Plagiarism Detection

Noah Carnahan Marcus Huderle Nick Jones Cole Stephan Tony Tran
Zach Wood-Doughty

March 17, 2014

1 Introduction

Plagiarism is certainly a problem in today's world, and it probably has been ever since writing was invented. Developing an effective, automated tool for detecting plagiarism is both practically useful and theoretically interesting. Our project aimed explore and implement various methods of plagiarism detection. This field is relatively young, so many of the papers we read were published in the past fifteen years.

There are two overarching plagiarism detection schemes that we studied and implemented. The first is called *intrinsic detection*, and the second is called *extrinsic detection*. While the ultimate goal of each is to detect potential plagiarism, they are completely different approaches to the problem and address different types of plagiarism.

Intrinsic detection does not rely on any external documents to detect plagiarism. Rather, it tries to detect shifts in writing style within a suspicious document. For example, if a student paid his or her older brother to write the conclusion of a paper, intrinsic detection should be able to detect that the conclusion was not written by the same person as the rest of the paper.

Extrinsic detection is the more traditional model of plagiarism detection. Given a suspicious document and a body of source documents, extrinsic detection identifies pieces of the suspicious document that originate from the source document. This may seem like a simple string matching problem, however, we'll need to apply some more sophisticated methods to obtain any meaningful results.

1.1 Corpus Description

For this project, we made use of the PAN-plagiarism-corpus-2009 (see [1]) to test the effectiveness of our detection algorithms. The corpus is constructed of two sets of documents: suspicious documents and source documents. Suspicious documents are documents where plagiarism has been artificially inserted for others to try to detect, and source documents are the sources for said plagiarism. In each case of plagiarism, text has been taken from a source document, modified, and inserted into a suspicious document. The degree of difficulty to which the plagiarism is hidden is dependent on the modification the source text went through. In this corpus, there are three types of modifications: paraphrasing, part-of-speech reordering, and random text operations.

Paraphrasing: When a source text is paraphrased, a random number of words within the text are replaced with random synonyms or antonyms.

Source Text: "Mr. Dursley always sat with his back to the window in his office on the ninth floor."

Modified Text: "Mr. Dursley always propped his back on the glass window on the ninth floor of the office"

Parts-of-Speech Reordering: When a source text undergoes this type of modification, matching parts-of-speech tokens are randomly swapped within the sentence.

Source Text: "Mr. Dursley always sat with his back to the window in his office on the ninth floor."

Modified Text: "window always sat with his floor to the Mr. Dursley in his office on the ninth back."

Random Text Operations: This type of modification takes a source text and performs random word replacements and random re-ordering to create the artificial plagiarism.

Source Text: "Mr. Dursley always sat with his back to the window in his office on the ninth floor."

Modified Text: "sat with glass his work place ninth level Mr. always back on the Dursley to the."

Another instance of artificial plagiarism that can be found in our suspicious documents is a simple insertion where text is inserted into a suspicious document with no modifications.

1.2 ROC Curves

To measure the quality of our tool, we used the area under the curve of the receiver operating characteristic (ROC) curve. For each passage, our tool doesn't output a binary classification of plagiarism or not plagiarism, but instead outputs a confidence that each passage is plagiarized. These confidences range between 0 and 1. A threshold t can then be chosen. We then classify each passage with a confidence greater than t as plagiarism and all others as not plagiarism. The ROC curve plots the false positive rate and the true positive rate at various thresholds. To generate the curve, we vary t, adding points to the graph as we go, so that first all n passages have a greater confidence, then all $n-1$ passages, then all $n-2$, and so on.

Ideally, all plagiarized passages would have greater confidences than all non-plagiarized passages. In this case, the curve runs straight up the y-axis to the point (0,1) then straight across to the point (1,1). If the tool performs as well as random, then we would expect the false positive rate and the true positive rate to be equal at any threshold. Therefore, the curve would run along the diagonal in this situation.

We can compare two ROC curves to each other by looking at the area under the curve (AUC). In the ideal situation, the AUC is 1. In the situation where the tool performs as well as random, the AUC is 0.5. Thus, greater AUC corresponds to better performance. We want to get this number as close to 1 as possible.

1.3 Precision & Recall

Because our corpus came from the PAN-Plagiarism 2009 Conference, we also considered the measures used by this conference to measure the effectiveness of our tool. The conference defines precision and recall specifically for the plagiarism detection task. The base "unit of retrieval" is the character – in other words, a classifier is evaluated based on its precision and recall for each individual character. It should be noted, that these definitions expect binary inputs: we must classify each character as either "plagiarized" or "not plagiarized."

