012/

[14] U. Shahid, Z. Shafiq, S. Farooqi, P. Srinivasan, R. Ahmad, and F. Zaffar, ‘Accurate Detection of Automatically Spun Content via Stylometric Analysis’.

[15] ‘Understanding Cosine Similarity And Its Application | by Richmond Alake | Towards Data Science’. https://towardsdatascience.com/understanding-cosine-similarity-and-its-application-fd42f585296a (accessed Aug. 31, 2022).

[16] Y. Shkodkina and D. Pakauskas, ‘Comparative Analysis of Plagiarism Detection Systems’, *Business Ethics and Leadership*, vol. 1, no. 3, pp. 27–35, 2017, doi: 10.21272/bel.1(3).27-35.2017.

[17] J. Ferrero, L. Besacier, D. Schwab, and F. Agnès, ‘Deep Investigation of Cross-Language Plagiarism Detection Methods’, Jul. 2017, pp. 6–15. doi: 10.18653/v1/w17-2502.

[18] J. Ferrero, F. Agnes, L. Besacier, and D. Schwab, ‘UsingWord embedding for cross-language plagiarism detection’, in *15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017 - Proceedings of Conference*, 2017, vol. 2, pp. 415–421. doi: 10.18653/v1/e17-2066.

[19] Prasanth. S, Rajshree. R, and S. Balaji. B, ‘A Survey on Plagiarism Detection’, *Int J Comput Appl*, vol. 86, no. 19, pp. 21–23, Jan. 2014, doi: 10.5120/15104-3428.

[20] J. A. W. Faidhi and S. K. Robinson, ‘An empirical approach for detecting program similarity and plagiarism within a university programming environment’, *Comput Educ*, vol. 11, no. 1, pp. 11–19, Jan. 1987, doi: 10.1016/0360-1315(87)90042-X.

[21] T. Foltýnek, N. Meuschke, and B. Gipp, ‘Academic plagiarism detection: A systematic literature review’, *ACM Computing Surveys*, vol. 52, no. 6. Association for Computing Machinery, Oct. 01, 2019. doi: 10.1145/3345317.

[22] ‘Suffix Spelling Rules: 6 Keys for Adding Suffixes Correctly’. https://grammar.yourdictionary.com/grammar/spelling-and-word-lists/suffix-spelling-rules.html (accessed Sep. 07, 2022).

[23] ‘Readability Analyzer’. https://datayze.com/readability-analyzer (accessed Sep. 02, 2022).

[24] ‘About Readability – Readable, the home of readability’. https://readable.com/readability/ (accessed Sep. 02, 2022).

[25] ‘Free Article Spinner - Free Unlimited Web Content’. https://free-article-spinner.com/ (accessed Sep. 03, 2022).

5 Appendices

5.1 Spinner Results

Name	No. Words	Average Time	Average Similarity	% Words Spun	Spun FRE	Original FRE	Spun GFS Level	Original GFS Level
A	4313	1829	89.704	10.296	28.85	39.19	16.43	13.07
B	17277	6068	88.440	11.560	44.47	53.41	13.34	10.74
C	5668	875	89.487	10.513	30.81	41.72	16.42	12.68
D	11198	2680	89.586	10.414	36.54	43.61	14.29	12.35
E	1738	262	86.824	13.176	52.61	59.9	12.61	10.28
F	3994	531	86.373	13.627	43.18	56.05	14.35	10.17
G	5817	1114	87.004	12.996	42.09	52.29	13.99	10.95
H	4957	696	88.679	11.321	32.95	45.14	15.9	11.84
I	6298	412	82.185	17.815	32.07	38.78	15.45	13.25
J	11848	4555	87.776	12.224	49.47	59.09	13.03	9.97
K	11441	2392	87.937	12.063	33.95	44.83	15.53	12.04
L	7066	1151	89.784	10.216	43.24	49.82	13.25	11.41
M	761	91	89.309	10.691	33.05	44.26	15.92	12.1
N	11489	2473	88.990	11.010	32.45	43.29	16.06	12.57
O	2471	225	87.368	12.632	46.88	56.73	13.48	10.47
P	5823	915	89.664	10.336	48.95	56.66	13.2	11.01
Q	4550	588	88.313	11.687	55.23	63.13	10.95	8.83
R	8244	1793	87.655	12.345	40.82	53.2	14.69	10.46
S	8142	1642	88.167	11.833	37.4	46.68	14.83	11.9
T	7541	4692	87.133	12.867	38.72	48.63	14.49	11.51