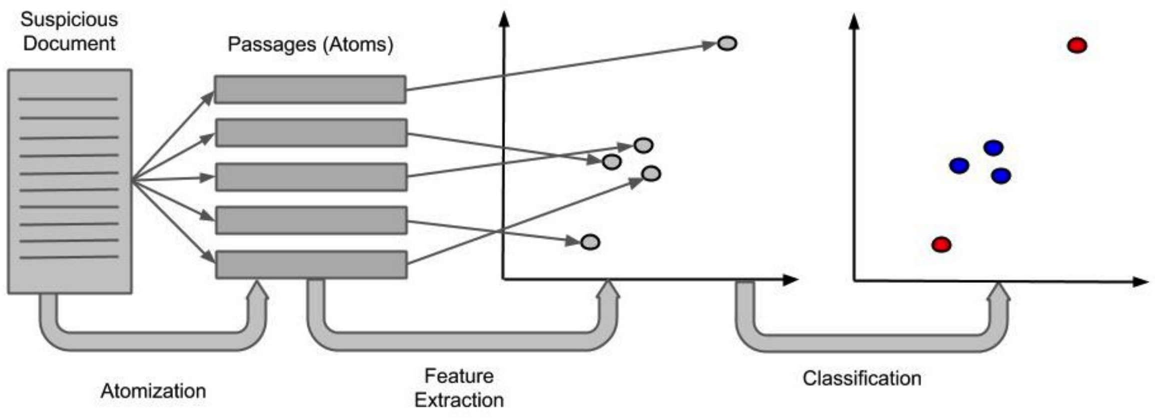
While we were intrigued by the idea of comparing our results to those of previous research, we found the above metrics to be limiting. In order to calculate these metrics, we must introduce another parameter to our system: a threshold value. When using AUC as our metric, we are able to capture the effectiveness of our system across a wide variety of thresholds. For that reason, we chose AUC as our primary metric of success. To see the specific details of these metrics, please see [1].

2 Intrinsic Detection

Recall that the goal of intrinsic plagiarism detection is to find passages within a document which appear to be significantly different from the rest of the document. In order to do so, we break the process down into three steps.

1. Atomization – Deconstruct a document into passages.
2. Feature Extraction – Quantify the style of each passage by extracting stylometric features based on linguistic properties of the text. Each passage is represented numerically as a vector of feature values.

3. Classification – Compare the feature vectors of passages to one another; those passages that are significantly different will have higher confidences of plagiarism. Return a confidence that a passage was plagiarized.



2.1 Atomization

The first step in analyzing a full document is to break the document down into smaller chunks, or passages. Each passage can then be analyzed independently and compared to one another. We use the terms *passage* and *atom* interchangeably, and refer to the smallest unit of text as an *atom type*. Possible atom types, from smallest to largest, include word, sentence, paragraph, and n-character.

We would like to use a small atom type in order to be as exact as possible with our plagiarism detection. In other words, it's preferable to be able to classify individual sentences as being plagiarized rather than classifying a two-page-long passage as plagiarism.

On the other hand, a small atom type leaves our detection system with less data to work with. Stylometric features (discussed below) are meant to quantify an author's writing style – with more text to work with, these features are more likely to capture the author's style. If a feature is extracted from a small atom type (such a sentence), there is less text to work with.

We immediately found that small atom types simply didn't provide enough to information to quantify a writer's style effectively. As a result, the two atom types used were paragraph and n-character, where $n = 5000$ was taken from Stein [2].

2.2 Stylometric Feature Extraction

Once we have split our text into passages via atomization, we need a way to analyze those passages quantitatively. The goal of feature extraction is to develop "features" that tell us something about the writing style of a particular passage. If a particular passage differs significantly in these features from the rest of the passages in the document, we might be suspicious that the passage was plagiarized.

A feature is analogous to a mathematical function: it takes in an input (a passage of text) and produces an output (a value that somehow quantifies of that text). We tried to use features that would assign meaningful numerical values to the text they were applied to. In particular, many of our features tried to measure the complexity of writing style in some way or another.

Two examples of features we implemented are *average sentence length* and *syntactic complexity*. *Average sentence length* is a fairly intuitive feature that simply returns the average number of words per sentence within the passage. While it is obvious that authors do not always maintain the same average sentence length throughout their work, we might hope that authors would be in general fairly consistent. This feature would at least be likely to distinguish between the writing of a PhD student and that of a fifth grader. The second example feature, *syntactic complexity*, is a more complicated quantification of writing style. It takes a weighted sum of various parts of speech—subordinating conjunctions, wh-pronouns, and verb forms—as a one-off measure of complexity at the syntax level. We took this formula (and slightly modified it) from a linguistics paper we found [3]. This feature proved to be fairly good at distinguishing between distinct authors.

We implemented 40 base features. To further extend our capabilities to analyze text, we implemented a framework that would allow us to take averages and standard deviations across feature values within the text. For example, a feature like *syntactic complexity*, which operated at the sentence level, could be averaged over each sentence in a paragraph, giving a paragraph feature, *average syntactic complexity*. This framework extended our total number of features from 40 to 74. All 74 features are listed in the features appendix.

The process of feature extraction takes in the passages created by atomization, and runs each of those passages through some subset (say, of size n) of our features. Each feature gives a value to each passage, and so each passage gets a "feature vector" of the n values it has received from the features. Once we have a feature vector for each passage, we could consider plotting them in n-dimensional space, and trying to analyze them graphically. Intuitively, we would expect the true author's passages to be close together in this graph, and any plagiarized passages would be far away. To do this analysis, we will turn to the task of classification, which tries to decide how suspicious a given passage is, based on its feature vector.

2.3 Classification

k-means Clustering

The first approach we took to classification was k-means clustering. The basic idea was to cluster the feature vectors from our passages and base our confidence of plagiarism on these clusters. Ideally, we could use $k = 2$, meaning one cluster would be the "plagiarized" cluster of passages and the other would be the "non-plagiarized" cluster. We made the assumption that the larger of the two clusters corresponded to the true author's passages, and the smaller was made up of plagiarized passages. Then, the confidence of a given passage being plagiarized was a scaled distance from the "non-plagiarized" centroid.

$$confidence = \frac{\text{distance\_from\_notplag}}{\text{distance\_from\_notplag} + \text{distance\_from\_plag}}$$

This notion of confidence can be extended to any $k > 2$. In preliminary testing, we found that using $k > 2$ did not improve our results and therefore stuck to $k = 2$ whenever using k-means.

Outlier Detection

Our outlier detection techniques were based on those discussed in [2]. Since outlier detection ended up being our most successful classification technique (see results), we'll describe it in more detail.

For a given document D, consider the distribution of feature f over all p passages in D. Ignoring the e most extreme values of f ($e = 1$, but we tried varying e), take the mean and standard deviation of the observed values of f. We assume that the true author writes within feature f drawing from a normal distribution derived from the observed mean and observed SD. Since the assumption that features are normally distributed is a stretch to begin with, we also tried centering the distribution at its median. This appeared to perform markedly worse.

For each passage, calculate the probability of feature f coming from the PDF of the normal distribution calculated above (this is considered to be the probability of not being plagiarized). Similarly, assume the plagiarized distribution to be uniform with minimum equal to $min(f)$ and maximum equal to $max(f)$. Then the probability of being plagiarized comes from the PDF of the uniform distribution described previously.

To combine these "probabilities", we simply use

$$\frac{P[\text{plagiarized}]}{P[\text{plagiarized}] + P[\text{not plagiarized}]}$$

to output a confidence of a given passage being plagiarized. The next step is to use multiple features and combine their confidences. Following the standard use in the literature [2], we follow a Naive Bayes assumption that features are distributed independent of one another. Thus, the confidence that a given passage is plagiarized comes from multiplying those confidences output by each individual feature.

Hidden Markov Modeling

Another form of classification that we used was Hidden Markov Modeling. In this system, we consider the atomized representation of our suspicious document to be the output of a Markovian State Machine. We consider a two-state machine whose states produce either plagiarized or non-plagiarized output. Using Viterbi's algorithm we generatively calculate the most probable state sequence of the machine, the idea being that when the machine transitions into the plagiarized state we believe the corresponding passage to be plagiarism.

This approach intuitively fits the problem set-up in that it takes into account the adjacency of the passages within the document. Normally, instances of plagiarism are continuous and once a transition into plagiarism is encountered we found that it is more probable for the next passage to be plagiarized than if the previous passage was non-plagiarized.

Determining confidence in these assignments is still done using distance from cluster centroid and can be expanded by including a notion of confidence based on the severity of the transition into the current plagiarized section. In this way, we account for the local degree to which the writing style changed in our confidence assignments by saying that a larger transition signifies a more obvious writing style switch. Similarly the extent of the transition out of a plagiarized section can be taken into account by measuring the degree to which the author's writing style has returned to normal i.e., compares to the last known non-plagiarized passage.

The machine was trained on a small set of 50 documents. The initial transition probabilities were taken from the actual transition probabilities however, the training set we used only partially accounted for the wide variety of documents found in our final evaluation set and some of the documents contained no plagiarism. The emission probabilities were left untrained as to account for each document's individual nature as the feature representations of the primary author vary by document.

2.4 Feature Selection

The classification schemes described above can be used with any arbitrary number of features simply by adding another dimension to each feature vector. Since we developed 74 features, a natural next step would be to find specific sets of features which work well together. Hypothetically, multiple features which perform well individually would perform even better when combined. With 2^{74} possible feature sets, we needed a simple way to test "logical" combinations of features.