5.2 Similarity Scores Between the De-Spun Form of the Spun Text Against the Original Forms of Each Paper

Title	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
A	94.223	18.782	32.203	24.239	27.114	32.264	33.449	33.140	32.202	24.290	26.381	32.160	13.804	25.406	30.094	32.726	31.060	28.742	29.337	31.900
B	19.128	94.241	21.955	32.611	9.204	17.465	23.890	21.889	16.820	33.606	32.049	25.912	4.078	32.489	11.346	23.909	20.147	29.873	29.464	26.093
C	32.484	21.720	95.196	27.000	22.699	32.468	30.689	28.782	31.717	27.182	29.066	32.095	11.286	28.062	26.094	30.692	31.600	30.819	31.184	32.011
D	24.585	32.217	27.243	94.392	12.592	22.663	29.536	27.862	22.094	28.991	31.911	31.534	5.689	30.383	15.123	29.651	25.803	32.858	33.270	31.619
E	26.680	8.949	22.379	12.319	94.180	27.866	21.258	23.043	28.567	12.223	13.727	18.926	25.616	13.009	33.728	20.592	24.626	15.438	16.052	18.889
F	32.110	17.126	32.256	22.363	28.410	94.313	32.683	33.151	29.097	22.433	24.590	30.682	14.927	23.530	31.051	31.903	33.070	26.933	27.571	30.697
G	33.811	23.108	30.738	28.855	21.998	33.117	92.473	31.624	32.114	29.211	30.992	32.637	10.725	30.058	25.236	30.678	33.368	32.639	32.873	32.866
H	33.332	21.201	28.663	27.112	23.695	33.500	31.867	93.905	32.823	26.985	29.027	32.787	11.759	27.921	26.780	31.844	31.543	31.243	31.383	32.622
I	32.284	16.345	31.413	21.617	29.332	29.196	31.493	32.418	95.257	21.284	23.398	29.818	15.536	22.458	31.774	30.939	32.481	25.761	26.322	29.299
J	24.476	33.152	27.204	29.177	12.415	22.600	29.746	27.501	21.643	93.904	32.818	31.321	5.612	31.433	15.057	29.544	25.511	33.605	33.606	31.496
K	26.748	31.566	29.268	31.677	14.017	24.882	31.688	29.676	23.874	32.618	95.234	32.810	6.475	42.504	16.827	31.452	27.751	32.627	34.314	32.990
L	32.475	25.382	32.333	31.095	19.278	31.057	32.691	33.021	30.291	31.028	32.533	94.890	9.261	31.680	22.537	31.728	33.358	32.819	32.607	28.795
M	13.572	3.984	11.130	5.598	24.652	14.563	10.358	11.383	15.044	5.563	6.368	9.106	94.177	6.030	22.078	9.992	12.453	7.314	7.596	9.067
N	25.986	31.779	28.548	30.482	13.457	24.084	30.930	28.887	23.203	31.550	42.145	32.240	6.194	93.986	16.196	30.782	26.952	32.968	33.441	32.473
O	29.908	11.210	25.990	15.029	33.858	30.783	24.763	26.279	31.416	15.058	16.719	22.395	22.568	15.916	95.581	23.983	28.069	18.638	19.147	22.127
P	33.140	23.461	30.983	29.288	21.006	32.137	30.463	31.987	31.371	29.300	31.124	31.618	10.201	30.171	24.174	95.479	33.502	32.544	32.409	31.135
Q	31.455	19.759	31.230	25.470	25.165	33.399	33.280	31.673	32.765	25.315	27.416	33.174	12.710	26.348	28.360	33.326	94.464	29.780	30.125	32.612
R	29.196	29.130	31.136	32.454	15.854	27.332	33.177	31.845	26.367	33.293	32.383	33.110	7.464	32.725	18.868	32.844	30.091	93.788	30.578	33.250
S	29.963	28.532	31.590	32.910	16.652	28.164	33.492	32.275	27.193	33.269	33.956	32.987	7.848	33.236	19.557	32.813	30.793	31.084	93.921	32.846
T	32.474	25.378	32.240	31.041	19.509	31.342	32.963	33.023	29.972	31.071	32.555	29.094	9.358	31.824	22.479	31.056	32.925	33.011	32.810	94.764

5.3 Online-Spun Text Similarity Scores with Original by Word Count

Name	No. Words	Local Similarity Score	free-article-spinner.com Similarity Score
A	4313	94.223	24.25
B	17277	94.241	8.14
C	5668	95.196	21.81
D	11198	94.392	31.9
E	1738	94.180	60.86
F	3994	94.313	27.88
G	5817	92.473	20.09
H	4957	93.905	21.38
I	6298	95.257	18.5
J	11848	93.904	11.35
K	11441	95.234	11.97
L	7066	94.890	16.98
M	761	94.177	72.97
N	11489	93.986	11.75
O	2471	95.581	42.09
P	5823	95.479	18.95
Q	4550	94.464	24.26
R	8244	93.788	14.5
S	8142	93.921	14.51
T	7541	94.764	17

5.4 Average De-Spinning Execution Time involving each Paper by Word Count

Short Name	No. Words	Average Execution Time Involving Paper
A	4313	6231.717949
B	17277	21970.53846
C	5668	7290.435897
D	11198	15072.41026
E	1738	3090.051282
F	3994	5465.153846
G	5817	7962.076923
H	4957	7147.487179
I	6298	5269.102564
J	11848	15968.64103
K	11441	13256.94872
L	7066	9025.897436
M	761	1856.487179
N	11489	13900.5641
O	2471	3514.717949
P	5823	8094.282051
Q	4550	6377.846154
R	8244	11428.41026
S	8142	10890.97436
T	7541	10725.33333