One approach we took was a greedy algorithm for combining good features. The basic idea is to find the best individual feature; try combining every other feature, and select the best pair of features based on the above combinations. Once the best pair has been selected, try all sets of three that include the two already chosen features. The algorithm proceeds in this manner until using all features.

More formally:

Algorithm 1: Greedy Feature Selection

```
$remaining\ features$ := all features
**for** $n = 1$ ... *total number of features* **do**
    **for** $f \in$ *remaining features* **do**
        Try best seen (of size $n - 1$) + $f$
    **end**
    Record best combination of size $n$ (from above for-loop)
**end**
```

We found that this approach didn't yield promising results. Adding additional features did not significantly improve the feature set's performance. Summary results will be given below. It should be noted that none of the smaller subsets of features performed better than using all features at once.

While finding a good set of features to work with is one possibility, we could generalize this approach to place weights on each feature. "Better" features would learn to have high weights and more influence on the final classification. This approach is outlined next.

2.5 Feature Combination

A logical step to take when using features to identify potentially plagiarized passages in a document is to combine features. Individually, features tend to yield mediocre results. However, it is a reasonable assumption that layering multiple features together could produce better plagiarism classifications. There are many different approaches to this problem, and we have implemented a variety of techniques, including neural networks, evolutionary algorithms, genetic programming, and logistic regression. Some of these methods output a set of weights that are directly multiplied with the raw values of features for a given passage. More commonly, the output is a set of weights are are applied to the individual confidences obtained by clustering on each individual feature. The methods with which we experimented are detailed below.

Neural Networks
The neural network approach outputs a set of confidence weights for a given feature set. The process is fairly straightforward. First, the network is trained until convergence or specified number of iterations. Then, the trained network can be used as a classifier, just like k-means or outlier detection. We won't discuss how neural networks work, but these are the important facts about the way we trained our neural networks. The input nodes are mapped to confidences of the individual features obtained by classifying using outlier detection. The hidden layer consisted of roughly twice the number of features, and there was a single output node whose values are clamped to [0, 1] when evaluating the network. To train the network, we feed the individual confidences of the features into the network and provide the correct classification for the current passage (1=plagiarized, 0=not plagiarized).

Unfortunately, we weren't able to train any neural networks that were effective. The problem of overfitting to the training data was challenging and was never resolved. Much more work could be done here.

Evolutionary Algorithm
We used an evolutionary algorithm setup to output both raw feature weights and confidence weights for a given feature set. However, no significant results were produced for raw feature weights. Unfortunately, evolving the weights effectively requires a massive amount of computing, so we needed to limit the population size, number of generations, and the quality of the fitness function to ensure that training didn't take days. We used a population size of around fifteen individuals, which was evolved over 100 generations. The genome of an individual was simply an array containing weights. Elements of the genome were mutated by adding a sample from a gaussian distribution. The fitness function is arguably the most critical part of the evolutionary algorithm. We evaluated fitness by computing the area under the ROC curve when evaluating suspect 100 documents in our corpus. Of course, the fitness (area under the ROC curve) is trying to be maximized in our case.

This approach had the same problems as neural networks. They took a long time to train, and overfitting was an obstacle that wasn't solved. There are an overwhelming amount of parameters to tweak in evolutionary algorithms, so much more work could also be done with this approach to combining features.

Genetic Programming
We also tried using a genetic programming library, PyGene, to evolve an evaluation tree that would combine features together in some clever way. PyGene includes a straightforward genetic programming implementation, into which we plugged in feature values as terminals, and math functions (addition, multiplication, exponentiation, etc) as nonterminals. The fitness function for the evolution was the area under the ROC curve for the first 50 documents of our training set. We ran the implementation with a starting population of 100 individuals, for ten generations. We used PyGene's default settings for mutation and crossover rates.

2.6 Intrinsic Results

For brevity's sake, we have only included a small subset of our results. The results table compares different atom types and classification types. As is suggested by these results (and others which we have omitted), we found that using 5000-characters as our atom type and outlier detection as our classification method were most successful. Thus, we used these parameters when working on feature selection and

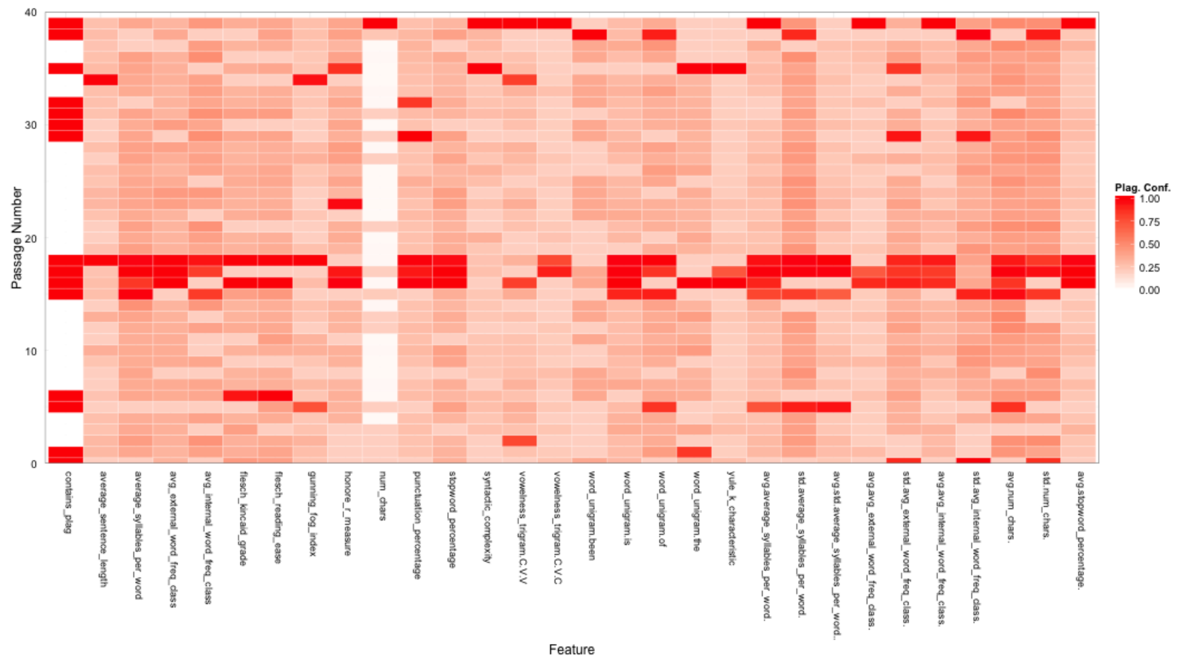


Figure 1: 5000-character atom type and outlier detection on one document. Each row corresponds to a passage. The leftmost column shows the ground truth: red indicates the passage was plagiarized, white that it was not.

feature weighting.

Features Description	Atom Type	Classification	AUC
Best single (Gunning Fog Index)	5000-chars	Outlier	.622
Best single (Avg. Intern. Word Freq. Class)	5000-chars	k-means	.596
Best single (Flesch Reading Ease)	Paragraph	Outlier	.568
Best single (Flesch Reading Ease)	Paragraph	k-means	.575
All	5000-chars	Outlier	.645
All	5000-chars	k-means	.604
All	Paragraph	Outlier	.488
All	Paragraph	k-means	.562

2.7 Discussion

Combining features did not work as well as we had anticipated. Based on the results displayed above there are two questions to consider:

1. How can adding more features make our system worse?

2. Why don't the addition of more features improve our performance

1. It should be noted that only certain atom type/classification techniques combinations (such as outlier detection and paragraph atom type) saw a decreased performance when adding more features. In these cases, the intuition is as follows: some features perform well using paragraphs and outlier detection – these features capture the signal of an author's writing style. If we add in "bad" features, we're essentially adding noise to our classification problem, thereby covering the true signal given by the good features.

2. As is shown in the heatmap above, most features succeed at catching the same plagiarized passages, while most features fail to capture other passages. There are two explanations for this behavior:

first of all, some passages may simply be more difficult to detect. In addition, it's possible that our features are more "similar" than we thought originally, and are measuring similar aspects of writing style.

Consider one "good" feature which catches some subset of plagiarized passages. When we add additional features which also capture those same passages, we don't catch any of the other plagiarized passages. Thus we won't see an improvement in our performance – we're simply more confident that the already-identified passages were plagiarized.

3 Extrinsic Detection

Extrinsic detection is the second method we used, which takes advantage of comparisons with other documents to detect plagiarism. In a real-world detection tool, we might test a document for plagiarism by comparing its text to a vast number of documents from the internet. With the corpus we used, we instead compared our suspicious documents to a large set of source documents (from which all plagiarized sections were taken). Due to time constraints, we cannot simply compare every word of a suspicious document to every word of all source documents – this would take far too long. To address this, we use "fingerprinting" to create a compact representation of our documents to lessen the number of comparisons made, speeding up our computations. We will soon explain what fingerprinting is and the different techniques we used to do it.

Much like in our intrinsic detection, however, we first need to work with a reasonably-sized pieces of text, before we can do fingerprinting and comparisons. This motivated the following framework:

1. Atomization – Deconstruct a document into passages.
2. Fingerprinting – Fingerprint the passage to obtain a compact representation of that passage.
3. Comparison – Use similarity measures to determine similarity between fingerprints obtained from our suspicious document and any source documents.

3.1 Fingerprinting

As previously mentioned, fingerprinting is a method for representing text in a compact form. Specifically, a fingerprint is simply a set of integers, so it logically follows that the process of fingerprinting is the conversion of text to a set of integers. On a fundamental level, fingerprinting addresses the potentially-problematic string matching problem. Given some suspicious passage, one could imagine a scheme in which parts of the passage are compared against parts of documents in an external corpus. This would work perfectly if plagiarism only came in the form of blatant copy/pasting. When word rearrangements and synonym replacements are thrown into the mix, naive string matching won't work as well.

By identifying individual parts of a passage, rather than the whole passage, fingerprinting removes the notion of word order in a passage. This means that it should be able to detect word reorderings along with synonym replacements, since most of the original passage is preserved. Note that fingerprinting is highly reliant on the preservation of some of the original passage. Fingerprinting will not solve the problem of complete paraphrasing, since none of the original passage would be left intact. Hoad and Zobel [4] present a thorough overview of these methods, and we have based much of our work on theirs.

There are two steps to generating a fingerprint for a given passage.

1. Select a subset of n-grams from the passage
2. Generate a hash value for each n-gram

An n-gram is simply a sequence of n words. For example, "Dursley went to" is an example of a 3-gram. We implemented four different n-gram selection algorithms, and each is attempting to identify the n-grams in a passage that are potentially more significant than other n-grams in the passage. Each algorithm also makes the important tradeoff on the size of the fingerprint. On one hand, imagine using every possible n-gram in a passage. The fingerprint will be as big as the original passage, so the fingerprint is actually no less compact than the original passage. Therefore, this will slow down the comparison with other fingerprints. On the other hand, imagine only using a single n-gram. The fingerprint representation of the passage will be extremely compact! Comparisons will be fast, but very little information about the original passage is stored in the fingerprint, which is obviously a problem.

Once a set of n-grams for a passage has been generated, each n-gram is run through a hash function, which produces an integer. For example, we might have h(“Dursley went to”) = 4235. The details of the actual hash function aren’t important, but the values generated by the hash function are modded by a large number, so there is a limit on the highest possible value.

Each of the four n-gram selection algorithms is described below using the following sentence as the input passage:

“Mr. Dursley always sat with his back to the window in his office on the ninth floor.”

Full Fingerprinting
Full fingerprinting is a very naive method of generating a fingerprint. It creates a large fingerprint that is essentially as large as the number of words in the original passage. The algorithm is simple. Simply select all possible n-grams in a passage. If $n = 3$, the resulting fingerprint for the example passage will be the following:

3-grams = {“Mr. Dursley always”, “Dursley always sat”, . . . , “the ninth floor”}

***k*th-in-sentence**
This is the simplest n-gram selection method. Simply split the passage into sentence. Then, for each sentence, select the n-gram that is centered around the kth word in the sentence. Using $k = 5$, $n = 3$ with the example sentence yields the following 3-grams:

3-grams = {“with his back”}

Note that kth-in-sentence yields very small fingerprints, so it results in the fastest running time. Although, it doesn’t store much information from the original passage, so other selection methods yield better performance.

Anchor Selection
Anchor selection aims to be clever about which n-grams it selects. Anchors are defined as 2-character strings, such as “th” or “ma”. For a given set of anchors and a given passage, the algorithm identifies all words that contain any of the anchors and centers n-grams around them. This selection method is highly dependent on the anchors that are used. If extremely common anchors are used, such as “th”, then too many n-grams will be selected, which results in a large fingerprint size. Of course, if rare anchors are used, such as “zi”, then very few n-grams will be selected, resulting in miniscule fingerprints.

By analyzing anchor frequencies in Project Gutenberg, we chose a set of ten anchors with moderately high frequency. Here is the set:

anchors = {“ul”, “ay”, “oo”, “yo”, “si”, “ca”, “am”, “ie”, “mo”, “rt”, }

As you can see, the example sentence contains two instances of these anchors.

“Mr. Dursley always sat with his back to the window in his office on the ninth floor.”

Using 3-grams, we’ll center a 3-gram around each of the words that contain an anchor. In this example, the resulting set of 3-grams is

3-grams = {“Dursley always sat”, “the ninth floor”}

Winnowing
Winnowing is the most complex n-gram selection method we implemented, and it is based directly off of the work of Schleimer, Wilkerson, and Aiken[5]. Like anchor selection, it aims to generate n-grams around significant parts of a passage. However, the significance some part of a sentence is determined by the hash function, not the actual language being used in the passage. Furthermore, winnowing works at the character level, rather than the word level. Specifically, it works with k-grams, which are simply k contiguous characters in a passage.

The algorithm works as follows:

1. Generate array a of all k-grams
2. Move a sliding window of width w across a. Each time you move the window, select the k-gram in the window with the smallest hash value.

Walking through this algorithm on the example sentence is not important enough to illustrate this algorithm. One could imagine the results being something like this:

$$\text{3-grams} = \{\text{“Dur”}, \text{“ays”}, \text{“bac”}, \text{“ind”}, \text{“off”}, \text{“the”}, \text{“thf”}\}$$

3.2 Fingerprint Similarity Measures

After generating fingerprints for passages, we need a way to compare the fingerprints to determine whether or not plagiarism might be present. Recall the fact that fingerprints are sets of integers. This means that we can use set similarity measures to represent the confidence of plagiarism between some suspicious passage and a potential source passage. Ideally, a similarity of 1 will indicate 100% confidence of plagiarism, and a similarity of 0 will indicate 0% confidence of plagiarism. We implemented and tested two different set similarity measures, which are described below.

Jaccard Similarity
Jaccard similarity is a very common set similarity measure that is used in a wide variety of applications. It is defined as

$$jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where A is the suspect fingerprint and B is the source fingerprint.

Containment Similarity
Containment similarity is not a standard set similarity measure, but it has an interesting property that seemed like it could be more effective than jaccard similarity. Containment similarity is nearly identical to jaccard similarity, except the denominator is only the number of elements in the suspect fingerprint. Again, let A be the suspect fingerprint and B be the source fingerprint.

$$containment(A, B) = \frac{|A \cap B|}{|A|}$$

The interesting property of containment similarity is that if a suspect passage is plagiarized from a small portion of some source passage, containment similarity will yield a very high similarity. However, jaccard similarity will yield a much smaller similarity in this case. Therefore, we hypothesized that containment similarity would do a better job at generating confidences of plagiarism. From our tests, our hypothesis was rejected because both similarity measures performed equally.

3.3 Database

Testing both the intrinsic and extrinsic parts of our tool required the use of a database. We used a Postgres database for this purpose.

On the intrinsic side, we stored feature values in the database to allow for easy reuse. We also wanted to get results for thousands of documents for any given set of parameters, and extracting the feature vectors for all of these documents took some time. By storing the feature values in a database, we could easily try different combinations of features and different clustering methods and such without having to recalculate the features.

On the extrinsic side, we used a database to store passage fingerprints. We used the database as a sort of index to look up potential source passages by hashes. To avoid calculating the similarity between every suspect passage and every source passage, we used the database to retrieve only passages whose fingerprints overlapped with a given suspect passage by at least one hash. The database had the following structure:

```
CREATE TABLE documents (
    did         serial,
    name        text,
    path        text,
    xml_path    text,
    is_source   boolean
);

CREATE TABLE passages (
    pid         serial,
    did         integer,
    atom_num    integer,
    atom_type   text
);
```

```
CREATE TABLE methods (
    mid         serial,
    atom_type   text,
    method_name text,
    n           integer,
    k           integer,
    hash_size   integer
);

CREATE TABLE hashes (
    is_source   boolean,
    pid         integer,
    mid         integer,
    hash_value  integer
);
```

The hashes table made it easy to retrieve overlapping fingerprints. We simply used the following select statement:

```
SELECT pid FROM hashes WHERE
     hash_value = <my_val> AND mid = <my_mid> AND is_source = 't';
```

3.4 Extrinsic Results

There are many parameters to play around with when testing our extrinsic system. One can change the fingerprinting method, the comparison method, hash length, atom type. One thing that we found was that 5000-character passages performed better than paragraph passages:

Fingerprint Method	Similarity Measure	Atom Type	AUC
full	jaccard	5000-chars	.95
full	jaccard	paragraph	.90

We also found that full fingerprinting gave us the best ROC AUC. However, anchor fingerprinting, which is much faster than full fingerprinting, does not perform worse by a very large margin:

Fingerprint Method	Similarity Measure	Atom Type	AUC
full	jaccard	5000-chars	.95
winnow-k	jaccard	5000-chars	.94
anchor	jaccard	5000-chars	.92
kth in sent	jaccard	5000-chars	.89

Additionally, the two similarity measures almost always result in the same ROC AUC:

Fingerprint Method	Similarity Measure	Atom Type	AUC
full	jaccard	5000-chars	.95
full	containment	5000-chars	.95

4 Combining Intrinsic & Extrinsic

After implementing both intrinsic and extrinsic detection schemes, we were interested to see whether it might be possible to combine the two approaches and get better results than either could by itself. We approached this task in a fairly simple way–we classified each passage in a suspicious document using both intrinsic and extrinsic detection, and then looked for a clever way to combine the two resulting confidences of plagiarism. We tried several ways of combining the two confidence values, but it turned out that the simple arithmetic mean garnered the best results.

The intuition behind this approach of combining confidences was that if one detection scheme completely missed a plagiarized passage, the other scheme might not. Similarly, if one scheme very wrongly accused a particular passage of plagiarism, the other scheme might balance out the incorrect accusation.

We found that our results for this combination method depended heavily on how well intrinsic and extrinsic detection did on their own. While intrinsic never performed incredibly well, extrinsic could get near-perfect results. When extrinsic had an area under the curve measure of .95 or .97 while intrinsic was around .6 or .65, combining the two schemes did not improve our results. But under different conditions, when extrinsic was not as dominant, the combination was able to significantly improve the area under the curve measure by .05 or more.

While this analysis didn't give us anything conclusive about the results of combining intrinsic and extrinsic, it hints that if we were to actually attempt to detect plagiarism in the real world (where extrinsic detection might not have access to all the source documents it requires), combining both approaches might be a very beneficial technique.

5 Future Work

Given more time, we would focus our work primarily on improving our intrinsic detection tool. While we were unable to find an effective method for combining features, there are still a number of approaches worth considering. More specifically, ensemble learning is a natural next step. If we consider each feature as an independent classifier, we could use various ensemble learning methods to combine these independent classifiers into one single classifier. On a similar note, we could investigate better combination methods designed specifically for one-class classifiers. While we use a simple Naive Bayes approach to combining one-class classifiers, Tax [6] suggests a number of alternatives.

We would also spend more time implementing and researching new features. In particular, we would implement features that measure different aspects of a writer's style. While the majority of our features measure some sort of "complexity" of an author's writing, there are a number of other possibilities. Sentiment analysis is one example: for a given passage, how "positive" or "negative" is the mood of the author's writing? "Recency" of an author's writing would also be an interesting feature that could potentially capture the era of a writer. For example, such a feature could answer the question: "what was the average year that the words from a given passage were added to the Merriam-Webster dictionary?"

6 Appendix: Features

```
average_sentence_length
average_syllables_per_word
avg_external_word_freq_class
avg_internal_word_freq_class
evolved_feature_five
evolved_feature_four
evolved_feature_six
evolved_feature_three
evolved_feature_two
flesch_kincaid_grade
flesch_reading_ease
gunning_fog_index
honore_r_measure
num_chars
punctuation_percentage
stopword_percentage
syntactic_complexity
syntactic_complexity_average
yule_k_characteristic
avg(average_syllables_per_word)
std(average_syllables_per_word)
word_unigram,is
word_unigram,of
avg(avg(average_syllables_per_word))
avg(std(average_syllables_per_word))
avg(avg_external_word_freq_class)
std(avg_external_word_freq_class)
avg(avg(avg_external_word_freq_class))
avg(std(avg_external_word_freq_class))
avg(avg_internal_word_freq_class)
std(avg_internal_word_freq_class)
avg(avg(avg_internal_word_freq_class))
avg(std(avg_internal_word_freq_class))
avg(num_chars)
std(num_chars)
avg(avg(num_chars))
avg(std(num_chars))
avg(stopword_percentage)
std(stopword_percentage)
avg(avg(stopword_percentage))
avg(std(stopword_percentage))
avg(syntactic_complexity)
std(syntactic_complexity)
avg(avg(syntactic_complexity))
word_unigram,been
word_unigram,the
avg(std(syntactic_complexity))
avg(average_sentence_length)
std(average_sentence_length)
avg(syntactic_complexity_average)
std(syntactic_complexity_average)
pos_trigram,NN,VB,NN
pos_trigram,NN,NN,VB
pos_trigram,VB,NN,NN
pos_trigram,NN,IN,NP
pos_trigram,NN,NN,CC
pos_trigram,NNS,IN,DT
pos_trigram,DT,NNS,IN
pos_trigram,VB,NN,VB
pos_trigram,DT,NN,IN
pos_trigram,NN,NN,NN
pos_trigram,NN,IN,DT
pos_trigram,NN,IN,NN
pos_trigram,VB,IN,DT
vowelness_trigram,C,V,C
vowelness_trigram,C,V,V
vowelness_trigram,V,V,C
vowelness_trigram,V,V,V
```

References

[1] Martin Potthast Benno Stein Andreas Eiselt and Alberto Barrón-Cedeno Paolo Rosso. Overview of the 1st international competition on plagiarism detection. In *3rd PAN WORKSHOP. UNCOVERING PLAGIARISM, AUTHORSHIP AND SOCIAL SOFTWARE MISUSE*, page 1.

[2] Benno Stein, Nedim Lipka, and Peter Prettenhofer. Intrinsic plagiarism analysis. *Language Resources and Evaluation*, 45(1):63–82, 2011.

[3] Benedikt Szmrecsanyi. On operationalizing syntactic complexity. *Jadt-04*, 2:1032–1039, 2004.

[4] Timothy C Hoad and Justin Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American society for information science and technology*, 54(3):203–215, 2003.

[5] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[6] David MJ Tax. One-class classification. 2001.